USING AUTOMATIC ABSTRACTION FOR PROBLEM-SOLVING AND LEARNING

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

By
Amy Unruh
May 1993

© Copyright 1993 by Amy Unruh
All Rights Reserved

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of Doctor of Philosophy.

Paul Rosenbloom
(Principal Adviser)

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of Doctor of Philosophy.

Jean-Claude Latombe

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of Doctor of Philosophy.

Richard M. Keller
(NASA Ames Research Center)

Approved for the University Committee on Graduate Studies:

Abstract

Abstraction has proven to be a powerful tool for controlling the combinatorics of a problem-solving search. It is also of critical importance for learning systems. This research develops a set of abstraction techniques which provide a problem solver with a domain-independent *weak method* for abstraction. The method allows the problem solver to: (1) automatically determine when to abstract; (2) automatically determine what to abstract, and dynamically create abstract problem spaces from the original domain spaces; and (3) provides the problem solver with an integrated model of abstract problem-solving and learning.

The abstraction method has been implemented and empirically evaluated. It has been shown to: reduce planning time, while still yielding solutions of acceptable quality; reduce learning time; and increase the effectiveness of learned rules by enabling them to transfer to a wider range of situations.

The core idea underlying the abstraction techniques is that abstraction can arise as an obviation response to impasses in planning. This basic idea is used to reduce the amount of effort required to perform look-ahead searches during problem solving (searches performed in service of a control decision, during which the available options are explored and evaluated), by performing abstract search in problem spaces which are dynamically and automatically abstracted from the ground spaces during search. New search control rules are learned based on the abstract searches; they constitute an abstract plan the problem solver can use in future situations, and are used to produce an emergent multi-level abstraction behavior.

Although this basic abstraction method is broadly applicable, it is too weak and does not yield good performance in all of the domains to which it is applied. In response to this, several domain-independent method increments have been developed to strengthen the method; added to the basic abstraction method, they have succeeded in making tractable a number of problems which were intractable with both non-abstract problem-solving and the simpler weak abstraction method. The two primary method increments are called

assumption counting and *iterative abstraction*.

Assumption counting involves adding a component to the plan evaluation function that counts the number of times the ground domain theory was reformulated before a solution was reached. This is a measure – though not an exact one – of the amount of instantiation that will be required of the abstract plan, and enables abstract detection of interactions between subgoals.

Iterative abstraction can be viewed as a search through a space of plans at varying levels of abstraction. It uses a heuristic which suggests that in the absence of more specific knowledge, a useful level of abstraction for a given control decision during problem solving is that at which one of the choices at the decision appears clearly the best. Implementation of this situation-dependent heuristic enables a unique approach to abstraction creation, during which the problem solver combines selection and synthesis by experimenting with successively less abstract versions of a situation in an effort to estimate the most abstract (hence cheapest) level of description at which useful decision-making can still occur for a situation. With the iterative abstraction method increment, more effort is spent in searching for the initial abstract plan, so as to increase the chances of being able to effectively and efficiently implement it.

Using iterative abstraction, upon making a decision about the level of abstraction it considers appropriate for a particular situation, the system learns plan fragments for the situation at that level of abstraction. Thus, the system accumulates plans containing information at multiple abstraction levels. In new situations, the context determines the level of abstraction of the plans used.

Acknowledgements

I would like to thank:

My advisor, Paul Rosenbloom, for much support, guidance, and patient reading of drafts.

The other members of my reading committee, Rich Keller and Jean-Claude Latombe, for valuable feedback. Rich's comments on an early draft were particularly useful. I'd also like to thank Richard Fikes for helpful comments.

My friends at KSL, including Rich, Andy, Mike, Pandu, Janet, and Alon, who provided lots of fun times and opportunities to procrastinate, both at and away from the lab.

Bob, Margaret, Grace, and Michelle, for providing a great work environment at KSL, and who worked hard to find the resources for me to stay during the last year or so.

Barbara Hayes-Roth.

The Soar group, especially Gregg for his work with R1-Soar.

Allen Newell, whose immeasurable contributions and enthusiasm for the research process remain an inspiration.

My family, who provided constant encouragement and support.

Paul and Lois, who have been a second family.

And most of all, Moon, who is as relieved as I am that I'm finally finished! His love and support made this research possible.

Support for this research was provided by Hughes Aircraft Company Artificial Intelligence Center, by the Defense Advanced Research Projects Agency (DOD) under

contract number N00039-86C-0033, and by the Defense Advanced Research Projects Agency (DOD) and the Office of Naval Research (ONR) under contract number N00014-89-K-0155.

Contents

Abstract iv

Acknowledgements vi

1 Introduction 1

1.1 Background . . . 1

1.1.1 Classes of Abstractions . . . 3

1.1.2 Proof-Increasing Abstractions . . . 5

1.1.2.1 Multiple Levels of Abstraction . . . 8

1.2 Contributions of the Dissertation . . . 9

1.2.1 General *Weak Method* . . . 9

1.2.2 Automatic Determination of When to Abstract . . . 10

1.2.3 Automatic Determination of What to Abstract . . . 10

1.2.3.1 Context-dependent Selection of Abstraction Level . 12

1.2.4 Integration of Abstract Problem-Solving and Learning . . . 12

1.2.5 Implementation and Empirical Evaluation . . . 13

1.3 Overview of Approach . . . 13

1.3.1 The Basic Abstraction Method . . . 14

1.3.1.1 Soar . . . 14

1.3.1.2 The Context in which Abstraction Occurs . . . 15

1.3.1.3 Basic Abstraction Technique: Precondition Abstraction 17

1.3.2 Using the Abstract Searches . . . 18

1.3.2.1 Learning from Abstract Search 19
1.3.2.2 Abstract Plan Refinement and Repair 20
1.3.3 Finding a useful level of abstraction:
Method Increments 20
1.3.3.1 Assumption Counting 21
1.3.3.2 Iterative Abstraction 21
1.3.3.3 Additional Method Increments 22
1.4 Guide to Thesis 23

2 Overview of Soar 24
2.1 Introduction 24
2.2 Goal-oriented Problem Solving 26
2.3 The Decision Cycle 29
2.4 Memory 31
2.4.1 Operator Representation and Implementation 32
2.4.2 Plan Representation 34
2.5 Learning from Impasse Resolution 35
2.6 Problem-solving Methods and
Default Knowledge 36
2.6.1 Lookahead Search 36
2.6.2 Operator Subgoaling 37
2.6.3 Goal Achievement and Protection 38
2.7 Conclusion 38

3 Creating an Abstract Problem Space 40
3.1 Basic Abstraction Method:
Creating an Abstract Problem Space 41
3.1.1 Non-abstract Search 42
3.1.2 Abstract Search 43
3.2 Specification of Precondition Abstraction
Technique 47

3.3 Propagation of Abstractions:
Guidelines For Problem-Space Design 51
3.4 Completeness of the Abstract Search 52
3.4.1 Completeness of Operator Creation 54
3.4.2 Default Behavior of Problem Solver 55
3.4.3 Non-Restrictive use of Search Control 55
3.5 Guidelines for Abstract Search Utility 56
3.5.1 Factorization 57
3.5.1.1 Example 58
3.5.1.2 Implementation 59
3.5.1.3 Factorization of Operator applications 63
3.5.1.4 Factorization of Object Creation from Object Augmentations 63
3.5.1.5 Factorization of Goal Testing. 65
3.5.2 Access of State Information from Operators 65
3.5.2.1 Approach 69
3.5.2.2 Implementation in Soar 69
3.5.2.3 Discussion 73
3.5.3 Avoidance of Implicit Assumptions about Problem State . . . 75
3.5.3.1 Search Control 76
3.5.3.2 Goal Tests 76
3.5.3.3 Precondition Tests 76
3.6 Impact of the Spatula Abstraction
Techniques on Search Characteristics 77
3.6.1 Branching Factor 78
3.6.2 Solution Length 78
3.6.3 Search Expense 79
3.7 Discussion and Summary 80
3.7.1 Use of Problem Space Design Guidelines 80
3.7.2 Discussion: Operator Representations 81
3.7.3 Summary 83

4 Using Abstract Plans for Problem Solving **84**

4.1 The Abstraction Context Revisited:
Reducing Decision Time 85

4.2 Abstract Search Occurs Only When
Necessary 90

4.3 Learning From Search in the Abstract Space 90

4.3.1 Examples of Abstract Learning 91

4.3.2 Learning MEA Knowledge 97

4.3.3 Inductive Learning using Abstraction 99

4.4 Using Abstract Control Rules: Plans 100

4.4.1 Multi-Level Refinement and Repair of Abstract Plans 101

4.4.2 Context of the Abstraction Refinement 106

4.5 Summary 109

5 Abstraction Method Increments **111**

5.1 Introduction 111

5.2 Assumption counting 113

5.2.1 Example of Assumption Counting 114

5.2.2 Factorization: Operator Precondition Testing 118

5.3 Iterative Abstraction 118

5.3.1 Implementation 123

5.3.2 Examples of Iterative Abstraction 125

5.3.2.1 Eight-Puzzle 128

5.3.3 Learning Multi-Level Abstract Plans With Iterative Abstraction 131

5.3.4 Discussion 132

5.4 Extended Example: Iterative Abstraction
and Assumption Counting 134

5.5 Problem Representation and SPATULA's
Method-Increment Abstractions 141

5.5.1 Problem Representation and Assumption Counting 143

5.5.2 Problem Representation and Iterative Abstraction 144

5.6 Extended Plan Use Method Increment 146
5.6.1 Extended Plan Use: Part 1 146
5.6.2 Extended Plan Use: Part 2 (Conservative Version) 149
5.6.3 Discussion: A Spectrum of Plan Use Methods 150
5.7 Efficiency-Driven Method Increments 151
5.7.1 The Abstraction-Gradient Method Increment 152
5.7.2 Iteration on Goal Conjunct Achievement 156
5.7.2.1 Implementation 158
5.7.2.2 Example 160
5.7.2.3 Factorization of Goal Conjunct Tests 161
5.7.3 Discussion 161
5.8 Discussion: Method Increment Biases 162
5.9 Search Complexity using SPATULA 164
5.9.1 Search to apply an operator 165
5.9.2 *N* goal conjuncts 166
5.10 Summary 168

6 Basic Experimental Results 170

6.1 Introduction 170
6.2 Description of Experimental Domains 171
6.2.1 Eight-Puzzle 172
6.2.1.1 Search Control 172
6.2.1.2 Domain Problem-solving Characteristics 173
6.2.2 Robot Domain 173
6.2.2.1 Search Control 176
6.2.2.2 Domain Problem-solving Characteristics 176
6.2.3 Tower of Hanoi 176
6.2.3.1 Search Control 177
6.2.3.2 Domain Problem-solving Characteristics 177
6.3 General Experimental Methodology 178
6.3.1 Default *Weak* Problem-solving Methods 178

6.3.2 Learning . . . 180
6.3.3 Comparison with Non-abstract Problem-Solving Methods . . . 180
6.3.4 Data Collection . . . 181
6.3.4.1 Solution Quality and Problem-solving efficiency . . . 182
6.4 Results: Basic Abstraction Method . . . 185
6.5 Results: Solution Quality and Problem-Solving Efficiency with SPATULA's Method Increments . . . 186
6.5.1 The Eight Puzzle . . . 187
6.5.2 The Tower of Hanoi . . . 189
6.5.3 Robot Domain . . . 192
6.5.3.1 2-Goal-Conjunct Tasks . . . 192
6.5.3.2 3-Goal-Conjunct Tasks . . . 194
6.5.3.3 4-Goal-Conjunct Tasks . . . 196
6.5.4 Discussion . . . 197
6.5.4.1 Trends in Abstraction Across Increasing Numbers of Goal Conjuncts . . . 198
6.5.4.2 Trends in Abstraction Across Increasing Domain Complexity . . . 200
6.6 Abstract Plan Utility . . . 202
6.6.1 Expense of Building Abstract Rules . . . 202
6.6.2 Plan Transfer . . . 204
6.6.2.1 Within-Task Plan Transfer . . . 205
6.6.2.2 Across-Task Plan Transfer . . . 205
6.6.2.3 Discussion . . . 209
6.7 Summary . . . 210

7 Further Experimental Results **211**
7.1 Introduction . . . 211
7.2 Qualitative Analysis: Iterative Abstraction and Assumption Counting 212

7.2.1 The Eight Puzzle 212
7.2.2 Tower of Hanoi and Robot Domain 214
7.2.2.1 Tower of Hanoi 215
7.2.2.2 Robot Domain 218
7.3 Learning and Plan Utility 219
7.3.1 The Generality of Abstract Plans and Problem-Solving Time . 219
7.3.2 Plan Transfer: Detection and Correction of
Overgeneral Plans 221
7.3.3 Extended-Plan-Use Method Increment 222
7.3.3.1 Tower of Hanoi 223
7.3.3.2 Robot Domain 223
7.3.3.3 Extended Plan Use, Conservative Version 226
7.3.3.4 The Spectrum of Abstract Plan Usage 226
7.4 Impact of Efficiency-Driven Method
Increments 227
7.4.1 Tower of Hanoi 228
7.4.2 Robot Domain 229
7.4.2.1 The Abstraction-Gradient Method Increment 229
7.4.2.2 Goal Achievement Iteration 230
7.4.3 Discussion 232
7.5 The Method Increments and Goal Conjunct Ordering 233
7.6 Summary 234

8 Related Work 237

8.1 Classes of abstractions 238
8.2 Abstract Search Representations 239
8.2.1 State-space planners 240
8.2.2 Plan-space planners 241
8.3 Use of Abstract Search Information 242
8.3.1 Using Abstractions to Produce Admissible Search Evaluations 242
8.3.2 Using Abstraction to Suggest Actions or Search Subgoals . . 243

8.3.2.1 Extent to which the Abstract Plan is Used 244
8.3.2.2 Interpretation of the Abstract Plan 245
8.3.2.3 Context of Abstract Plan Use 246
8.4 Automatic Creation of Abstractions 247
8.4.1 Determination of abstractions prior to search 248
8.4.2 Determination of abstractions during search 251
8.5 Learning Abstract Heuristics 253
8.5.1 Learning from abstract problem-solving 254
8.5.2 Abstracting from more detailed problem-solving 254
8.6 Analyses of Abstraction Properties 256
8.6.1 Aggregation Abstractions 257
8.7 Related search reduction approaches 258

9 Conclusion 260

9.1 Summary of Approach 260
9.2 Review of Contributions and Results 261
9.3 Factors Which Influence SPATULA's Abstractions 263
9.4 Capabilities required to implement SPATULA 266
9.5 Limitations of SPATULA 268
9.5.1 Domain Representations and Method Increment Parameters 268
9.5.2 Deep Manifestation of Plan Refinement Difficulties and Interactions 269
9.5.3 Shallow Operator Subgoals 269
9.5.4 Goal conjunct ordering 269
9.6 Future Work 271
9.6.1 Learning Experiments 271
9.6.1.1 Plan Utility 271
9.6.1.2 Knowledge Acquisition in Knowledge-Poor Domains 272
9.6.1.3 Detection of Over-General Search Control 272
9.6.2 Least-Commitment Planning 273

9.6.3 Extensions to Method Increments 273

A Default Rules: Implementing the Abstraction Methods 275

A.1 Basic Abstraction Method 276

A.1.1 Detection of Search Subgoals 276

A.1.2 Abstracting Preconditions 276

A.2 Method Increment Parameters:
Initialization and Management 278

A.2.1 Task Initialization 278

A.2.2 Initializing at new goals and copying to subgoals 279

A.3 Evaluation Rules 283

A.4 Detection of Search Completion 286

A.5 Method Increments 290

A.5.1 Iterative abstraction and Goal Achievement Iteration 290

A.5.2 Assumption Counting 293

A.5.3 The Abstraction-Gradient Method Increment 294

A.5.4 The Extended Plan Use method increment 295

A.6 Conflict Resolution 296

B Eight-Puzzle Domain 299

B.1 Operator Application Rules 300

B.2 EP tasks 302

C Tower of Hanoi Domain 306

C.1 Operator Application Rules 308

C.2 Tower of Hanoi Tasks 309

D Robot Domain 312

D.1 Operator Application Rules 313

D.2 Operator precondition-testing rules 325

D.3 2-goal-conjunct tasks 332

D.4 3-goal-conjunct tasks 340

D.5 4-goal-conjunct tasks 347

E Experimental Results: Detailed Information **354**

E.1 Solution Quality 354

E.2 Problem-solving Efficiency 359

E.3 Learning 371

E.3.1 The Extended-Plan-Use Method Increment 375

E.3.2 Plan Transfer 384

Bibliography **386**

List of Figures

1.1 Search expense often increases exponentially with search depth. (The dark nodes indicate a solution path in the search tree). 2

1.2 Search using an abstract representation of a problem (the overlaid nodes) can be shallower — as shown here — when abstract operators require fewer steps to achieve an (abstract) effect. Abstraction may also reduce the branching factor of search, because fewer problem details mean fewer choices during search (the encircled nodes show several ground states mapped to an abstract state). 3

1.3 The results of abstract problem-solving can serve to constrain search in more detailed spaces. E.g., abstract states can map to subgoals in the ground space (suggested by the dark circles), and abstract operators can be used to constrain information about which ground-space operator(s) are considered at a search point (suggested by the dark lines). 4

1.4 A simple Robot Domain task. 15

1.5 The Abstraction Context. 16

2.1 Overview of Problem-Solving in Soar 27

2.2 Soar's decision cycles. 30

2.3 Example pseudo-code representation of a `move-tile` operator for a sliding-tile puzzle (some simplifications have been made for explanatory purposes). 33

3.1 A non-abstract application of the `go-through-door` operator. 42

3.2 The corresponding abstract application of the `go-through-door` operator; partial operator application allows propagation of the abstraction. 45

3.3 Pseudo-code representations of rules to test the preconditions of the `go-through-door` operator. For readability, most of the rule variables are set in bold-face, and constants are set in italics. 48

3.4 Templates showing the format of operator application rules. 49

3.5 The rule for abstracting an unmet precondition when it is appropriate to do so (i.e., when *in-abstraction-context*). 50

3.6 An unfactored operator application. 57

3.7 The operator of Figure 3.6, factored. 58

3.8 A new precondition of the go-through-door operator, which checks that the door is open. (Although not shown here, a test for precondition-4 would also be added to the last rule in Figure 3.3, which checks that all preconditions are met). 67

3.9 One representation of an extension to the go-through-door operator application, where as a robot moves through, the door shuts by an amount proportional to the robot's size. Using this representation, difficulties are encountered if `go-through-door-precond-4` is abstracted, since the operator application is aborted if the door appears closed. 67

3.10 Precondition testing rules for the new version of the `go-through-door` operator, revised to facilitate more useful operator application if abstraction occurs. 70

3.11 Representation of an extension to the `go-through-door` operator application, in which a sequence of operators is moved through the door. (After each robot in the sequence is moved through the door, the next one is "popped" from the list, and made to be the operator's current robot). Here, the extended operator is represented using the information-access guideline, such that the operator application is facilitated in an abstract space. 72

3.12 Branching factor of state-space search without and with precondition abstraction. 77

3.13 Summary of guidelines for design of a problem space so that abstraction may effectively occur using SPATULA. 81

4.1 The context in which abstraction takes place during problem-solving. 87
4.2 Terms used in the default rule descriptions of Figures 4.3 and 4.4. . . 88
4.3 Default knowledge for detection of abstraction context. 89
4.4 Default knowledge for initial abstraction of preconditions. This figure is a generalization of the rule shown in Chapter 3, Figure 3.5. 89
4.5 Example robot domain task and operators. 92
4.6 An explanation of goal success for the task of Figure 4.6. Arrows represent explanation dependencies. Circled area and dotted arrows represent information ignored during abstract problem-solving. With non-abstract problem-solving, all of the leaves (leftmost nodes) are conditions of the explanation. With abstract problem-solving, only the unshaded leaves are conditions of the explanation. 93
4.7 Example initial state and goal for Robot Domain task. 94
4.8 A non-abstract rule produced for the example task of Figure 4.7 . . . 95
4.9 The corresponding abstract rule produced for the example task. . . . 96
4.10 Example of MEA knowledge learned during abstract search. 98
4.11 Successive refinement of a plan for a simple task in a robot domain. . 103
4.12 Initial state and goal for the robot domain example of Figure 4.11. . . 104

5.1 Default knowledge for the *assumption counting* method increment. . . 113
5.2 Modifications of existing default evaluation knowledge for *assumption counting* method increment. 115
5.3 Example of utility of assumption counting. 116
5.4 The general algorithm for iterative abstraction. 119
5.5 Iterative abstraction; first iteration level lookahead searches for each of three options. 120
5.6 Iterative abstraction, second iteration level. 122
5.7 Default knowledge for the *iterative abstraction* method increment. . . 124

5.8 Modification of default long-term-memory evaluation knowledge for *iterative abstraction* method increment. 125
5.9 Example of iterative abstraction. 126
5.10 First-level abstract lookahead search for iterative abstraction example. 127
5.11 Second-level abstract lookahead search for iterative abstraction example.127
5.12 Eight-Puzzle lookahead search using iterative abstraction: first level. . 129
5.13 Eight-Puzzle lookahead search using iterative abstraction: second level. 130
5.14 Initial state for a 4-goal-conjunct task in a robot domain. 135
5.15 Initial operator tie generated for the task of Figure 5.14. 136
5.16 Beginning of second-level iteration for the `Open-door(Room-4,Room-5)` operator: achieving the `Open-door` preconditions. The top unshaded box proves to be the best operator sequence. 136
5.17 Result of applying the `Open-door(Room-4,Room-5)` operator at the second level. 138
5.18 Continuance of the second level search starting with the `Open-door(Room-4,Room-5)` operator. Top unshaded box indicates best choice. 139
5.19 Second level abstract plan segments for all operators in initial tie. Top unshaded boxes indicate best choices. 140
5.20 Default knowledge for the first part of the extended plan use method increment. These rules allow the system to reason about the levels of detail encoded by its learned control rules from lookahead sub-searches. 148
5.21 Default knowledge for the second part of the extended plan use method increment. 149
5.22 An illustration of the abstraction-gradient method increment. The initial abstraction level of a search determines how much more abstract it becomes as search becomes deeper. 153
5.23 Default knowledge for *abstraction-gradient* method increment. 154
5.24 The searches of Figure 5.19, with the abstraction-gradient method increment used as well. The same evaluation is obtained with less effort. 155
5.25 Default knowledge for the *goal-achievement-iteration* method increment.159
5.26 Example of bias in abstraction. 162

5.27 Summary of SPATULA's method increments. 169

6.1 A typical task in the Robot Domain, using the "ABStrips" room layout. 174

6.2 The same task as in Figure 6.1, using the more complex room layout. 175

6.3 Eight-puzzle: comparison of search methods. 188

6.4 TOH: average solution quality and problem-solving steps. 190

6.5 2-goal-conjunct tasks in Robot Domain: average solution quality, percentage of tasks completed, and average problem-solving steps. 19 tasks were tested with the original layout. A subset of 14 of these tasks was tested with the complex layout. 193

6.6 3-goal-conjunct tasks in Robot Domain: average solution quality, percentage of tasks completed, and average problem-solving steps. 18 tasks were tested with the original layout. A subset of 9 of these tasks was tested with the complex layout. 195

6.7 4-goal-conjunct tasks in Robot Domain: average solution quality, percentage of tasks completed, and average problem-solving steps. 12 tasks were tested with the original layout. Evidence suggests that the relatively smaller difference in solution quality (as compared to the easier sets of tasks) is spurious, and caused by the increasing intractability of the solutions generated by the first-path searches (only the solutions from the tractable searches are recorded). 197

6.8 Summary of trends across increasing numbers of goal conjuncts in the Robot Domain, with the original room layout. Evidence suggests that the relatively smaller difference in solution quality for the 4-goal tasks is spurious, and caused by the increasing intractability of the solutions generated by the first-path searches (only the solutions from the tractable searches are recorded). 199

6.9 Summary of complexity trends in Robot Domain. 201

6.10 Summary of learning expense for abstract and non-abstract rules. The graph shows average difference as a percentage of non-abstract expense (i.e., $\frac{\text{non-abstract} - \text{abstract}}{\text{non-abstract}}$). 203

6.11 Summary of plan transfer results in the Tower of Hanoi. 207

6.12 Summary of plan transfer results in the Robot Domain, shown as average percent difference from the results obtained without use of the rules from other tasks (i.e., $\frac{\text{non-transfer}-\text{transfer}}{\text{non-transfer}}$). 208

7.1 Easiest-subgoal-first strategy employed with SPATULA in the TOH. . . 216

7.2 Relative problem-solving times of abstract and first-path search in the Robot Domain (i.e., $\frac{\text{abstract}-\text{first-path}}{\text{abstract}}$). 220

7.3 Summary of extended plan use results in Robot Domain. 224

B.1 Example state in eight-puzzle task. The dark square represents the empty spot in the grid, and the white squares represent tiles. 300

C.1 Selected steps from a typical Tower of Hanoi task. 307

E.1 Robot Domain: Solution length for 2-goal-conjunct tasks, original room layout. 355

E.2 Robot Domain: Solution length for 2-goal-conjunct tasks in complex room layout, as compared with the corresponding tasks in the original room layout. Optimal solutions remain the same. Missing entries indicate which tasks were not able to finish. 356

E.3 Robot Domain: Solution length for 3-goal-conjunct tasks, original room layout. Missing entries indicate which tasks did not finish. Only 1 task with best-path search was able to finish. 357

E.4 Robot Domain: Solution length produced by using abstraction for 3-goal-conjunct tasks in complex room layout, as compared with the corresponding tasks in the original room layout. Missing entries indicate which tasks did not finish. Optimal solutions remain the same. . . . 357

E.5 Robot Domain: Solution length for 4-goal-conjunct tasks, original room layout. Missing entries indicate which tasks did not finish. Best-path search was intractable. 358

E.6 Eight-puzzle: problem-solving steps. 359

E.7 Eight-puzzle: Total problem-solving time for a subset of the EP tasks. Tasks marked with '(ga)' used a modified version of goal achievement iteration. 360

E.8 Tower of Hanoi, 4-disk tasks: problem-solving steps. 360

E.9 Tower of Hanoi, 4-disk tasks: problem-solving time in seconds. 361

E.10 Tower of Hanoi: number of tokens per production firing. 361

E.11 Robot Domain: problem-solving steps for 2-goal-conjunct tasks. Tasks which took more than 2600 steps were cut off. 362

E.12 Robot Domain, 2 goal conjuncts: problem-solving time (secs.) Entries are missing for one first-path and one non-abstract best-path search because of recording errors. 363

E.13 Robot Domain, 2 goal conjuncts: tokens per production firing. 364

E.14 Robot Domain: problem-solving steps for 2-goal-conjunct tasks in complex room layout. Blank entries indicate tasks which did not complete. 364

E.15 Robot Domain, 2 goal conjuncts: problem-solving time in complex room layout. Blank entries indicate tasks which did not complete. . . 365

E.16 Robot Domain, 2 goal conjuncts: tokens per production firing in complex room layout. Blank entries indicate tasks which did not complete. 365

E.17 Robot Domain, 3 goal conjuncts: problem-solving steps; original room layout. Non-abstract best-path search was not tractable. Missing entries indicate tasks which did not complete. 366

E.18 Robot Domain, 3 goal conjuncts: problem-solving time (secs.); original room layout. Non-abstract best-path search was not tractable. Missing entries indicate tasks which did not complete. 367

E.19 Robot Domain, 3 goal conjuncts: tokens per production firing; original room layout. 367

E.20 Robot Domain, 3 goal conjuncts: problem-solving steps; complex room layout. Non-abstract best-path search was not tractable. Missing entries indicate tasks which did not complete. 368

E.21 Robot Domain, 3 goal conjuncts: problem-solving time; complex room layout. Non-abstract best-path search was not tractable. Missing entries indicate tasks which did not complete. 368

E.22 Robot Domain: problem-solving steps for 4-goal-conjunct tasks; original room layout. Non-abstract best-path search was not tractable. Missing entries indicate tasks which did not complete. 369

E.23 Robot Domain, 4 goal conjuncts: problem-solving time (secs.); original room layout. Non-abstract best-path search was not tractable. Missing entries indicate tasks which did not complete. Time for "7x" task is probably unreliable, as suggested by tokens changes graph of Figure E.24.369

E.24 Robot Domain, 4 goal conjuncts: total token changes, original room layout. Non-abstract best-path search was not tractable. Missing entries indicate tasks which did not complete. 370

E.25 Robot Domain, 4 goal conjuncts: tokens per production firing. 370

E.26 Tower of Hanoi: chunking expense. 371

E.27 Tower of Hanoi: Number of chunks. 372

E.28 Robot Domain, 2 goal conjuncts: chunking expense. Missing data is due to errors in recording data. 373

E.29 Robot Domain, 2 goal conjuncts: Number of chunks. Missing data is due to errors in recording. 374

E.30 Robot Domain, original layout, 2 goal conjuncts, extended-plan-use method increment: solution length. 375

E.31 Robot Domain, original layout, 3 goal conjuncts, extended-plan-use method increment: solution length. Blank entries indicate tasks which did not finish. 376

E.32 Robot Domain, original layout, 4 goal conjuncts, extended-plan-use method increment: solution length. Blank entries indicate tasks which did not finish. 376

E.33 Robot Domain, original layout, 2 goal conjuncts, extended-plan-use method increment: problem-solving steps. 377

E.34 Robot Domain, original layout, 2 goal conjuncts, extended-plan-use method increment: problem-solving time. 377
E.35 Robot Domain, original layout, 3 goal conjuncts, extended-plan-use method increment: problem-solving steps. Blank entries indicate tasks which did not finish. 378
E.36 Robot Domain, original layout, 3 goal conjuncts, extended-plan-use method increment: problem-solving time. Blank entries indicate tasks which did not finish. 378
E.37 Robot Domain, original layout, 3 goal conjuncts, extended-plan-use method increment: token changes. 379
E.38 Robot Domain, original layout, 4 goal conjuncts, extended-plan-use method increment: problem-solving steps. Blank entries indicate tasks which did not finish. 380
E.39 Robot Domain, original layout, 4 goal conjuncts, extended-plan-use method increment: problem-solving time. Blank entries indicate tasks which did not finish. 381
E.40 Robot Domain, original layout, 4 goal conjuncts, extended-plan-use method increment: token changes. 381
E.41 Robot Domain, original layout, 2 goal conjuncts, extended-plan-use method increment: tokens per production firing. 382
E.42 Robot Domain, original layout, 3 goal conjuncts, extended-plan-use method increment: tokens per production firing. 382
E.43 Robot Domain, original layout, 4 goal conjuncts, extended-plan-use method increment: tokens per production firing. 383
E.44 Robot Domain, 3 goal conjuncts: problem-solving with the extended plan use method in the complex room layout. 383
E.45 Effect of transfer of rules on number of problem-solving steps in the Tower of Hanoi. 384
E.46 Effect of transfer of rules on number of problem-solving steps in the Robot Domain. 385

Chapter 1

Introduction

As the field of AI matures, there has been a growing realization that it is difficult to solve problems of any complexity without abstracting away at times from the full detail of a problem. The general planning problem has, for example, been shown to be intractable [Chapman, 1987]. Abstractions of a task can provide a system with a "general picture" of a problem with less effort than needed to solve the full problem, and thus suggest heuristics about how to solve the original task. Therefore, the generation as well as the use of abstractions is of much interest.

This thesis will describe an integrated set of methods for both creating abstractions and using them during problem solving. Before the method is introduced, however, we will give some background on the different general categories of abstractions, and some of the issues which can arise with their use.

1.1 Background

Informally, an abstraction can be described as a mapping of a representation of a "ground" problem into a more abstract representation of the problem in which some details of the ground problem are removed, such that more than one ground problem can map to the same abstract problem. E.g., ground and abstract problems may be represented as axiomatic systems, in which are specified a language; a set of true statements, or theorems, in that language; and a deductive mechanism

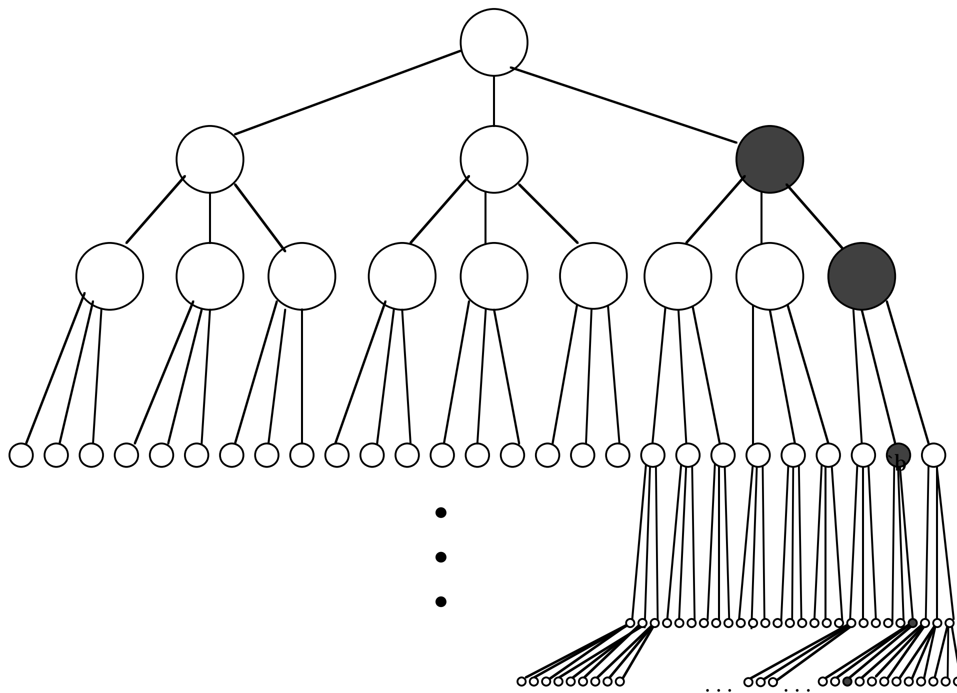


Figure 1.1: Search expense often increases exponentially with search depth. (The dark nodes indicate a solution path in the search tree).

(not necessarily logical deduction) by which other true statements can be generated. An abstraction is then such a mapping from one system to another [Plaisted, 1981; Giunchiglia and Walsh, 1990a]].

The utility of an abstraction mapping is that information obtained by problem-solving in an abstract space can then be used to guide the problem-solving in a corresponding ground space, thus reducing the combinatorics of the ground-space search. For a graphical representation of this process, consider the following set of figures. Figure 1.1 shows what a non-abstract problem search might look like. If the branching factor of the search is relatively constant, then the search expense will on average increase exponentially with search depth.

Using an abstraction mapping, the non-abstract problem representation can be mapped to a more abstract one, and the abstract problem solved using the operations, or theorems, of the abstract representation. If the search depth in the abstract space is shallower (because the abstract operations require fewer steps to achieve an (abstract)

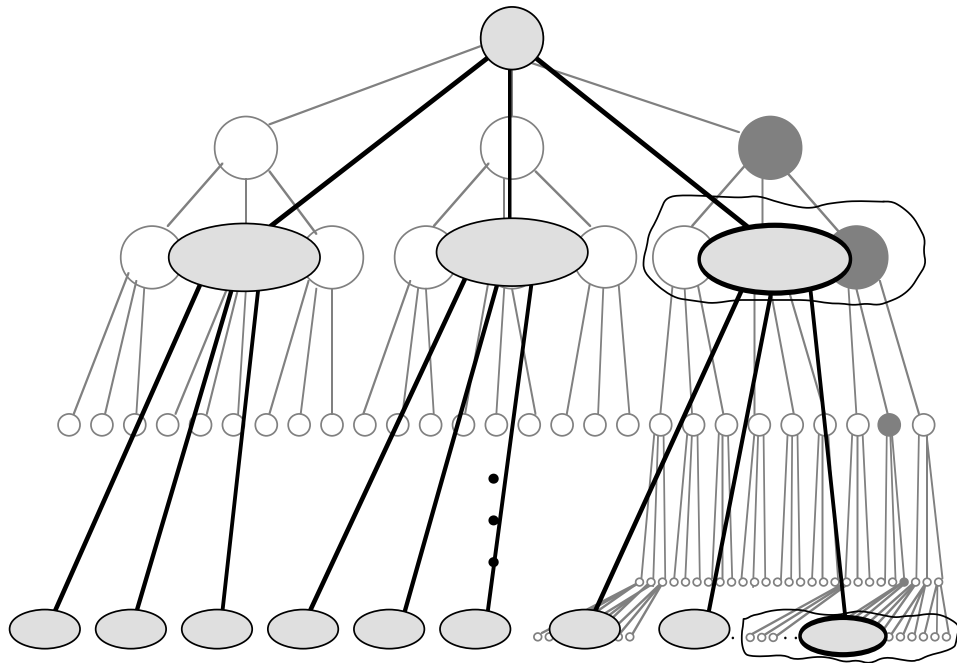


Figure 1.2: Search using an abstract representation of a problem (the overlaid nodes) can be shallower — as shown here — when abstract operators require fewer steps to achieve an (abstract) effect. Abstraction may also reduce the branching factor of search, because fewer problem details mean fewer choices during search (the encircled nodes show several ground states mapped to an abstract state).

effect, as suggested by the overlay in Figure 1.2); or if the branching factor is smaller (because there are fewer problem details to consider), then the abstract search can be less expensive than the ground-space search.

As suggested in Figure 1.3, this less expensive abstract search may then be used to reduce the expense of the non-abstract search, by using, or mapping back, information from the abstract search to constrain the choices made during the non-abstract search. The way in which an abstract search is most usefully interpreted depends not only upon the domain, but upon the type of abstraction.

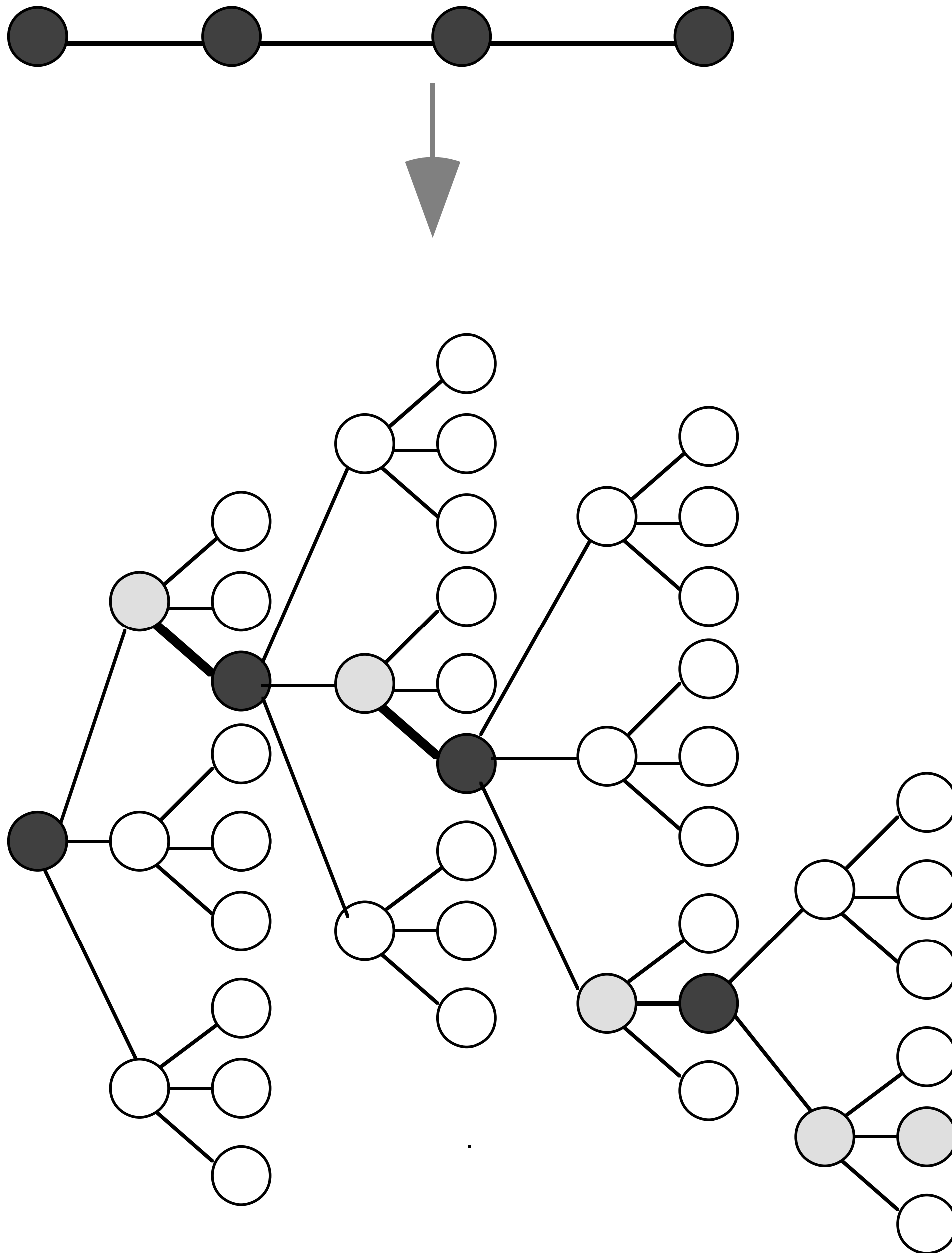


Figure 1.3: The results of abstract problem-solving can serve to constrain search in more detailed spaces. E.g., abstract states can map to subgoals in the ground space (suggested by the dark circles), and abstract operators can be used to constrain information about which ground-space operator(s) are considered at a search point (suggested by the dark lines).

1.1.1 Classes of Abstractions

First, abstraction mappings fall into three general categories. Mappings can be *complete* (the abstraction of any ground theorem, or result, is a theorem in the abstract space) but not necessarily *correct* with respect to the ground space (there may be theorems in the abstract space which do not correspond to any theorem in the ground space). Or, they can be correct but not necessarily complete (all theorems in the abstract space are abstractions of ground-space theorems, but not all ground-space theorems necessarily have corresponding theorems in the abstract space). Alternatively, they may be both complete and correct. [Giunchiglia and Walsh, 1990a] name these three kinds of mappings Theorem-Increasing, Theorem-Decreasing, and Theorem-Constant abstractions, respectively. Equivalently, [Tenenberg, 1988] defines Theorem-Increasing abstractions as having an "upward solution property", and Theorem-Decreasing abstractions as having a "downward solution property". Examples exist of systems which use all types of abstractions; see Chapter 8 for descriptions[1]. Knowledge of which category an abstraction falls into can be used to help interpret the success or failure of an abstract search with respect to its corresponding ground search. E.g., when applying a Theorem-Increasing abstraction to a problem, a proof that there was no solution to the abstracted version of the problem would imply that no non-abstract solution existed as well.

Second, the information available from the abstract search can be used to different extents in the ground space. If the structure of the abstract problem-solving is not expected to bear a resemblance to the structure of the more detailed problem-solving, then it might make sense to make use of just the result, or output, of the abstract search. However, if the structure of the abstract search is expected to bear some similarity to the ground-space search, then information about the intermediate steps of the abstract solution may be used as well. The greater the amount of abstract information usable in the ground space, the greater the extent to which the abstract search can constrain the ground search.

[1] It is also possible to construct abstractions that are both incomplete and incorrect.

1.1.2 Proof-Increasing Abstractions

A particularly useful subclass of Theorem-Increasing, or TI-Abstractions, has been identified. These abstractions have been called Proof-Increasing Abstractions (or PI-Abstractions) [Giunchiglia and Walsh, 1990a; Giunchiglia and Walsh, 1990b], and preserve the structure of proofs. PI-Abstractions are of utility because all of the steps of an abstract proof can be used to guide the production of a more detailed proof[2]; thus, they allow a large amount of abstract information to be used in the ground space.

To define PI-Abstractions, Giunchiglia and Walsh first introduce a definition of *tree subsumption*. Informally, an abstract proof tree Π_2 subsumes a ground proof tree Π_1 if all wffs in Π_2 are in Π_1, with the same global ordering (the "below", "above", and "adjacency" relations are maintained). Tree subsumption is a monotonicity property on the depth, the number of formulae occurrences, the ordering of wffs, and the branches of the proof trees.

Let f be an abstraction mapping from a non-abstract system to an abstract system, Π be a proof tree constructed in the non-abstract system, and $f(\Pi)$ be the tree constructed by applying f to every node in Π. An abstraction is then classified as a PI-Abstraction iff for any non-abstract proof Π_1 of a theorem φ in the non-abstract system, there exists a proof Π_2 of $f(\varphi)$ in the abstract system such that Π_2 subsumes $f(\Pi_1)$. This property holds across multiple levels of abstraction (an abstraction hierarchy), when all levels of the hierarchy are PI-Abstractions. As an example, the abstractions produced by ABStrips [Sacerdoti, 1974] (further described in chapter 8) are PI-Abstractions.

Similarly, Knoblock [1991], defines a class of abstraction mappings in which a set of literals of a non-abstract language are removed from the language, states, and operators of the non-abstract system to produce an abstract system. He then shows that for this class of abstraction mappings, the existence of a ground-level proof implies the existence of an abstract proof that can be refined into a ground-level solution while leaving the established literals in the abstract plan unchanged (again,

[2]The term "proof" does not necessarily refer to the result of using logical deduction, but is used more generally to refer to the result of using any deductive mechanism.

this result holds for abstraction hierarchies as well as for a single abstract level). These abstractions are then said to have the *monotonicity property*, and in fact are PI-Abstractions[3].

To make use of the structure of a PI-abstract proof of a problem, the abstract proof must first be "unabstracted". This can be done by constructing a ground-level "skeleton proof" or schema whose abstraction equals the abstract proof. There may be more than one such schema, but representations can be used which allow the ground-level system to reason about the range of possible unabstractions. For example, an abstract state may map back to more than one non-abstract state, but may be represented in the ground system as a subgoal which is satisfied when one such non-abstract state is reached. Or, the operators used in the abstract proof may provide information about which ground-level operators or sets of operators to consider at various points in the ground-level search. Figure 1.3 in fact shows a PI-Abstraction and suggests both these processes.

The problem solver can then attempt to refine, or fill in, the skeleton proof; the skeleton proof serves to guide and constrain the way in which the problem solver will try to find a ground-level solution to the problem, therefore reducing search. Note that because PI-Abstractions are Theorem-Increasing abstractions, it is not guaranteed that a refinement will exist for every abstract proof. However, it is guaranteed that a refinement will exist for *some* abstract proof, such that a ground solution may be constructed by monotonically adding to the abstract proof without deleting or "moving" any of the abstract proof steps.

The abstract proof provides ordering information about some of the components in the ground-level solution, but does not in itself specify how the problem solver will use the information. For example, if the states of the abstract proof were to be mapped to a series of ground-space subgoals, this information might be interpreted by the problem solver as specifying an ordering on subgoal *achievement*, but still allowing interleaved work on more than one subgoal at once. Alternatively, the series of subgoals might be interpreted by the problem solver as a series of independent subproblems,

[3]Knoblock further describes a subclass of such mappings which have a stronger property; that of *ordered monotonicity*. This work is discussed in Chapter 8.

each achieved before the next is begun. This is the approach taken by many systems (see Chapter 8). The assumption of subgoal independence allows a best-case exponential reduction in search complexity from the abstract to the ground space [Korf, 1987; Knoblock, 1991]. However, with PI-Abstractions it is *not* guaranteed, for an arbitrary domain and abstraction, that a ground solution is reachable by assuming subgoal independence; thus this is not always a reasonable assumption.

The interpretation of an abstract solution in the less abstract space (e.g., whether or not the subgoals generated by the abstraction are treated as independent) is often viewed as being part of the abstraction method. However, this need not be the case. As will be illustrated in this thesis, the use of abstraction may be implemented such that the refinement of an abstract proof is influenced by the various problem-solving methods that a problem solver uses for a task; thus, a system may implement different refinement approaches at different times, as appropriate in different domains.

1.1.2.1 Multiple Levels of Abstraction

Thus far we have discussed the use of just one abstraction level, but in fact multiple levels of abstraction may be used for a problem, with each level guiding the refinement of more detailed level(s). The use of multiple abstraction levels can constrain the total search effort to a greater extent than is possible with just one level [Korf, 1987]. With PI-Abstractions, if each abstract solution step is interpreted as defining an independent subgoal for more detailed problem solving, and with optimal abstraction mapping characteristics – e.g., with respect to the number of abstraction levels and the amount of refinement required at each abstraction level – it is theoretically possible to reduce the search complexity to linear in the length of the ground-level solution [Korf, 1987; Knoblock, 1991]. In practice this rarely occurs, except in domains of extreme regularity.

Many existing abstraction problem solvers — problem solvers which use abstract planning to guide more detailed problem-solving — in fact use some form of PI-Abstraction. Often, such abstractions are called "approximation" abstractions. Precondition relaxation abstractions (in which certain preconditions of operators in the

non-abstract system are removed from the operator descriptions in the abstract system) are one type of commonly-used PI-Abstraction, as are abstractions in which literals are removed from the states and operators of a problem space.

1.2 Contributions of the Dissertation

In this thesis, we have developed a problem-solving method for automatic abstraction called SPATULA, which creates and uses PI-Abstractions. SPATULA is implemented in (and is motivated by) the Soar general problem-solving architecture [Laird *et al.*, 1987a; Rosenbloom *et al.*, 1991a]. Although some systems hard-wire particular behaviors into their architectures, with Soar all problem-solving methods are determined by the knowledge in Soar's memory– different knowledge can produce different problem-solving methods. Consistently with this approach, SPATULA does not require any modifications to the Soar architecture, but rather is implemented by providing Soar with knowledge about how to abstract. That is, the problem solver, using the knowledge provided by SPATULA, acquires the capability to create and use abstractions.

In this section, the contributions of the thesis are first introduced. Then, in Section 1.3, an overview is given of the approach used to achieve the contributions and produce SPATULA's abstraction behavior.

1.2.1 General *Weak Method*

SPATULA is designed to be a *general weak problem-solving method* for abstraction [Laird *et al.*, 1987b].

Weak methods are a class of problem-solving methods which require little domain-specific knowledge about a task. Thus they are applicable in most problem domains in the absence of (or in conjunction with) more domain-specific knowledge. By definition, although weak methods are useful in a wide range of situations, their limited operational demands sometimes translate into poor performance; they are not guaranteed to work well in every domain in which they are applicable. *Hill climbing* [Nilsson, 1980] is an example of such a weak method; with hill climbing, the problem

solver makes a next move based on a local estimation of the situation. While this may not always be an optimal strategy, it is one which may be used for many problems, and can often allow a problem solver to make progress when more domain-specific methods are not available.

SPATULA, as a general weak method, has been designed to:

- increase problem-solving efficiency;
- produce good solutions[4];
- allow a system to learn more easily from its problem solving — that is, require less effort to build new knowledge about its tasks;
- and increase the transfer of learned knowledge to new situations.

By specifying when and how to abstract, SPATULA allows Soar to become a system with an integrated framework for learning, using, refining, and repairing abstract plans, and to accomplish the goals above via this framework. Because it is a weak method, SPATULA may not meet these goals for every task to which it is applied. However, amortized over all the problem instances to which the method is applied, we would like these goals to hold.

Using SPATULA, the problem solver exhibits the following characteristics.

1.2.2 Automatic Determination of When to Abstract

Using SPATULA, the problem solver only abstracts when necessary. Most other abstraction planners use abstraction in all situations — that is, given a task, problem-solving is always first carried out in an abstract space and then the abstract results used for more detailed problem-solving. However, in some situations a problem solver may know exactly what to do and may not need to use abstraction planning to provide search heuristics. In these cases its use would only generate unnecessary computation. Therefore, the problem solver should use abstraction planning only when

[4]The definition of "good" will be elaborated upon in the following chapters; however, the solutions produced using the abstraction method should at the least be better than those produced when the problem solver makes a choice at a control impasse based only upon its preexisting search control knowledge about the situation.

existing search control is not sufficient to allow it to make progress on a task; there is no need for it to abstract in situations when it already knows what to do next.

1.2.3 Automatic Determination of What to Abstract

Most previous abstraction research has used abstraction mappings which are generated by a human designer. Such abstractions can be very effective, since much work can be done to tailor them so that they focus on the most important aspects of a problem. However, because they must be tailored to a domain by a human designer, a dependency upon them is restrictive. Clearly, if a problem-solver is able to generate its own useful abstractions, this will be an advantage in situations for which abstractions have not been provided.

To be a weak method, SPATULA must be able to provide information to the problem solver about how to automatically construct its own abstractions; the system should not be dependent upon abstraction information from a human designer. In addition, as a weak method, SPATULA should depend as little as possible upon specific descriptive information about its domains, since such information may not always be available. The greater the extent to which this is the case, the wider the range of situations in which the method will be usable. This dissertation describes an automatic abstraction method which constructs PI-Abstractions – using precondition relaxation – from a domain's original non-abstract problem spaces. It does not require that abstract problem spaces be provided, or that changes be made to the problem solver's architecture.

There has been a recent and growing body of work towards the automatic generation of abstractions (some such systems are discussed in Chapter 8). The research has primarily been focused on developing abstractions by performing pre-task analyses of a problem-solving domain; that is, before problem solving is initiated. Such analyses make certain demands on the system's knowledge about its domains and usually require a new "module" attached to the problem solver to perform the analysis. For example, some automatic abstraction methods, such as [Christensen, 1990; Benjamin, 1989; Ellman, 1988; Knoblock, 1989; Tenenberg, 1988], derive abstractions before problem solving for a task begins, based on a description of the domain and

task.

The abstraction method described here draws on a different source of knowledge, that of difficulties encountered while problem solving, and enables a unique approach to dynamically generated abstract problem spaces and abstraction levels; it does not assume access to declarative descriptions of task domains. This thesis does not claim that SPATULA's approach is to be a replacement for more knowledge-intensive methods of generating abstractions, but that it serves a different purpose. Because SPATULA's source of knowledge is obtained *during* problem solving, it may be available when more declarative descriptions of the domain problem spaces are not (or when there is not an opportunity to do pre-task analysis).

1.2.3.1 Context-dependent Selection of Abstraction Level

An important component of SPATULA's weak-method approach is the implementation of a heuristic which the problem solver uses to help it dynamically determine the abstractions used in a given situation. This heuristic (called *iterative abstraction*) suggests that in the absence of more specific knowledge, a useful level of abstraction for a given control decision during problem solving is that at which one of the choices at the decision appears clearly the best. Implementation of this situation-dependent heuristic enables a unique approach to abstraction creation, during which the problem solver experiments with successively more detailed versions of a situation in an effort to estimate the most abstract (hence cheapest) level of description at which useful decision-making can still occur for a situation.

1.2.4 Integration of Abstract Problem-Solving and Learning

Until recently, abstraction planning research and learning research have been almost entirely disassociated; there has been very little exploration of the ways in which a system can learn *from* its abstract problem solving, and then use the information that it has learned in new situations. However, it is easy to see the utility of such an ability. Such an approach has interest not only for efficiency reasons (abstract plans

don't have to be re-discovered in the future), but because of the *kind* of learning that may occur.

Soar is able to learn from its problem-solving experiences; in fact, in Soar all learning occurs as a result of problem-solving activity. Soar's capabilities allow SPATULA to utilize a uniquely integrated approach to learning about and using its automatically generated abstractions.

Learning produces abstract plans which are *inductively generalized* from the ground-level domain theory; thus SPATULA provides the system with the capacity for *knowledge-level* learning [Rosenbloom *et al.*, 1987; Rosenbloom *et al.*, 1991b; Dietterich, 1986]. The cost of building abstract plans is less than for their non-abstract counterparts, and the abstract plans are applicable in a wider range of situations.

The abstraction process creates abstract plans at multiple levels of abstraction. The level of abstraction used to guide a given decision is context-sensitive; plans from different levels of abstraction may be applied simultaneously to different aspects of the same situation as appropriate. The abstract plans are indexed, refined, and repaired in a situated manner, producing an emergent multi-level abstraction behavior.

1.2.5 Implementation and Empirical Evaluation

One goal of this research was to examine how much "mileage" could be obtained by the use of such a weak-method approach — what could be accomplished with minimal declarative information with which to analyze the domains. The SPATULA abstraction techniques, implemented in Soar, were empirically evaluated in three distinct domains. The experimental results showed that SPATULA was able to meet the goals of its weak-method approach.

1.3 Overview of Approach

This section gives an overview of SPATULA, and the way in which it achieves the contributions presented in the previous section. This description is in several parts. The first part outlines a technique for dynamically and automatically abstracting a

domain's problem spaces in a domain-independent fashion, and then describes the way in which this basic abstraction technique may be used by a problem solver to produce an integrated model of abstraction use. Next, we describe several augmentations to the basic abstraction method, called *method increments*. The method increments — in the form of additional knowledge given to the problem solver — modify and increase the utility of the basic abstraction method by obtaining more information from the problem-solving context, while allowing the abstraction process to remain domain-independent.

SPATULA, as a set of techniques, encompasses both the basic abstraction method and its method increments. (However, as will be seen below, it is not required that all of SPATULA's capabilities be used simultaneously by a problem solver.)

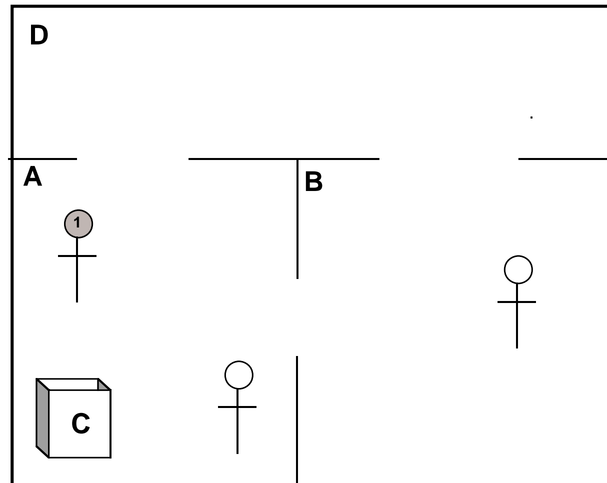
1.3.1 The Basic Abstraction Method

1.3.1.1 Soar

As discussed above, SPATULA is implemented in the Soar problem-solving architecture [Laird *et al.*, 1987a; Rosenbloom *et al.*, 1991a], and motivated by its capabilities. In Soar, problems are solved by search in *problem spaces*, in which operators are applied to yield new states. Long-term knowledge is represented in the form of rules. Inadequate knowledge about how to proceed in a situation produces an *impasse*, which the system tries to resolve by problem solving in a subgoal generated to resolve the impasse. Further subgoals may be recursively generated. A *control impasse* is a particular type of impasse which occurs if the problem solver is in a situation where more than one option exists for its next action, and it does not know what to do next.

Learning occurs in Soar by converting the results of subgoal-based search into new long-term memory rules which generate comparable results under similar conditions, so that in the future the system has compiled knowledge about what to do in relevantly similar situations. Soar's learning technique is a variant of *explanation-based learning* [Rosenbloom and Laird, 1986; Mitchell *et al.*, 1986].

The Soar architecture does not need to be altered to implement SPATULA. Rather, new domain-independent information is added to Soar's long-term memory to tell it



Goals include: Robot1 in RoomB

Figure 1.4: A simple Robot Domain task.

when and how to abstract. These new abstraction rules then work in conjunction with domain knowledge to produce abstract problem-solving behavior.

The abstraction method is presented using Soar's general problem-solving framework, but the ideas behind the method are not Soar-specific, and are applicable to other problem solvers which perform search in problem spaces and apply explanation-based learning techniques. This will be further discussed in Chapter 9.

1.3.1.2 The Context in which Abstraction Occurs

SPATULA is used during problem solving to reduce the amount of effort required to perform *lookahead*, or projection, searches. These are searches performed towards the resolution of a control impasse, during which the implications of choosing each of the available options are explored and evaluated in turn. Without specifying yet how this occurs, the lookahead searches are performed abstractly; within lookahead, the abstract problem spaces are created from the original spaces as the search proceeds.

Figure 1.4 shows a simple task in a robot domain, and Figure 1.5 shows a typical control decision which might be encountered while carrying out the task. At some point during problem solving, a control impasse may be reached (indicated by a "?")

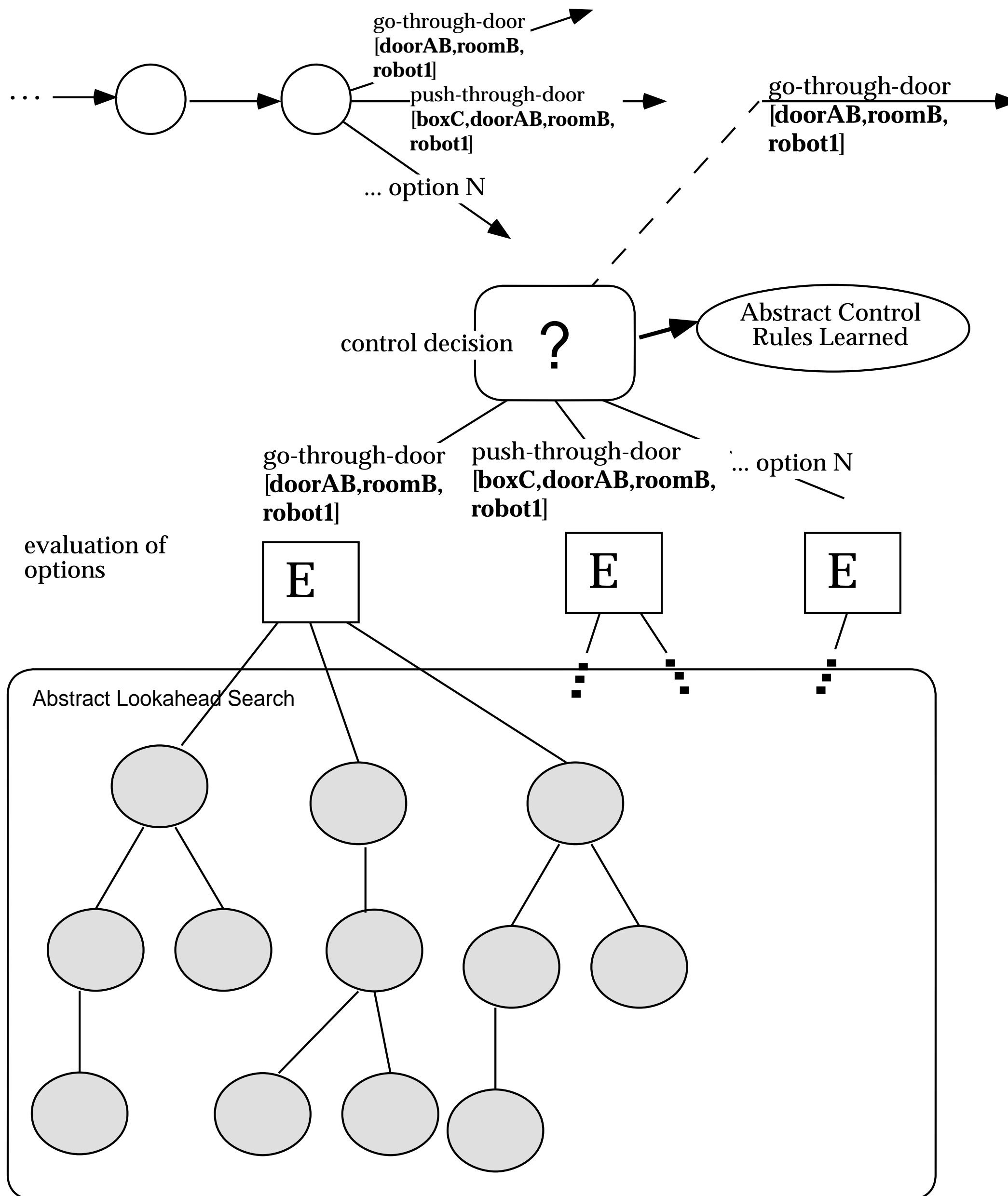


Figure 1.5: The Abstraction Context.

because the problem solver can not decide how to get `Robot1` into `RoomB`. The options in this example include moving `Robot1` into `RoomB` from `RoomA`, and having `Robot1` push a box into `RoomB` (although we have not described the operators used in this figure, their effects should be intuitive from their names). Other options might include moving the robot through `RoomD`, etc. The problem solver will, for each of its options, generate a subgoal (marked "E") to evaluate the option. Typically, it will perform these evaluations using some form of lookahead search. A decision will then be made based on these evaluations, thus resolving the control impasse (in this example, the choice was made to move the robot into `RoomB` from `RoomA`).

Using SPATULA, the lookahead searches on which the evaluations are based are performed abstractly rather than in full detail. If the abstract lookahead searches take less time than non-abstract searches, then the control decision will be made more quickly and tractably (though it might not necessarily be correct). Execution of the task will then continue from that point. If a new control impasse is generated, then abstract search can occur again. Note that the system does not abstract unless it has reached a control impasse; in parts of a task for which it has adequate search control to guide problem solving, abstraction is unnecessary and will not be used.

1.3.1.3 Basic Abstraction Technique: Precondition Abstraction

Using SPATULA, the problem solver abstracts its lookahead searches in the following basic manner. If the problem solver is searching in a context in which it wishes to abstract, then it does so by dynamically abstracting any unmet operator-precondition impasses (impasses generated because a precondition is not achieved) of the operators that it applies. For instance, during abstract search in the example above, the problem solver would abstract an unmet precondition of the "go-through-door" operator which required that the robot be "next to" the door.

The unmet-precondition impasses are abstracted by having the problem solver "assume" that the preconditions have been met, terminating the unmet-precondition impasses, and applying its operators as best as possible. When operator preconditions are abstracted, partial operator application may occur if there is not enough information available for the operator to apply completely. As a result of such partial

operator application, abstract states — with incomplete or incorrect information — may be generated. Therefore, as the lookahead search continues, further problem solving will be most useful if the domain operators can still be appropriately selected given the abstract states, and if they can be partially applied when there is not enough information for full operator application.

If the domain operators are in fact able to be appropriately selected and partially applied from an abstract state, the deliberate precondition abstractions will *propagate*. In this way, a dynamic *reformulation* of the original non-abstract problem space occurs during lookahead search, creating an abstract space on the fly. No special "abstract" problem spaces or operators need to be supplied for this to occur, nor are any architectural changes to the problem-solver necessary.

A set of domain-independent problem-space *design guidelines* has been developed which, when followed, facilitates the problem solver's ability to proceed with problem solving in a useful manner within a dynamically abstracted space. The guidelines make primarily syntactic suggestions about how the domain knowledge should be represented in memory. It is not required that these guidelines be followed for abstraction to take place; however, if the guidelines are followed the abstract search is more likely to be useful.

The initial abstractions produced by SPATULA are PI-Abstractions; more specifically, they are precondition-relaxation abstractions, such as those used by ABStrips [Sacerdoti, 1974]. Further propagation of the abstractions via partial operator application can create reduced states as well. SPATULA's abstractions differ in several ways from those of other systems which use precondition relaxation. For example, with SPATULA the abstract problem spaces are not created ahead of time via predicate removal, but rather are created dynamically. In addition, the abstractions are created for an operator instance (an operator schema with its parameters instantiated) at a particular state, rather than more globally (e.g., for all instantiations of an operator schema, or for an operator instance across all states). This approach allows a greater flexibility and context-sensitivity in the types of abstractions which can occur.

1.3.2 Using the Abstract Searches

The basic abstraction method described above can be used by the problem solver to learn and refine abstract plans at multiple levels of abstraction.

1.3.2.1 Learning from Abstract Search

In Soar, new search control rules are learned during the process of resolving a control impasse[5]. Using SPATULA, abstract look-ahead search will yield abstract, or generalized, search control rules. This is because the "explanation" of the control decision, used to construct the new rules, includes fewer problem details when abstraction is used. The abstract search constructs an explanation using an abstract, dynamically reformulated theory (generated during the abstract search), and the new search control rules are learned by backtracing over the explanation for the reformulated theory rather than the original one.

Because the abstract explanations are simpler, the backtracing process is simpler as well. Thus, the abstract rules are easier to learn, and apply in a wider range of new situations. Although the abstract rules are based on deductive consequences of the reformulated abstract theory, they are *inductively generalized* with respect to the original domain theory. Soar's learning mechanism remains unchanged. Thus, the abstraction process illustrates that deductive learning mechanisms such as EBL can provide inductive concept learning; the abstractions provide the system with a learning *bias* [Rosenbloom *et al.*, 1992; Ellman, 1990; Bennett, 1990b; Knoblock *et al.*, 1991]. SPATULA was in fact the first system to learn inductively via abstract search using EBL.

The learned abstract rules form an *abstract plan*. To see this, consider that the accumulation of new search control rules may be viewed as the incremental construction of a plan. Such a plan is not a monolithic declarative structure to be interpreted by the problem solver. Rather, in relevantly similar future situations, the learned search control rules will reactively apply to provide guidance[6]. (Examples of

[5] More exactly, new rules are learned whenever results of the subgoal search are added to the parent goal.

[6] See [Rosenbloom *et al.*, 1992] for a more detailed general discussion of plans and planning in

other systems which take a similar approach to plan use include [Minton *et al.*, 1989; Drummond *et al.*, 1992].) If lookahead search has been abstract, then the learned plan will be abstract as well. If abstraction has been used for only a portion of a task, then the plan may be partially abstract.

1.3.2.2 Abstract Plan Refinement and Repair

Once a decision is made for an impasse such as that shown in Figure 1.5, ground-space execution continues. The abstract plans learned while resolving the impasse will apply in the non-abstract space and serve to constrain the more detailed search.

Plan refinement and repair occurs in a situated manner. If an abstract plan does not cover all details of a task — whether because a detail has not yet been worked out or because an abstract plan no longer matches the current situation — then further control impasses will be generated in response to the gaps. The problem solver will again do abstract planning to resolve these new impasses. The new abstract searches, filling in the gaps in the plan, will tend to address more detailed aspects of the task than did the previous searches.

The problem-solver interleaves planning and execution in this manner — it does not need to have a full plan for the current task before beginning execution. (The implications of such an interleaved approach are discussed in Chapter 4). As further plan details are successively worked out, a form of *multi-level abstraction* occurs.

1.3.3 Finding a useful level of abstraction: Method Increments

A problem with the abstraction technique as presented thus far is that abstracting all unmet preconditions during look-ahead search is often too extreme; too much information is lost, and therefore the decisions based on the abstract search are not always useful. Thus, the system needs to obtain more information about the abstractions it is making, while making no further demands on the knowledge available to it.

Soar.

To address the problem of making more discriminate abstractions without domain-specific knowledge, five *method increments*, or augmentations to the basic weak method, have been developed. The method increment knowledge — when added to the "basic" abstraction method knowledge used by the problem solver — produces new abstraction methods which use more information about the global problem-solving state than does the basic method, though still making only limited demands on knowledge about the problem domain.

The two primary method increments developed as part of SPATULA are called *assumption counting* and *iterative abstraction*. These method increments combine synergistically with each other, and allow the system to use abstraction effectively in a broader range of situations. They accomplish this by obtaining leverage from the problem-solving context itself; as was the case with the basic abstraction technique, they are driven by the problem solving process.

1.3.3.1 Assumption Counting

With the assumption-counting method increment, existing domain evaluation criteria are combined with a meta-evaluation based on the number of *assumptions*, or abstracted preconditions, required to complete an abstract search. The new combined evaluation is used instead of the domain function to compare the results of the abstract lookahead searches for each option.

Assumption counting is a relatively simple but surprisingly effective method increment. Without requiring semantic knowledge about operator preconditions, it offers a measure (though not an exact one) of the relative difficulty of instantiating the abstract plans discovered during search. It also provides an estimate of the relative amounts of subgoal interaction (both useful and detrimental) in different subgoal orderings. This approach bears a similarity to work done in abduction research to estimate which abductive explanation (that is, which set of hypotheses) is most appropriate. Some examples of such systems will be discussed in Chapter 8.

1.3.3.2 Iterative Abstraction

The *iterative abstraction* method increment is based on the hypothesis that if the problem-solver can't distinguish between the results of evaluating two options at a control impasse, it's operating at too high a level of abstraction. Another way to state this is that with iterative abstraction, the problem solver uses the heuristic that a useful level of abstraction for a particular control decision is that at which it can discriminate among its options.

The iterative-abstraction technique uses the following general algorithm: The system first tries to resolve a control impasse while evaluating the candidate options with much of the problem detail abstracted. If this provides insufficient information to completely discriminate between options, then rather than making a random choice, it re-evaluates those options which looked the best at an increased level of detail (it does not reconsider the options which looked worse). The problem solver continues to iterate, increasing the level of detail at each iteration, until it is able to distinguish between the remaining options (or ascertain that they are equivalent for its purposes).

As discussed in Section 1.3.2.1, the problem solver learns search-control rules as a result of the abstract evaluations. A search control rule is learned each time the problem solver is able to distinguish between a pair of options. This can happen at any iteration cycle — at each iteration, some options may be ruled out. Therefore, during the process of making a single control decision, search control rules may be learned at varying levels of abstraction. The problem solver is acquiring, during problem solving, context-sensitive information about the amount of detail expected to be useful in making various decisions. Thus, it is learning about what abstractions to use.

Using iterative abstraction, more effort than with the basic abstraction method is being spent searching for an initial abstract plan, so as to increase the chances of being able to effectively and efficiently implement it. The process is related to traditional multi-level planning, but the multiple levels occur here in service of creating the abstract plan, rather than instantiating it.

1.3.3.3 Additional Method Increments

In addition to the method increments described above, three other method increments were developed. The *extended plan use* method increment, which is used with iterative abstraction, allows the system to deliberate about the extent to which it will use its abstract plans, and to differentiate between plan fragments learned at different iteration levels so as to use only the most detailed available. Two additional method increments — *goal achievement iteration* and the *abstraction-gradient* method increment — reduce the abstract search complexity while employing heuristics about how to focus on the most relevant aspects of the search.

1.4 Guide to Thesis

The remaining chapters describe and evaluate SPATULA. Chapter 2 describes the foundations of Soar necessary to understanding the implementation of SPATULA. Chapter 3 describes how SPATULA is used to dynamically create abstract problems spaces from the given ground-level spaces during problem solving. Chapter 4 discusses how the problem solver uses the basic abstraction method in an integrated approach to abstract problem-solving. Then, SPATULA's abstraction method increments are described in Chapter 5 and a complexity analysis is given of the abstraction problem-solving framework.

Chapters 6 and 7 describe the results of the empirical evaluation of SPATULA in three experimental domains. Chapter 8 describes other research related to SPATULA. Chapter 9 discusses research issues, outlines future work, and concludes.

Chapter 2

Overview of Soar

The SPATULA abstraction methods are implemented for the Soar general problem-solving architecture [Laird *et al.*, 1987a; Rosenbloom *et al.*, 1991a], and motivated by its capabilities. Thus, in the chapters to follow, SPATULA will be described within the context of a Soar-like problem solver. Towards that end, an overview of those aspects of Soar important to the thesis is given here.

2.1 Introduction

Soar performs goal-oriented problem solving. Essentially, this means that whenever Soar becomes stuck while trying to perform a task, it sets itself a subgoal to get itself unstuck. Soar's goals can be generated recursively: while working to solve one problem-solving impasse, new subgoals may be generated to resolve new difficulties.

Soar is based on the hypothesis that all symbolic goal-oriented problem-solving behavior may be represented in terms of problem spaces, where a problem space is defined by a set of (possible) states and a set of operators [Newell, 1990]. The states represent situations, and the operators represent actions which, when applied to states, produce new (changed) states. A problem space thus can be viewed as representing a context in which a task or subtask will be attempted. A domain may be described by one or many problem spaces, each with its own set of operators associated with it, and each used at appropriate points during problem-solving to

achieve a part of a given task.

When given a new goal or subgoal to achieve, Soar must first decide what problem space it will choose for the situation. This serves to focus and constrain the system's problem solving to that which is relevant for the task at hand. Once a problem space is selected, Soar applies operators to states within that problem space in an effort to move towards its goal. Knowledge may be available in Soar's memory to help it decide what actions to take. If this knowledge turns out to be insufficient at some point, creating a difficulty in deciding what to do next, a subgoal will be generated to resolve the difficulty. This new subgoal might require an entirely different problem-solving approach than that used in the first problem space. A problem space will again be selected for the new subgoal — this problem space may be the same as that used for a previously-generated goal, or it may be different, with its own operators and legal states. (E.g., a problem space to parse an English sentence might be supported by problem spaces used to interpret the semantics of words in the sentence [Lewis, 1992].)

All of Soar's problem-solving activity takes place in this manner, by using subgoal-based search in problem spaces. The types of subgoals which may be generated in response to impasses, and the range of knowledge which may be represented by the problem spaces used within the subgoals, allow a wide variety of problem-solving behaviors.

Soar is able to learn from its problem-solving, and in fact in Soar all learning occurs in the context of problem-solving behavior; one of Soar's tenets is that learning should not occur in a vacuum, and that what is learned is dependent upon the type of task being addressed. Soar's learning mechanism is a form of *explanation-based learning* [Rosenbloom and Laird, 1986; Mitchell *et al.*, 1986].

The description of Soar will be given in a top-down manner. First, we will describe Soar's goal-oriented problem-solving behavior in more detail. Then, we will describe Soar's *decision procedure* — the way in which Soar decides what to do next at any point during problem solving — and show how the goal-oriented behavior emerges from it. Next, we will discuss the memory access representations and mechanisms which form the foundation of these capabilities. Then, we will describe the way in

which Soar learns from its problem-solving. After Soar has been described, a final section will discuss some of the default problem-solving behaviors which may be utilized by Soar, by adding default knowledge to Soar's memory.

2.2 Goal-oriented Problem Solving

The activity of generating and selecting problem spaces, states, and operators completely determines the behavior of Soar. All problem-solving behavior is rooted to an initial, or "top-level" goal, which can be viewed as the goal of interacting with the world. Since problem spaces define sets of applicable states and operators, a problem-solving session must first start by choosing an appropriate problem space to focus the problem-solving within the current top-level goal. For example, if the system was assigned the task of solving a sliding-tile puzzle, it might be appropriate to select a problem space that defined a set of operators which could move tiles. Next, since operators must always apply from states, Soar will always try to generate and select a suitable state once it has selected an appropriate problem space for a situation. Typically, for the initial goal, the initial state will be created using information about the current "state of the world" with respect to the task at hand[1]. Next, an operator may be selected and applied. For example, with the sliding-tile task, an operator might be selected to move a tile. The actions of the operator change the state. From the new state, a new operator may be selected (e.g., to move another tile), and so on[2]. Progress continues in this manner; if a new problem space is selected, it will define new sets of potential states and operators.

Soar tries to select problem spaces, states, and operators based on information stored in its memory. If this is not possible — if the system does not have sufficient information about a situation to make a selection for that situation — then an *impasse* will be generated. There are various types of impasses. For example, the system may reach a situation in which several operators look equally appropriate, and it does not

[1]Much recent Soar research has been devoted to Soar's input and output capabilities — perception and action. However, detailed discussion of these capabilities is beyond the scope of this chapter.

[2]Operators modify the current state when they apply, and so there is no need to explicitly create a new state as the result of an operator application.

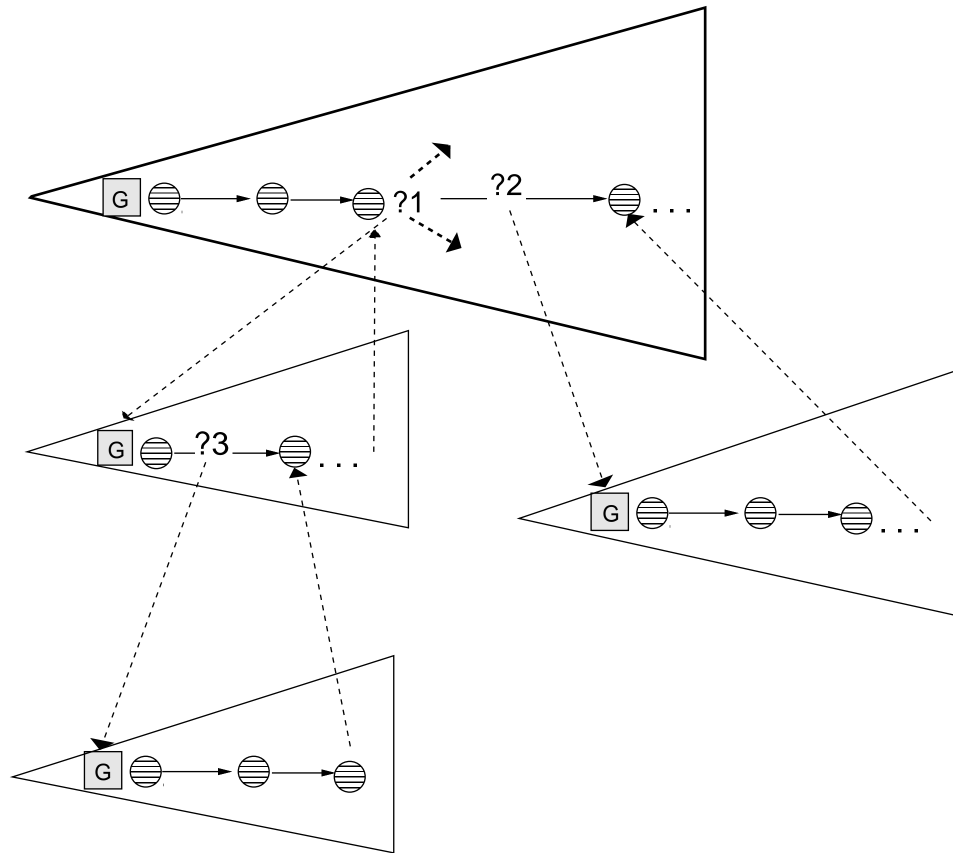


Figure 2.1: Overview of Problem-Solving in Soar

know which one to select next. Such an impasse is called a *tie impasse*. Other types of impasses will be generated if the system can not find any appropriate operators, or does not have stored information about how to apply an operator. In response to an impasse, a subgoal will be automatically generated by the architecture. Within this subgoal, problem-solving will take place to try to resolve the impasse; towards this end, the system again generates and selects problem spaces, states, and operators. For example, if a subgoal was generated because of a tie impasse, it might be appropriate to select a problem space for the subgoal which encoded knowledge about how to deal with ties. (Section 1.3.1.2 in Chapter 1 discusses such a problem space, when describing the technique of lookahead search).

Subgoal generation is recursive; in the process of resolving one impasse, new ones

may be generated. In each subgoal, problem-solving occurs in the same manner, by generating and selecting problem spaces, states, and operators. The current problem space, state, and operator for a goal define that goal's *context*, and are referred to as *context objects*. Each goal has its own current context. In an effort to make progress in the top-level goal, a stack of goals may be generated, with the initial goal at the top of the stack; the entire stack describes the global problem-solving context.

An impasse is resolved when the system has done sufficient problem solving to acquire information about what to do at the impasse. For example, with an operator tie impasse, the impasse will be resolved when the system is able (as the result of search in a subgoal) to choose an operator to select. When the choice is made, all subgoals created because of the impasse will be terminated, and problem solving will continue in the goal in which the impasse was encountered. If an impasse high in the goal stack is resolved, all descendent subgoals lower in the goal stack (more recent) than the impasse will be terminated, since they were all created in service of the impasse.

Figure 2.1 illustrates the way in which problem solving takes place in recursively generated goals. In the figure, squares represent goals. "?"s represent impasses; each impasse causes a new subgoal to be generated. Triangles represent problem spaces. A problem space must be selected for each new goal. The circles represent states, and the solid arrows represent selected operators (selected from the set of operators defined by the problem space), which may apply to the states to generate new states. Impasse #1, a tie impasse, is generated because the system does not have enough stored information to choose among several operators (the short dashed arrows at Impasse #1 represent operators under consideration). Impasses #2 and #3 are generated because the system does not have enough stored information about how to apply a selected operator. In all cases, a new subgoal is generated to resolve the impasse. Impasse #3 shows how a second subgoal can be generated recursively within a first. The long upward-pointing dashed arrows show the return from a subgoal upon the resolution of the impasse which created it, after some amount of problem solving.

2.3 The Decision Cycle

To generate and select problem spaces, states, and operators — context objects — Soar uses a control cycle called a *decision cycle*. The decision cycle can be viewed as a problem-solving step; at each step, Soar tries to make progress by selecting a new context object for at least one of its goals on the goal stack. It does this by accessing its stored knowledge in memory, and using this knowledge to attempt to make a selection.

Each decision cycle has two phases. The first is a memory access phase, called the *elaboration phase*. During this phase, Soar accesses and retrieves from its memory all information relevant to any aspect of any goal context on the goal stack (the details of memory access and representation are described in the following section). For example, the retrieved information can cause a currently selected operator to be applied; provide search control knowledge (suggestions about what to do next, given the current global context); or add annotations to a problem space, state, or operator. No selective reasoning is done during the memory access phase about what to retrieve from memory; all relevant information is always retrieved. The retrieval process may draw from memory more than once; initially retrieved information may then cause additional information to become newly relevant, and it will be retrieved as well. The retrieval process does not halt until there is no more relevant information to access.

When there is no more relevant information to access, the second phase of the decision cycle begins. It is called the *decision procedure*. In this phase, Soar takes into account new information produced during the memory access phase, in conjunction with any previously retrieved information about the global problem-solving context, and uses it to try to decide on a new context object for one (or more) of its goals. The decision procedure is architecturally fixed; Soar utilizes a fixed semantic interpretation of its available search control information about a situation to try to decide on a change. For example, a decision cycle might result in the selection of a new operator within some goal. A context change may be to any goal context(s) in the goal stack. If the system is able to make a change to a "higher-level" goal context and in doing

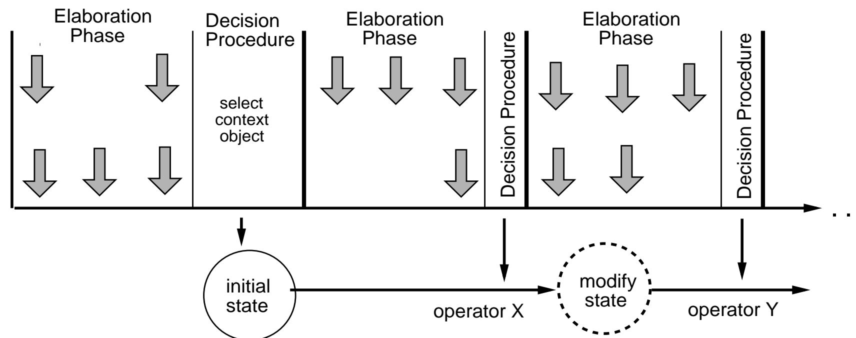


Figure 2.2: Soar's decision cycles.

so resolve an impasse generated earlier, then this decision will terminate any lower-level subgoals generated in service of the impasse. For example, if an operator is selected for a goal which had earlier generated an operator tie impasse, the selection of the operator will resolve the impasse and terminate any subgoal(s) generated in its behalf. Figure 2.1 showed such a situation; Impasse #1 (an operator tie impasse) is resolved when enough problem solving has taken place in a subgoal to determine which operator to select within the highest-level goal context. When an operator is selected, the subgoal is terminated.

In each new decision cycle during problem solving, the elaboration (memory access) phase again retrieves any new information relevant to the changed global context. For example, if an operator is selected during one decision cycle, then in the next decision cycle, information might be retrieved not only about the effects of performing the operator, but — consequently based upon the operator's effects — about which operator to choose next. Thus, the result of the following decision cycle could be that a new operator is then selected to replace the first one.

Figure 2.2 shows this process for three decision cycles; each wide vertical arrow in the elaboration phase represents the retrieval from memory of a piece of information. There may be successive memory accesses (suggested in the figure by successive rows of arrows) in the same elaboration phase, as additional information in memory becomes relevant due to previous retrievals. When no more information is newly relevant, the decision procedure is initiated for that decision cycle. The figure shows

the way in which the decision cycle maps to the goal-oriented behavior of Figure 2.1; here, each decision causes states and operators to be selected.

If Soar does not have enough information to make any context changes at the end of one of its decision cycles, then this means that it has been unable to retrieve enough information to make a decision (i.e., make problem-solving progress) in its most recently generated subgoal or any other. Therefore, a new impasse would be generated within the most recent subgoal— the type of impasse is determined by the results of the decision procedure — and a new subgoal added to the goal stack in service of resolving that impasse.

2.4 Memory

Soar has both a *long-term* and a *short-term* memory. During the elaboration (memory access) phase of the decision cycle, as shown in Figure 2.2, information is retrieved by accessing the long-term memory. The retrieved information is then added to a global short-term, or *working* memory. Soar's long-term memory is represented by production rules, where each rule is a condition-action pair. The conditions of the rules match against the contents of the working memory (rules may contain variables). If a rule's conditions are matched, the rule is executed, or fired, by adding the results specified by the rule actions to working memory. All the rules matched to a given situation are fired conceptually in parallel; there is no conflict resolution during memory access (in contrast to traditional production systems). Rule execution changes the contents of working memory, and thus after matched rules have fired, additional rules may then match and execute, and add new information to working memory. During the elaboration phase of the decision cycle, this match/execute process continues until no new rules match against the current working memory contents[3].

The process of information retrieval from long-term memory is important because Soar is not able to examine the structure of its own long-term-memory rules. This was a deliberate design decision; when Soar's architecture is viewed as a cognitive model, it is hypothesized that the rules provide roughly the same memory access

[3] Provisions are made to terminate infinite match/execute cycles.

functionality as the brain's neural pathways, and thus that they should be equally impenetrable to introspection. This means that Soar can only access its long-term memory via the results of the rules' execution; working memory can thus be viewed as a worksheet, which allows Soar to access currently relevant portions of long-term memory.

All goal context information is held in working memory; problem spaces, states, and operators (as well as annotations to these objects such as information describing what is *in* a state) are all created and added to working memory by rule execution.

Another important type of knowledge which rules may add to working memory is the *preference*[4]. Preferences encode control knowledge about the acceptability and desirability of objects in working memory. Examples of preferences include *indifferent* preferences (two working-memory objects are to be considered equally good with respect to each other within a given context), *better* preferences (one working-memory object is better than another), and *reject* preferences (an object is not appropriate within a given context). It is preference information which is used during the decision procedure described in the previous section, to try make a decision about what problem space, state, or operator to select next in a context[5]. Thus, the preferences encode search control knowledge about a situation.

Although it will not be necessary to consider Soar's knowledge representation in detail for the purposes of this thesis, two aspects of memory representation are of importance: operator representation and plan representation.

2.4.1 Operator Representation and Implementation

Operators are working-memory objects. They may be *implemented*, or their actions specified, by sets of production rules. This is in contrast to some traditional production systems, in which individual production rules have a conceptual one-to-one

[4]Strictly speaking, preferences have their own separate working memory now in the current version of Soar.

[5]Although we will not go into detail here, the preferences also determine which annotations to context objects are currently supported as well (e.g., they determine what information is currently in a state). Working memory is built upon a TMS, so that if a working-memory object is not currently "selected", it becomes inaccessible, as do any other objects whose preferences depend upon the first object's accessibility.

Operator creation rule:

if the problem space for a goal is "sliding tiles"
$\wedge$ there exists a tile `tile` and location `loc`
$\wedge$ `loc` is adjacent to `tile`'s current location
$\Rightarrow$ create an operator for that goal to move `tile` to `loc`

Rules to check operator preconditions:

if the problem space is "sliding tiles"
$\wedge$ there exists a `move-tile` operator with parameters tile `tile` and location `loc`
$\wedge$ `loc` has no other tiles on it
$\Rightarrow$ `precondition-loc-clear` is true for the operator

if the problem space is "sliding tiles"
$\wedge$ there exists a `move-tile` operator such that `precondition-loc-clear` is true for the operator
$\Rightarrow$ the operator `may-apply`

Operator application rule:

if the problem space is "sliding tiles"
$\wedge$ the current operator for a goal is a `move-tile` operator with parameters `tile` and `loc`
$\wedge$ the operator `may-apply`
$\Rightarrow$ change the state for that goal by moving `tile` to `loc`

Figure 2.3: Example pseudo-code representation of a `move-tile` operator for a sliding-tile puzzle (some simplifications have been made for explanatory purposes).

mapping to operators. Consider an operator used in a sliding-tile-puzzle task, to move a tile on a grid. Such an operator, called, e.g., `move-tile`, might be represented (using pseudo-code) as shown in Figure 2.3.

In the figure, one set of long-term memory rules determines when it is appropriate to create and add to working memory an instance of an operator; another set tests that the operator's preconditions (conditions which must be true before the operator may be applied) are met, and adds that information to working memory; and a third set describes how to apply the operator once it has been selected (via the decision procedure) to be the current operator for a goal context, and its preconditions are met. Though just one such operator application rule is shown in the figure, for a more complex operator more than one rule might be used. These three sets of rules together form the `move-tile` operator description[6].

2.4.2 Plan Representation

As discussed in Section 2.4, search control is expressed using preferences. Thus, search control knowledge is encoded in long-term memory by rules which add preferences to working memory. E.g., an operator search control rule might state that if certain conditions are met, add to working memory the preference that one operator is *better* than another.

The search control rules in Soar's long-term memory serve as Soar's *plans*. The rules implicitly represent a sequence of actions (or a partial ordering of actions) to be taken in a situation. If a set of search control rules is applicable in a given class of situations, then this set of rules serves as a generalized plan for the class of situations. Soar's plans are used reactively: plan fragments are applied whenever their conditions match the current situation. Thus, a complete plan for one task may be partially applicable to another. This will be discussed in more detail in the following chapters; see also [Rosenbloom *et al.*, 1992].

[6]The rules in Figure 2.3 do not illustrate the only way in which an operator may be represented. For example, an operator may be implemented in a subgoal: once an operator is selected, a subgoal is generated to apply it, possibly using a different problem space.

2.5 Learning from Impasse Resolution

Soar learns from its problem solving, using a form of *explanation-based learning* [Rosenbloom and Laird, 1986; Mitchell *et al.*, 1986]. Its learning results in new long-term memory rules, called *chunks*. Chunks have exactly the same syntax as Soar's other long-term memory rules, and summarize the problem-solving that has occurred in subgoals. The rules are learned in the following manner.

Each *result* of a subgoal — knowledge added to a higher-level goal context as a result of the subgoal search — causes a new chunk, or rule, to be learned. The actions of the new rule represent the result generated; the conditions of the rule test for those elements, associated with the parent goals, upon which the results depended. The conditions are determined by analyzing the traces of the rules which fired within the subgoal. This dependency analysis determines what information in existence prior to the creation of the subgoal explains the results; this information then provides the conditions of the new rule. The learned rules are deductively generalized; only those aspects of the situation relevant to a particular result are incorporated into rules learned for that result. In future relevantly similar situations, the new rules will match and execute, and allow decisions to be based on direct retrieval of information from long-term memory rather than on problem-solving within a subgoal. In this way, similar impasses may be avoided.

Soar's learning mechanism allows it to exhibit a wide variety of behaviors, determined by the type of subgoals from which the rules are learned. For example, when preferences about higher-level goal contexts are added to working memory as the result of subgoal processing, search control rules will be learned. As discussed above, the search control rules constitute new plan fragments — in similar situations, Soar can directly access its plans in memory to decide what to do next. Thus, through learning, plans are acquired incrementally. Partial plans learned in several different situations may be used together for a new task. If previously learned search control turns out to be incorrect, it is possible to learn new search control which overrides the old [Laird, 1988].

2.6 Problem-solving Methods and Default Knowledge

The Soar architecture provides a framework for a large variety of behaviors; the behaviors are shaped by the knowledge in Soar's long-term-memory, which tells it how to react in various situations; e.g., by making suggestions about what problem space to choose in a certain situation, or what operators to prefer in a problem space. Problem-solving methods are considered "strong" if they contain a large amount of domain knowledge, and are tailored to a specific set of tasks. In contrast, "weak" problem-solving methods make only minimal requirements on the amount of knowledge available about a domain. Thus, they are applicable in a wider range of situations, but will not always be as useful as the more knowledge-intensive strong methods [Laird *et al.*, 1987b].

Some weak problem-solving methods are provided as *default knowledge* to Soar. As defaults, they are methods to be used by Soar in the absence of (or in conjunction with) stronger and more domain-specific problem-solving techniques. Soar's architecture is not changed in any way to provide the default knowledge. Rather, long-term-memory rules are added to Soar which make suggestions about how to react to the different types of impasses which may be generated. These default methods are applicable over a wide range of tasks, and — as defaults — may be overridden by stronger and more domain-specific long-term-memory knowledge if it should become available.

2.6.1 Lookahead Search

Two default weak method approaches used by Soar and pertinent to this thesis are the use of the "selection" problem space and the *lookahead search* technique introduced in Chapter 1[7]. When Soar generates a *tie* or *conflict* subgoal — i.e., when it can not decide which of several candidate operators should be selected during a decision cycle — it takes the default action of creating a selection problem space in which it will

[7]These are two separate methods, but are often used together.

try to evaluate its options and make a choice. Within this problem space, Soar has knowledge about how to set up new recursively-generated subgoals to evaluate each option in turn. Once such an *evaluation* subgoal is generated for a control decision option, the default "lookahead search" knowledge is relevant. This knowledge tells Soar to set up and perform a lookahead search for the option— that is, to perform projections to see what could occur if the option was chosen at that point in the task. Within the lookahead search, Soar tries to reach a state from which it can evaluate the utility of the option (such a state need not be a task goal state). Using the evaluations produced by the lookahead searches, the selection space tries to reach a decision and thus resolve the impasse; in the process, it will learn new rules encoding the results of the decision. Thus, these default weak methods let Soar compare its options at a control decision when it does not yet have compiled long-term knowledge about how to do the evaluations. As a result of problem-solving using the weak methods, Soar will learn search control knowledge which will allow it to choose an option and avoid the control decision impasse in relevantly similar future situations.

2.6.2 Operator Subgoaling

Another default problem-solving method used by Soar is called *operator subgoaling*. Again, the method behavior is produced by adding knowledge to Soar's long-term memory rather than altering the architecture. When an operator is selected but can not apply because some of its preconditions are unmet, a subgoal will always be architecturally generated in response to this impasse. At that point, the operator-subgoaling knowledge suggests that the system perform problem solving within this subgoal, using the same problem space and state as used in the parent goal, to change the state such that the operator in the parent goal can in fact apply. When such a state is reached, the operator applies, the operator application impasse is resolved, and the subgoal is terminated. For example, in the sliding tile domain introduced earlier, suppose an operator was selected to move a tile to a spot occupied by another tile. If the operator could not immediately apply, an operator subgoal might be generated to search for a way to reach a state from which the operator could apply; e.g., a state in which the destination spot was empty.

It is important to note that operator subgoaling alone does *not* specify which operators will be selected within an operator-subgoal; this depends upon the system's search control knowledge for such a situation. (The problem solving in the operator subgoal may recursively generate new subgoals, in order to search different potential operator sequences to see if they acceptably achieve the unmet preconditions.) The operator-subgoaling method simply provides the framework within which to use any available control knowledge about operator subgoals. In practice (both in Soar and in other systems which use similar approaches), the use of operator subgoaling has often been associated with the use of some type of means-end analysis, or MEA — knowledge about which operators are likely to achieve which effects — during the search to reach a state from which the operator can apply. However, this need not necessarily be the case; in Soar, the operator subgoaling method places no restrictions on the type of search control used within the operator subgoal. In fact, MEA knowledge is not part of Soar's repertoire of default methods, and must be explicitly provided as (or derivable from) domain knowledge if it is to be used. (This is because Soar is deliberately designed such that it is not able to examine its own long-term memory rules, and thus means-end knowledge can not be automatically extracted from the operator representations, or provided by the architecture.)

2.6.3 Goal Achievement and Protection

Soar does not architecturally enforce any particular techniques for goal conjunction achievement and protection. The problem-solving knowledge used for a given task will determine whether or not, e.g., goal conjunct achievement is interleaved, and whether achieved conjuncts are protected.

2.7 Conclusion

This chapter has presented an overview of Soar. While by no means a complete description, it has outlined the general capabilities of Soar relevant to the SPATULA abstraction method described in this thesis. In the following chapters, we show how

SPATULA is motivated by these capabilities, and implemented by providing Soar with new long-term-memory rules about how to abstract. As is the case with the default problem solving methods described in Section 2.6, the Soar architecture does not need to be modified in any way to produce the abstraction techniques which will be described.

Chapter 3

Creating an Abstract Problem Space

SPATULA is an automatic and dynamic *weak problem-solving method* for abstraction. SPATULA has been implemented in Soar, and is designed to be a new problem-solving method used by Soar. As with Soar's default weak methods, some of which were described in Section 2.6, SPATULA's abstraction behavior is produced by adding new rules — "abstraction rules" — to Soar's long-term memory. These rules tell Soar how to abstract and how to use the results of its abstract search. Soar's architecture does not need to be changed in any way for it to make use of SPATULA's abstraction techniques.

SPATULA will be described here as used with a Soar-like impasse-driven problem solver, which represents long-term knowledge in the form of production rules and learns from its problem solving. However, SPATULA's approach can generalize beyond use with Soar; the general capabilities required by a problem solver for it to use SPATULA will be discussed in Chapter 9.

As mentioned in Chapter 1, the SPATULA abstraction method is used when Soar *plans* (performs a lookahead search, or a projection). Non-abstract domain problem spaces are dynamically and automatically transformed into abstract spaces during lookahead search. The abstract lookahead searches are easier to perform than would be their non-abstract counterparts. They provide the problem solver with control

information which may be used to restrict more detailed search and thus make the task more tractable.

The description of SPATULA can be broken down into three parts. The first part describes the basic techniques by which the problem solver performs an abstract search using SPATULA, including the way in which abstract problem-spaces are created automatically during problem solving from non-abstract spaces. Effectively, the abstract search creates a dynamic reformulation of a non-abstract space into an abstract one. This chapter describes the techniques which produce and support the reformulation process.

The second part of SPATULA's description describes the way in which the abstract searches may be used by the problem-solver to provide problem-solving heuristics, increase problem-solving efficiency, and learn abstract plans. The third part describes how, given the basic method of abstract plan generation and use, it is possible to build on this basic method in several domain-independent ways (called *method increments*), to produce abstractions of increased utility. These topics are presented in Chapter 4 and 5, respectively.

3.1 Basic Abstraction Method: Creating an Abstract Problem Space

The abstraction technique which forms the foundation of SPATULA is motivated by the idea that it is possible to use performance impasses to influence what to abstract, and the philosophy that all learning activity (including learning about abstractions) should arise from problem-solving.

Many automatic abstraction methods, e.g. [Christensen, 1991; Ellman, 1990; Knoblock, 1991], derive abstractions before problem solving on a task begins, based on a prior analysis of the domain and task descriptions. The abstraction method to be described here draws on a different source of knowledge, that of difficulties encountered during problem solving, and enables dynamically generated abstract problem

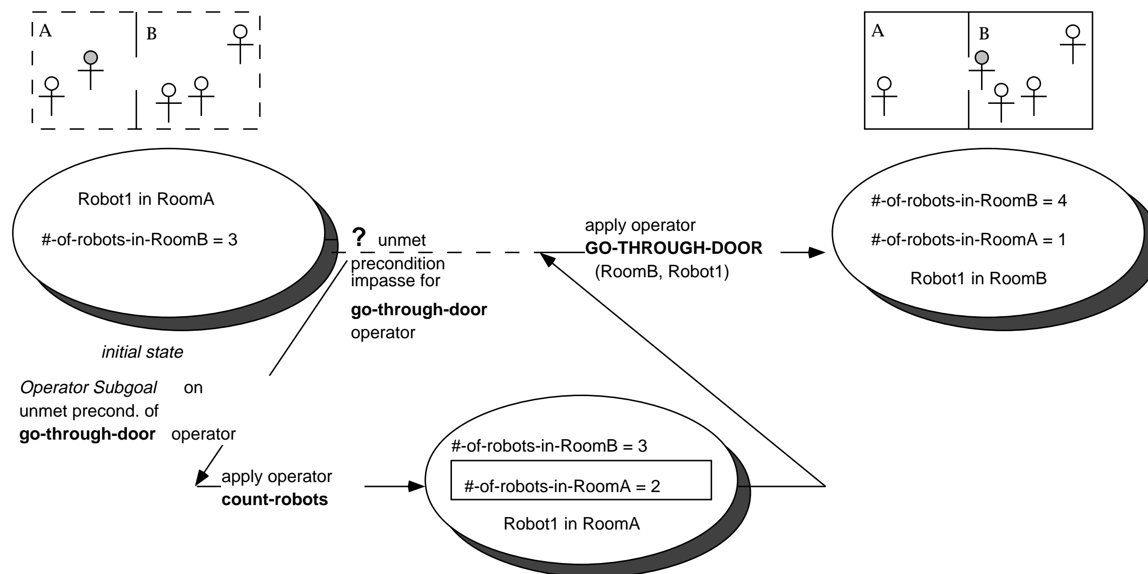


Figure 3.1: A non-abstract application of the `go-through-door` operator.

spaces and abstraction levels. Using SPATULA, the problem solver's abstraction capabilities are obtained not by modifying the problem-solving architecture, but by adding new default knowledge about how to abstract to the problem-solver's memory, and by representing problem spaces so that abstraction can usefully occur.

Because SPATULA is designed to be a *weak* problem-solving method, it is not expected to be useful in every situation, nor is it expected to be as powerful as methods which draw more heavily from knowledge about the domain. However, its sources of knowledge may be available when knowledge which permits problem-space analysis is not — or when there is no opportunity to perform a problem space analysis — and may perhaps be used in conjunction with additional knowledge and abstraction methods to produce even more effective behavior.

3.1.1 Non-abstract Search

To describe the way in which the system uses SPATULA to abstract during planning, it is helpful to first consider the way in which an impasse-driven problem solver such as Soar would behave during a non-abstract lookahead search. Consider a simple ABStrips-like robot domain which contains an operator to cause a robot to go through

a door into another room. The operator application for the `go-through-door` operator includes updating a count of the number of robots in each room of the domain, as well as moving the robot. The preconditions of the operator are that 1) the robot is in a room adjoining the door; and 2) the problem solver knows how many robots are currently in both the robot's current and destination rooms.

Figure 3.1 shows the way in which a *non-abstract* application of the `go-through-door` operator might occur during lookahead search, where the shaded robot in RoomA is being moved. Suppose that the problem-solver has previously counted the number of robots in RoomB, but has not yet counted the robots in RoomA. An unmet precondition impasse is generated, a subgoal is created, and within the subgoal an operator is proposed to count them. When the robots are counted, the `go-through-door` operator's preconditions are met, and the system is able to apply the operator and terminate the subgoal. It does so, moving the operator into RoomB, and updating the robot-count information for each room. The lookahead search may continue from that point.

3.1.2 Abstract Search

SPATULA's dynamic creation of an abstract problem space from the non-abstract space during lookahead search has two facets, described below. In the first, abstractions are deliberately initiated. In the second, these deliberate abstractions are propagated via further problem-solving, creating new incidental abstractions. Both the deliberate initial abstractions and their subsequent propagations contribute to the resultant abstractness of a problem space. In this section, we will describe these processes in general terms. Then, in the remainder of the chapter, the mechanics of abstraction in Soar will be described in more detail.

When the problem solver is performing a lookahead search in a context within which it is appropriate to abstract[1], it *initiates* abstractions at unmet operator precondition-achievement impasses. ("Precondition", as in the ABStrips work [Sacerdoti, 1974], is used to mean a condition which must hold before an operator can

[1]This context will be precisely described in Chapters 4 and 5.

be applied). A precondition-achievement impasse is generated after an operator is selected, when the operator's preconditions are not achieved and the problem solver does not have direct knowledge in its long-term memory about how to achieve them[2]. The unmet precondition impasses are abstracted by having the problem solver assume that the preconditions have been met, and thus terminate the impasses. An alternative way of viewing this process is that the problem solver doesn't care whether or not the unmet preconditions are achieved.

The particular precondition abstractions which occur using SPATULA are dynamically determined by which operators are selected, and which of their preconditions are unmet. Abstraction of an operator precondition does not occur if, once an operator is selected, knowledge is accessible to achieve the unmet precondition. In addition, the problem solver's search-control knowledge determines when an operator is suggested, and thus what preconditions are unmet at that point. The abstractions are performed on operator instances — individual instantiated operators — rather than necessarily abstracting a precondition in the same way across an entire task. Thus, depending upon what is learned during problem solving, the problem-solver's search control knowledge, and the particular task, different sets of preconditions may be abstracted at different times. Figure 3.2 shows the precondition abstraction which would occur for the scenario shown in Figure 3.1. The unmet precondition impasse for the `go-through-door` operator, generated because of the missing count information, is ignored.

After an operator's unmet preconditions are assumed met, the problem solver may then apply the operator. During the operator application process, the system does not reason about whether or not abstraction has occurred, but simply applies the operator as completely as possible, given the (possibly abstract) state information available to it. So, in Figure 3.2, the problem solver will attempt to apply the

[2]The weak method for abstraction described in this chapter provides knowledge about how to abstract at operator precondition impasses only. However, abstraction may occur at other types of problem-solving impasses as well. In particular, abstraction may occur at operator implementation impasses. [Unruh and Rosenbloom, 1989] describes an investigation of operator implementation abstraction.

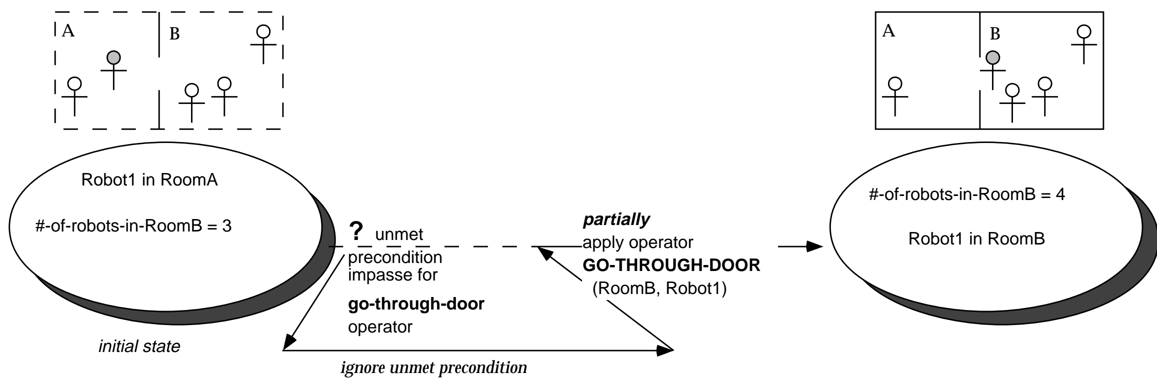


Figure 3.2: The corresponding abstract application of the `go-through-door` operator; partial operator application allows propagation of the abstraction.

`go-through-door` operator once its unmet precondition has been abstracted. However, normal application information is missing (that which deals with the number of robots in RoomA). If the problem-solver is able to partially apply the operator based on the information it does have, then it can still move the robot to its new room, update the number of operators in RoomB, and continue problem-solving from that point. (Section 3.3 will describe in more detail the way in which this partial application occurs.)

The example of Figure 3.2 illustrates two different stages of the process by which an abstract problem space is dynamically created from the original space when unmet preconditions are abstracted. First, the initial deliberate abstraction of the preconditions creates an abstraction; the operator is achieving only an approximation of what its full impact would have been had all its preconditions been met. Abstraction of preconditions may result in a situation in which state information is incomplete (e.g., the missing robot count in the example), or incorrect, or both. It will be incomplete if achieving an ignored precondition would have created a new object or relation, and incorrect if an ignored precondition achievement would have changed or deleted an object or relation.

Second, the problem solver may not have enough information to apply the operator in full if the preconditions are not achieved. E.g., in this example, it does not have enough information to update the robot counts. If this is the case, but if the operator

is able to apply *partially* based on the information it does have, then the problem solver will not be stuck, and will be able to make some progress towards the goal. As the operator applies partially, it may create new abstractions.

A third stage in the creation of an abstract problem space — not illustrated in the example — occurs if abstractions remove information needed for subsequent operator application as well. For example, if the scenario shown above is continued, a subsequent operator application might also reference the number of robots in each room. Again, if these subsequent operators are able to apply partially given the abstract state information, then the effects of previous abstractions can cause them to apply incompletely as well, creating further abstractions. This can occur regardless of whether or not the preconditions of these subsequent operators are themselves abstracted.

Thus, via partial application, abstractions can *propagate* through the abstract search, with initial deliberate precondition abstractions spawning additional abstractions as well. In this way, the assumption process effectively initiates a *dynamic reformulation* of the original non-abstract problem space theory into a new, abstract, theory. With this process, the capability for partial operator application is useful precisely because the abstractions are created dynamically, rather than having been constructed before problem solving begins. As shown in the example, when partial operator application is possible, then precondition abstraction can allow the problem solver to make progress towards its goal with less effort. In Figure 3.2, if the task goals do not deal with the number of robots in RoomA, then the abstracted information is not likely to be necessary for subsequent search — as long as partial application can occur, the problem solver can continue applying operators without this information.

SPATULA's abstractions are Proof-Increasing (PI) abstractions [Giunchiglia and Walsh, 1990a]. The initial abstractions are *precondition relaxation* abstractions (the most well-known example of which is used in ABStrips [Sacerdoti, 1974]), but subsequent propagation of the initial abstractions can cause reduction of the information in the problem-space as well. As discussed in Chapter 1, PI-Abstractions have the characteristic that if there exists a ground-level solution path for a task, then there exists an abstract solution path which subsumes it. As will be seen in Sections 3.4

and 3.5, the dynamic and situated aspects of SPATULA's use disallow a guarantee that for every non-abstract solution in an arbitrarily-designed problem space, such an abstract solution may be constructed. However, it is possible to provide problem-space design guidelines which, if followed, allow this guarantee.

The description above outlined the way in which an abstract problem-space can be dynamically created from a non-abstract space during search. In the following sections, we examine in more detail how this occurs. In Section 3.2, we describe the mechanics of the initial deliberate precondition abstractions. In Sections 3.4 and 3.5, we discuss those aspects of problem-space design which can impact the propagation of the initial abstractions to subsequent additional abstractions, and present guidelines for designing domain problem-spaces such that the abstract search may be complete and useful. Then, in Section 3.6, we discuss how abstraction can change the shape of a search space.

3.2 Specification of Precondition Abstraction Technique

An important aspect of the SPATULA abstraction method is that Soar's abstraction capabilities are obtained not from adding any new modules or pre-processing to the architecture, but by giving Soar new long-term memory rules about how to react in certain situations, and by designing problem spaces so that abstractions can occur usefully once initiated. (These abstraction rules are listed in Appendix A.) Soar's knowledge about how to initiate precondition abstractions is provided by one of these abstraction rules. Figures 3.3–3.5 show the way in which this is accomplished.

As described in Chapter 2, operators are objects in memory. Rules provide knowledge about the operators (e.g., when to propose the operators, how to test that their preconditions are met, and how to apply them). As an example, the first three rules in Figure 3.3 show pseudo-code versions of the domain knowledge which tests that the individual preconditions of the `go-through-door` operator in Figure 3.2 are met. These rules will fire (whether or not the search is abstract) when each precondition is

```
operator op is go-through-door
 ∧ op is instantiated with door, robot
 ∧ state is augmented with adjoins(room-x, door)
 ∧ state is augmented with inroom(robot, room-x)
⇒ go-through-door-precond-1(op,true)

operator op is go-through-door
 ∧ op is instantiated with robot
 ∧ state is augmented with inroom(robot, room-x)
 ∧ state is augmented with #-of-robots-in-room(room-x, n)
⇒ go-through-door-precond-2(op,true)

operator op is go-through-door
 ∧ op is instantiated with new-room
 ∧ state is augmented with #-of-robots-in-room(new-room, n)
⇒ go-through-door-precond-3(op,true)

operator op is go-through-door
 ∧ go-through-door-precond-1(op,true)
 ∧ go-through-door-precond-2(op,true)
 ∧ go-through-door-precond-3(op,true)
⇒ operator-may-apply(op)
```

Figure 3.3: Pseudo-code representations of rules to test the preconditions of the `go-through-door` operator. For readability, most of the rule variables are set in bold-face, and constants are set in italics.

true. The last rule in the figure tests that all preconditions of the `go-through-door` operator are met. If this is true, then the operator `may-apply`.

Figure 3.4 sketches out the format of operator application rules for the `go-through-door` operator. If all rule conditions are met, including the test that the operator `may-apply`, then the operator effects are added to short-term memory. The domain knowledge about how to apply the operator may be encoded in more than one rule, each with a separate set of left-hand-side tests. In Section 3.5.1, the representation of operator application knowledge will be discussed in more detail.

Figure 3.5 then shows the abstraction rule which is added to long-term memory to produce abstractions of unmet preconditions when appropriate. The rule will be

```
operator op is go-through-door
 ∧ operator-may-apply(op)
 ∧ {Condition A}
 ∧ {Condition B}
 ∧ .  .  .
 ∧ {Condition X}
⇒ { Set 1 of operator effects }

operator op is go-through-door
 ∧ operator-may-apply(op)
 ∧ {Condition C}
 ∧ {Condition D}
 ∧ .  .  .
 ∧ {Condition Y}
⇒ { Set 2 of operator effects }
```

Figure 3.4: Templates showing the format of operator application rules.

matched when in an *abstraction context* (other rules, described in Chapters 4 and 5 but not shown here, determine whether or not this is the case), and when an impasse has been generated for the currently selected operator because of one or more unmet preconditions. The rule will fire for each unmet precondition of the current operator, and will add to working memory the information that the precondition has in fact been met. When the unmet operator preconditions are assumed met by using this rule, the last rule in Figure 3.3 will fire, and consequently the unmet precondition impasse will be resolved and operator application rules in Figure 3.4 may apply if their other left-hand-side conditions are matched[3]. As may be seen, the mechanics of initial precondition abstraction are very simple.

As an operator's unmet precondition impasse is resolved using the rule in Figure 3.5, new long-term memory rule(s) are learned, one for each unmet precondition. Then, if in the future the problem-solver is in an abstraction context and the same operator type has the same unmet preconditions, the learned rules will fire and mark the preconditions met, thus allowing the operator to apply abstractly without first

[3]It is possible that no long-term-memory rules will exist to tell the system how to directly apply the operator. In such a case, the system will probably try to apply the operator in an *implementation* subgoal.

```
operator op is a domain operator with name opname
 ∧ operator application of op has reached an impasse
 ∧ the problem-solver is in-abstraction-context
 ∧ <opname>-precond-<X>(op,false)
⇒ <opname>-precond-<X>(op,true)
```

Figure 3.5: The rule for abstracting an unmet precondition when it is appropriate to do so (i.e., when *in-abstraction-context*).

generating such an impasse.

SPATULA's technique for initial precondition abstraction does not require the problem solver to have knowledge of the semantics of the operator preconditions to abstract them; it is not required to know what the achievement of the preconditions will accomplish, but only whether or not they are met. However, if domain information is available which allows the problem solver to reason about whether or not certain of an operator's preconditions should be abstracted, this information could override the default abstraction behavior during abstract search, by specifying that some of the operator's preconditions should *not* be assumed met. In such a situation, the achievement signals for those preconditions could be superseded by the domain-specific abstraction information.

3.3 Propagation of Abstractions: Guidelines For Problem-Space Design

As illustrated in Section 3.1, the initial abstraction of unmet preconditions impasses is only part of the process of dynamically creating an abstraction. Problem solving must be able to continue in a useful manner after preconditions abstractions are initiated. That is the topic of this section.

If the operators in a problem space are designed so that they can apply partially based on whatever information is available to them, then a deliberate precondition abstraction may result in a situation in which both the initially abstracted operator and subsequent operators apply with abstract information as well, thus propagating

the initial abstractions and creating an abstract problem space on the fly. As an abstraction propagates, missing and incorrect information can generate explicit or implicit inconsistencies in the abstract space. This is a well-known consequence of precondition relaxation techniques [Nilsson, 1980]. However, this need not be problematic; recall that the problem solver is only using the abstract search to plan — that is, to make decisions about how to execute a task. If the incorrect or missing information does not hamper its ability to make progress in the abstract search space and to reach useful decisions, then it is not an issue.

However, problems with both search completeness and utility can potentially arise when preconditions are ignored, given an arbitrarily-designed problem space. First, it is possible for the abstractions to interact negatively with operator creation rules or domain search control, and prevent some operators from being suggested when they in fact would be useful or necessary. Second, if the domain operators are not able to apply partially, then this may engender further problem-solving and undermine the utility of the abstraction. (Examples of these problems will be given below).

Clearly, such difficulties should be avoided. However, the philosophy embodied by SPATULA's *weak-method* approach is that no special abstract operators should need to be supplied to the problem solver before a task is attempted, nor should any abstraction pre-processing be required, for the abstraction to occur. Therefore, a set of problem-space *design guidelines* have been developed. If followed, they let the problem solver avoid the potential problems listed above, and enable it to make progress in a dynamically created abstract version of a problem space, without being hindered by the abstractions. The guidelines are primarily syntactic rather than semantic in the sense that they do not constrain the content of the states and operators in a problem space– e.g., they do not suggest what the operators and objects in the problem space should be, or what preconditions the operators should have. However, they do make some semantic suggestions about the most useful representation of some types of domain knowledge; e.g., about the type of search control which will be the most useful.

The design guidelines listed in this section are not strictly part of the SPATULA problem-solving method, in the sense that they are not part of the default rules

which specify when and how to abstract. However, though the guidelines need not be followed for the abstract problem-solving to be successful, the extent to which they are followed will influence how useful the abstract problem solving will be. Thus they are included in the global description of SPATULA as an approach to abstraction, and in that sense are considered part of the method.

The guidelines can be organized into two groups; those which affect abstract problem-solving completeness (i.e., is the abstract search able to reach a goal state in every case for which the corresponding non-abstract search can reach the goal?), and those which affect abstract search efficiency (how efficiently does the abstract search reach its goal?). Guidelines affecting search completeness will be discussed in Section 3.4; those affecting search efficiency will be discussed in Section 3.5. Note that the discussions in this chapter will deal with issues involved in performing the abstract search; Chapters 4 and 5 then discuss similar issues within the framework of using the results of the abstract search to guide more detailed problem-solving.

3.4 Completeness of the Abstract Search

A problem space is *complete* with respect to the SPATULA abstraction method if for every non-abstract solution, the problem solver can construct an abstract solution which subsumes it (as defined in Chapter 1). Though all precondition-relaxation abstractions fall theoretically into the category of PI-Abstractions and thus provide this completeness property, an arbitrary problem space may not in fact allow completeness. Proofs that an abstraction has Proof-Increasing characteristics, e.g. [Giunchiglia and Walsh, 1990a; Knoblock, 1991], are based on the assumptions that 1) those operators whose parameters are instantiable in the abstract space can always be instantiated, or created if need be to reach a solution; and 2) during abstract search, the system is not prevented from exploring all solutions valid in the abstract space.

In actuality, a domain's operator creation knowledge or search control knowledge may prevent the construction of an abstract solution even if it theoretically exists. For example, suppose that in some task, goal conjunct X is achieved by applying operator A, and goal conjunct Y is achieved by satisfying the preconditions of A. Further

suppose that operator B achieves these preconditions. Suppose that goal Y only occurs when X is present, and hence the designer of the problem space has provided domain knowledge which only creates an instance of operator B *after* operator A has been selected and has generated an operator subgoal. That is, the connection between Y and B has been made implicit. This operator creation knowledge might work in the non-abstract space. However, if in the abstract space operator A's preconditions are ignored, then the problem solver will never subgoal to achieve these preconditions. Therefore, operator B would never be proposed at all in the abstract search, and goal conjunct Y would not be achievable[4].

Completeness of abstract search using SPATULA may be guaranteed by ensuring that the problem solver has the knowledge necessary to create and suggest all operators appropriate to a problem-space situation, and if necessary can apply them in an order different from that which is *initially* suggested by the domain's search control. In Soar, abstract search completeness for any problem-space representation is assured by providing three things:

1. the ability to create all instantiable operators within a problem space;

2. adequate default knowledge about handling bad search paths;

3. and non-restrictive use of search control

(all are described in more detail below).

If the system has these capabilities, then it will always be able to find a solution during abstract search if a non-abstract solution exists, regardless of the problem space's capability for partial operator application. First, if the initially preferred search control does not lead to a solution in the abstract space, then the system will always be able to instantiate and try other sequences of operators. Second, if certain abstractions prevent operators from being created or applied at some point, then the system will always be able to construct an alternative operator sequence in which operator preconditions are first achieved such that the abstractions causing the difficulties need not occur.

[4]The example also illustrates that we do not assume that MEA knowledge will necessarily be architecturally provided by the operator definitions.

Thus, if the problem solver has the three capabilities above, it will always be able to produce complete abstract searches using SPATULA. This may be seen by considering that in the worst case, it is always possible to construct non-abstract solutions: solutions in which all of the operators' preconditions are achieved before they are selected, so that no abstractions need occur. Note that in this worst case, the solutions produced within the abstract search space will be only trivially PI-Abstract; that is, they will be equivalent to the non-abstract solutions. Section 3.5 will describe how to design a problem space such that an abstract search is less expensive than its corresponding non-abstract search, as well as complete. First, we discuss in more detail the three requirements for search space completeness using SPATULA.

3.4.1 Completeness of Operator Creation

One approach to problem solving is to only suggest operators when they are relevant to the current situation. The difficulty with this approach — as shown in the example above with operators A and B — is that a definition of relevance which works in a non-abstract space will not necessarily work in an abstract space. Therefore, to ensure search completeness for both search spaces, it is necessary to always suggest (create) all instantiable operators within a problem space. The domain's search control still determines which of these operators are considered first. Thus, the branching factor of the search will not increase as long as the existing search control is indeed able to guide the problem solver to its goal– only the best operators in the partial order produced by the search control knowledge are *initially* considered at a decision point. However, if none of the subset of operators first considered are successful, the problem solver will not be stuck.

3.4.2 Default Behavior of Problem Solver

To ensure abstract search completeness, the problem solver must be able to back up from any "dead ends" it encounters during abstract search, and try a different search path. This is always possible because abstract problem-solving can only occur during lookahead search, or projection — the system can always undo mistakes. In Soar,

such a backtracking capability is in fact already provided by Soar's default rules. The problem solver must also possess the knowledge to detect search loops, since it is possible for loops to occur in the abstract space which would not occur in the non-abstract space.

3.4.3 Non-Restrictive use of Search Control

To ensure abstract search completeness, the problem solver must not commit a priori to using or rejecting certain operators in given situations. This is because search control knowledge which guides the system to a goal in the non-abstract space may not necessarily be appropriate in the abstract space. If this is the case, the system must be able to try operator sequences other than the one first proposed. In Soar, this requirement affects the use of two preference values with special semantics, *require* and *prohibit*. If an operator is *required* at some state in a problem space, it must be applied from that state to reach the goal. If an operator is *prohibited*, it must not be applied if the goal is to be reached. Since the structure of abstract lookahead search could be different from that of non-abstract search, the semantics of *require* and *prohibit* preferences may not still be valid in the abstract space. To guarantee completeness of the abstract search given arbitrary search control knowledge, they should not be used.

3.5 Guidelines for Abstract Search Utility

In addition to the completeness of the abstract search, its utility can also be affected by the design of the domain problem spaces. Clearly, we would like the abstract search to be easier to perform than the corresponding non-abstract search. Two aspects of the problem-space representation impact the relative efficiency of the abstract search:

- If the problem space is designed so that partial problem-solving can proceed when information is abstracted, then the system will be able to construct abstract solutions which subsume their corresponding non-abstract solutions.

> Consider what would happen if the `go-through-door` operator of Section 3.1 had not been able to apply partially during abstract search. If this was the case, then the system would have had to do additional problem solving to count the robots in Room A before it could apply the operator, and the search in the abstract space would have provided no efficiency gains over the non-abstract. Thus, the greater the extent to which the system can make progress when information is missing, the less problem solving it will require to fill in the information before it can reach a solution. The more abstract a solution, the simpler and potentially easier to construct it will be.

We will describe two techniques for problem-space design, called *factorization* and the *information-access* guideline, which allow the problem solver to make progress towards an abstract solution with partial information.

- Abstract search will be most efficient when knowledge about a domain's search control, precondition tests, and goal tests can transfer from non-abstract to abstract situations. Such knowledge can be expected to translate most usefully between non-abstract and abstract spaces if it does not depend upon implicit assumptions about what should have happened during past problem-solving.

Below, we describe these guidelines for problem-space utility. Then, their impact on the abstract search space will be discussed.

3.5.1 Factorization

The less a particular piece of knowledge depends upon the existence of other unrelated problem space information, the more likely it is to remain accessible when other problem space information is abstracted. This is the motivation behind *factorization.* A problem space is *factored* if its long-term-memory representations of problem space knowledge (e.g., knowledge about state creation, operators, or goal tests) are separated into any independent sub-parts which compose them.

```
operator op is go-through-door
 ∧ op is instantiated with robot, door, new-room
 ∧ the operator may apply
 ∧ state is augmented with inroom(robot,old-room)
 ∧ state is augmented with #-of-robots-in-room(old-room,n1)
 ∧ state is augmented with #-of-robots-in-room(new-room,n2)
⇒ add(inroom(robot,new-room)) to state
 ∧ delete(inroom(robot,old-room)) from state
 ∧ delete(#-of-robots-in-room(new-room,n2)) from state
 ∧ delete(#-of-robots-in-room(old-room,n1)) from state
 ∧ add(#-of-robots-in-room(new-room,n2+1)) to state
 ∧ add(#-of-robots-in-room(old-room,n1-1)) to state
```

Figure 3.6: An unfactored operator application.

Soar suggests a particular approach to factorization; as discussed in Chapter 2, Soar was deliberately designed such that it does not reason about the contents of its own rules, or match the rule's conditions in a partial manner. If an abstracted rule condition no longer matches, its rule will not fire. Thus, with Soar, factorization occurs during the problem-space design process by representing independent actions as separate rules, as will be shown below. With a different problem-solver, different approaches to the implementation of factorization might be possible, but the need for factorization would remain the same. An example will first be given of a factored operator application, and then the process of factorization will be detailed.

3.5.1.1 Example

Consider the possible representations of operator application knowledge which could be employed for the `go-through-door` operator of Figures 3.1 and 3.2. The rules which test the operator preconditions were shown in Figure 3.3; if the preconditions are either met or abstracted, then it will be true that the operator `may-apply`. Figure 3.6 shows an unfactored representation (in pseudo-code) of the operator application knowledge. If, as in Figure 3.2, state information about the number of robots in a room is missing, this rule will not match on all conditions, and the operator will not be able to apply.

operator **op** is *go-through-door*
$\wedge$ **op** is instantiated with **robot**
$\wedge$ the operator *may apply*
$\wedge$ state is augmented with inroom(**robot**, **old-room**)
$\Rightarrow$ delete(inroom(**robot**,**old-room**)) from state

operator **op** is *go-through-door*
$\wedge$ **op** is instantiated with **robot**, **new-room**
$\wedge$ the operator *may apply*
$\Rightarrow$ add(inroom(**robot**,**new-room**)) to state

operator **op** is *go-through-door*
$\wedge$ **op** is instantiated with **robot**, **new-room**
$\wedge$ the operator *may apply*
$\wedge$ state is augmented with #-of-robots-in-room(**new-room**,**n2**)
$\Rightarrow$ delete(#-of-robots-in-room(**new-room**, **n2**)) from state
add(#-of-robots-in-room(**new-room**, **n2**+1)) to state

operator **op** is *go-through-door*
$\wedge$ **op** is instantiated with **robot**, **new-room**
$\wedge$ the operator *may apply*
$\wedge$ state is augmented with #-of-robots-in-room(**old-room**,**n1**)
$\wedge$ state is augmented with inroom(**robot**,**old-room**)
$\Rightarrow$ delete(#-of-robots-in-room(**old-room**,**n1**)) from state
add(#-of-robots-in-room(**old-room**,**n1**-1)) to state

Figure 3.7: The operator of Figure 3.6, factored.

Figure 3.7 shows a factored version of the same operator application (and instantiates Figure 3.4). Each independent action is represented separately. The factored operator allows partial application since the rules can fire separately; the robot may be moved and the number of robots in the target room updated even if the number of robots in the originating room remains unknown. In this way abstract problem-solving progress can still be made when information is missing.

3.5.1.2 Implementation

The process of factoring a problem space may be operationally defined as follows. The procedure described below splits unfactored rules, then merges some of the resultant rules back together as appropriate. In the following, "action" refers to the creation of a new object, relation, or piece of search control information. The term "legality conditions" is used for those conditions which are used only to determine whether or not a rule will apply, but don't bind any variables for the actions. For example, consider the following rule:

operator is *go-through-door*
$\wedge$ **op** is instantiated with **robot, new-room**
$\wedge$ state is augmented with color(**robot, color**)
$\wedge$ **color** is blue or green
$\wedge$ the operator *may apply*
$\Rightarrow$ add(inroom(**robot,new-room**)) to state.

In this rule, the clauses
$\wedge$ state is augmented with color(**robot, color**)
$\wedge$ **color** is blue or green
are legality conditions. The binding of the variables in the rule action do not change with a change in robot color.

To create a factored problem space:

1. First, ensure that operator precondition tests are separated from operator application rules and explicitly represented, if the problem space has not already

been designed in this manner. To do this, all legality conditions in operator application rules, except for a test that the operator "may apply", are by default assumed to be preconditions of the operator application. Any such preconditions are moved to separate rules. Any conditions of the original rule necessary to create bindings for the precondition tests are copied to the new precondition rule as well. A rule is created for each such precondition, according to the format described in Section 3.2. For example, in the operator application rule above, the test for robot color would be moved to a separate precondition-testing rule.

Note that any legality conditions which are not moved to separate precondition-testing rules will be treated as *critical* [Sacerdoti, 1974] preconditions by the system (preconditions which are unachievable if not already met, and thus should not be abstracted). If preconditions are not explicitly represented in separate rules, they will not be abstracted. Thus, if the system designer has semantic knowledge about which preconditions should be considered critical, this knowledge can be used to override the default factorization process, and retain those critical preconditions as part of the operator application rules.

Of course, in the absence of such knowledge, some of the explicitly represented preconditions may be critical under certain conditions. The problem-solver will discover for itself whether or not this is the case (and will repair its plan if necessary, as will be described in Chapter 4). In addition, if domain information is available which allows the problem solver to reason about whether or not certain of an operator's preconditions should be abstracted, this information could override the default abstraction behavior during abstract search, by specifying that some of the operator's preconditions should *not* be assumed met. However, if the problem-space representation does not allow many preconditions to be explicitly reasoned about, then the scope of possible abstractions will be small.

Next, for all problem-space rules (including the newly separated precondition-testing and operator application rules), do the following:

2. For each rule with more than one action on its right-hand-side, replace the

original rule with a set of new rules; one for each action of the original rule, and each with the same left-hand-side conditions as the original rule. E.g., one new rule would be created for each action in the rule of Figure 3.6.

3. It may be the case that multiple actions of the original unfactored rule are linked, such that the result of one action must be accessed to achieve another action of the same unfactored rule. In this case, augment each new rule as necessary such that it also includes in its left-hand-side any tests for objects or relations which were created in the original rule, and which the action of that new rule needs to access.

   For example, in the illustrative robot domain described above, suppose that an unfactored rule creates a `go-through-door` operator for a specific robot A, and augments the operator with information about the number of other robots in the same room as A. Two new rules will be constructed to replace the unfactored rule; the first will create the operator, and the second will add the number-of-robots augmentation. This second rule will need to incorporate on its left-hand-side a test for the existence of the operator, so that it may then augment it. Note that if several actions of the original unfactored rule produce a chain of related objects, then it may be necessary to add a corresponding chain of such tests to a new rule.

4. From each new rule, remove any conditions which are required solely for legality (that is, remove any conditions not used to bind the variables of the rule action)[5]. For example, when constructing the rules in Figure 3.7, the tests for `#-of-robots-in-room` were removed from the left-hand-side of the rule which adds "inroom(**robot,new-room**)" to the state, since the count information is not required when specifying the robot's new room.

   Compare the conditions of the resulting rules with the conditions of the original rule from which they were constructed. Let $SetC$ be any conditions of the original rule, not yet used in any new rule. From the group of new rules formed

[5] Recall that precondition tests will have already been removed from operator application rules before this step, and thus are not included in the legality conditions considered here.

from the original rule, find all rules whose conditions do not depend upon the actions of other rules in the group. Call this non-dependent subset of rules, $SetR$. Add the conditions in $SetC$ to each rule in $SetR$. For example, suppose that an original rule was split into two new rules, one which created a new state, and another which added some information to that state. Then, any legality conditions of the original rule, not required to bind variables for the actions of the new rules, would be added only to the rule creating the new state.

5. Any factored rules with identical left-hand-sides may now be merged. For example, two such rules were merged to produce the fourth rule in Figure 3.7. In addition, if there exist two rules 1 and 2 such that Rule 1 has action X and Rule 2's left-hand-side is a subset of Rule 1's left-hand-side plus a test for the object or relation created by X, then the two rules may be merged. The merged rule will have the conditions of Rule 1 on its left-hand-side, and the (linked) actions of both rules as its actions. For example, suppose that under all conditions in which the operator above is created, it is given the name "go-through-door". In this case, two actions (creating the operator and adding the name to the operator) can be merged into one rule which both creates the operator and then adds the name to the operator.

Factorization increases the efficiency of problem solving in a dynamically abstracted space. E.g., as illustrated in Figure 3.2, the problem solver can reach the goal in the abstract space using fewer operations if it can make progress using partial information. The greater the extent to which a problem space is factored, the more efficient abstract problem-solving can become.

The utility of factorization extends beyond operator application. All problem-space knowledge should be factored. Specifically, knowledge about: operator application, operator preconditions, object creation, and goal tests should all be factored. Note that factorization separates creation of objects from the creation of search control about the objects. Precondition factorization was discussed above; the examples below discuss in more detail the way in which factorization of the other types of problem-space knowledge listed above can increase abstract problem-solving utility.

3.5.1.3 Factorization of Operator applications

A factored operator application was shown in Figure 3.7 above. If an operator application is composed of a number of independent sub-actions, and if each sub-action is described separately, then some of the sub-actions may be able to apply even though there is not enough information available to allow the operator to apply in its entirety. In this manner as much of the operation is completed as possible.

In contrast, if the operator was *not* able to apply partially, then one of two things could happen with the abstract search. If the operator was required to apply fully, then more problem solving would have to be done to reach a state in which the operator could in fact apply fully. Alternatively, the operator could be considered vacuously "done". But in that case the problem-solving state would not have changed, and the problem solver would be no closer to its goal in the abstract space. Factorization allows the operators to apply as completely as possible in the abstract space, and thus progress can be made without doing the work necessary to reach a state in which the operator can apply in full. One way to view the operator application factorization process is that only those parts of the operator application which solely test those operator parameters required for operator creation, will always be able to apply and thus will be non-conditional in the abstract space. All other operator effects may become *conditional* during the abstract search.

3.5.1.4 Factorization of Object Creation from Object Augmentations

When a new object is created, the core knowledge needed to create or define it should be separated from knowledge which augments the object with auxiliary or deductive information (the auxiliary knowledge should of course be factored as well). This guideline should be followed for all problem-space objects. Two cases — state and operator creation — are discussed below in more detail.

New State Creation. Creation of initial states in lookahead subgoals is usually based on information in already-existing subgoals. When state information in these pre-existing subgoals is incomplete or incorrect, then knowledge about copying state information from one subgoal to another will be most effective when factored. If all

knowledge about new state initialization were to be stored in the same rule, then the rule would not fire unless it was completely matched. In contrast, factorization of initialization knowledge allows new states to be created as completely as possible during abstract problem solving, which increases the chance that useful problem solving can occur in the new subgoal. The extent to which the new state can be instantiated determines the extent to which it will be initially abstracted in the new subgoal.

Operator Creation. When operators have factorable effects, the knowledge about operator creation needs to be correspondingly factored as well. As an example, consider a `push-box` operator from a robot domain such as the one described above. Suppose that the primary function of the operator is to have a robot push a box from one location to another. Also suppose that the problem space designer additionally wishes to add to the operator application knowledge the following conditional effects: if there is anything "light" on top of the box when the robot pushes it, the object on top will be jolted and knocked off; and if there is something "heavy" on top of the box, it will shift but not fall off.

Operator creation, could then be accomplished in at least two ways. A first (non-factored) approach would be to define three different `push-box` operator-creation rules: one which was used if there was something light on top of the box to be pushed, one which was used if there was something heavy on the box, and one created when there was nothing on the box. Each would augment the new operator's parameters accordingly.

However, suppose that during an abstract search, the information about some object's weight was abstracted away. A problem arises if this object is on top of a box which needs to be pushed. In this situation, the problem solver would not be able to utilize any of the three `push-box` operator creation rules described above, since the first two rules need to know the weight of the object, and the third tests that there is no object on top of the box. Thus, even if the operator's application rules were factored, the problem-solver could make no progress, since it would not be able to create the operator.

An alternative, and more useful, approach to operator creation would be to factor the process by first creating the `push-box` operator with no reference to object weights or whether anything is on top of the box, and then augmenting it with such information if it was available. Since the knowledge about operator application should be correspondingly factored as well, such an operator would then apply partially even if object weight were unknown.

Operator Creation Rules and Critical Preconditions. After a problem space is factored, any legality conditions in operator creation rules will implicitly serve as critical (unabstractable) preconditions. If the legality conditions are not met, the operator will not be proposed (and thus there is no opportunity to abstract these conditions during abstract operator application). For example, in a robot domain, a problem space might be designed such that operators which push boxes are not proposed unless the boxes are "pushable". By designing the problem space so that the test for "pushable" is put in the operator creation rather than operator application rules, it is ensured that the system will not have the opportunity to abstract (or subgoal upon) this condition during operator application.

3.5.1.5 Factorization of Goal Testing.

Factorization of goal-testing knowledge can increase the efficiency of the abstract search as well. If knowledge about how to test for a goal is *not* factored, then the problem solver can only ascertain whether or not a goal has been entirely achieved–it can not test the status of any task sub-goals (conjuncts or disjuncts). The system has a better chance to make progress if it can reason about individual conjuncts or disjuncts of the goal.

3.5.2 Access of State Information from Operators

In addition to factorization, a second guideline for problem-space design has been defined, called the *information-access* guideline. It prevents a partial operator application from being overridden by information which is incorrect due to precondition

abstraction. The motivation behind the factorization technique of Section 3.5.1 is to ensure that the inapplicability of one part of a problem-solving operation will not prevent the application of other parts. However, it may sometimes be the case that information which is incorrect due to an ignored precondition can change or prevent the application of the primary operator actions. For example, if a precondition of an operator is that some amount of a resource exists, and this resource is monitored throughout operator application, then abstraction of the precondition can potentially cause the operator application to be aborted due to lack of resources. This issue may potentially arise with all precondition-relaxation techniques — unless precondition literals are systematically removed from the abstract language prior to problem-solving — and is not specific to SPATULA. The information-access guideline syntactically structures a problem space so that such problems are avoided.

Before describing the guideline, we must note that for the experimental domains which will be described in Chapter 6, design of the problem space according to this guideline had no additional effect on the abstractions produced, for reasons to be discussed below. (For the same reasons, it did not arise, e.g., in the ABStrips work [Sacerdoti, 1974]). Therefore, it has not yet been extensively utilized; this remains a topic for further work.

To introduce the information-access guideline, consider two extensions to the `go-through-door` operator introduced above. First, another precondition of the operator is added (Precondition 4), stating that the door between the current room and the new room must be open. Second, the operator application rules are changed so that the operator moves a given *sequence* of robots through the door rather than just one. Suppose that each time a robot moves through the door, a draft, proportional to the size of the robot, moves it shut a bit. The door movement is calculated as part of the operator application; if the door is shut too far to move the next robot through, then the operator application is terminated at that point (a subsequent go-through-door operator will be proposed to move any remaining robots in the sequence, after the door is reopened).

If the "open-door" precondition for the go-through-door operator is abstracted when unmet, then the door will stay at its prior status of `closed`. This has the

```
operator op is go-through-door
 ∧ op has door
 ∧ state has status(door, open)
⇒ go-through-door-precond-4(op, true)
```

Figure 3.8: A new precondition of the go-through-door operator, which checks that the door is open. (Although not shown here, a test for precondition-4 would also be added to the last rule in Figure 3.3, which checks that all preconditions are met).

```
operator op is go-through-door
 ∧ op has door
 ∧ op may apply
 ∧ state has status(door, closed)
⇒ abort application of op

operator op is go-through-door
 ∧ op has door, current-robot
 ∧ op may apply
 ∧ current-robot has size
 ∧ state has door-status(door, open by width)
⇒ delete(door-status(door, open by width)) from state
 add(door-status(door, new-status(width,size))) to state
```

Figure 3.9: One representation of an extension to the go-through-door operator application, where as a robot moves through, the door shuts by an amount proportional to the robot's size. Using this representation, difficulties are encountered if go-through-door-precond-4 is abstracted, since the operator application is aborted if the door appears closed.

potential to cause problems for the abstract operator application, since if the operator tests for the status of the door after the precondition has been abstracted, it will appear as though *no* robots in the sequence have room to move through. In this case, problem-solving would be at least as inefficient with abstraction as without — using abstraction, the problem solver would still have to determine that it needed to open the door before it could make any further progress. One potential representation of the extended operator, in which this problem does in fact occur, is shown in Figures 3.9 and 3.8.

However, if the operator is represented differently than as shown in Figure 3.9, so that information about the status of the door becomes *no longer accessible* to the `go-through-door` operator when the "open-door" precondition is abstracted, then the abstraction of Precondition 4 will be more useful. If the door-status information is no longer accessible — i.e. if the rules can no longer match against the door-status information — then via factorization those parts of the operator application which do not reason about door status will still apply, such as the rules which update the number of robots in each room. However, those parts of the operator which modify or test the door status will not match if the operator precondition has been abstracted. Specifically, the rules which modify and test the amount by which the door is open will not match. Thus, the operator would not fail when Precondition 4 was abstracted, but instead would apply abstractly by moving *all* the robots in the sequence through the door (since this is the default behavior as long as the door is sufficiently open).

As this scenario suggests, SPATULA's abstraction techniques will be more useful if abstracted precondition information is no longer accessible to the operator during application. This is the motivation behind the information-access guideline. However, because SPATULA's abstractions are *dynamically* determined, we do not want to remove information from a problem space before abstract problem solving starts — this approach in fact motivates all of the problem-space design guidelines in this chapter. Nor do we want to assume that the problem solver has been provided with knowledge about what state information would be affected by the abstraction of a particular precondition under a particular set of conditions. In the remainder of this section, we will first describe the way in which a problem space can be designed to address

this issue during dynamic abstraction, and then discuss some of the implications of our approach.

3.5.2.1 Approach

To allow the system to ignore state information which may be incorrect due to abstraction, we use the following technique: the operator precondition tests are used to dynamically annotate the operator with any state information which causes a precondition of a currently selected operator to be *met* for the current situation. Such state information can only be accessed by the operator via this dynamic annotation. Then, any state information which could potentially contribute to the achievement of a precondition but is not *explicitly noted* as doing so, is therefore not accessible to the operator. For example, a notation, recording the door status, will be added to the `go-through-door` operator if Precondition 4 is achieved. If the precondition is ignored, then no such notation will be added to the operator. The `go-through-door` operator application rules will be designed so that they can only learn about the door-status by looking at this notation. Non-abstract operator application will proceed as before (since the notation will always be present). However, abstraction of the precondition will simply cause any door-status information in the state to be ignored during operator application (since no "door-status" notation, with respect to the current `go-through-door` operator, will then exist). Through factorization, the abstract operator will apply partially without the door-status knowledge.

3.5.2.2 Implementation in Soar

The *information-access* problem-space design guideline is implemented in Soar as described below, and an example is given. To ensure that precondition abstractions do not in fact cause the failure of primary operator effects, the operators must be syntactically designed such that three things happen:

1. The state information tested on the left-hand-side of operator precondition test rules is copied and linked to the operator data structure at the same time that the precondition is noted as achieved. For convenience, call this subset of state

```
operator op is go-through-door
 ∧ op has door
 ∧ state has status(door, open by widest amount)
⇒ go-through-door-precond-4(op,true)
 add(door-status(door, open by widest-amount)) to op

operator op is go-through-door
 ∧ op has current-robot, door
 ∧ state has inroom(current-robot, roomx)
 ∧ state has adjoins(door, roomx)
⇒ go-through-door-precond-1(op,true)
 add(inroom(current-robot, roomx)) to op
 add(adjoins(door, roomx)) to op

operator op is go-through-door
 ∧ op has new-room
 ∧ state has #-of-robots-in-room(new-room, n)
⇒ go-through-door-precond-3(op,true)
 add(#-of-robots-in-room(new-room, n)) to op

operator op is go-through-door
 ∧ op has current-robot
 ∧ state has inroom(current-robot, roomx)
 ∧ state has #-of-robots-in-room(roomx, n)
⇒ go-through-door-precond-2(op,true)
 add(inroom(robot, roomx)) to op
 add(#-of-robots-in-room(roomx, n)) to op
```

Figure 3.10: Precondition testing rules for the new version of the `go-through-door` operator, revised to facilitate more useful operator application if abstraction occurs.

information, added to the operator, *Set X*; where "X" is uniquely instantiated for each different precondition test. If the precondition-testing rules are modified in this manner, then such an operator augmentation will be made for each achieved operator precondition; both those which happened to be already achieved before the operator was selected, and those which were achieved after the operator was selected, in an operator-subgoal.

Figure 3.10 shows the rule which checks Precondition 4 for the `go-through-door` operator, modified according to the information-access guideline. It also shows the precondition tests of Figure 3.3, similarly modified to reflect the guideline.

2. If the operator is partially (or completely) applied via compiled rules, and a left-hand-side of an an operator application rule tests for information always contained in some *Set X*, then the rule should be designed to access this information only through the operator augmentations added as described in Item 1 above. Tests for information not necessarily contained in any *Set X* should be accessed directly from the state rather than the operator augmentations. Any modification during operator application of information in some *Set X*, must modify the operator augmentations as well as the state information.

   Figure 3.11 shows the operator application rules for the new version of the `go-through-door` operator, designed according to the information-access guideline. The first four rules incorporate the new operator extensions. The door-status information is accessed from the operator rather than the state during application.

   The last four rules in the figure are the original operator application rules for the `go-through-door` operator, first shown in Figure 3.7. These rules will not in fact apply any differently when they are modified according to the information-access guideline. This is because the non-achievement of the preconditions in Figure 3.3 can not affect any state information during operator application other than the abstracted information itself. For example, abstraction of the preconditions that the robots be counted (preconditions 2 and 3) can not impact the way that the robot gets moved from one room to the other if the operator is

operator **op** is *go-through-door*
 ∧ **op** has **door**
 ∧ **op** *may apply*
 ∧ **op** has door-status(**door**, *closed*)
⇒ terminate application of **op**

operator **op** is *go-through-door*
 ∧ **op** has **door, current-robot**
 ∧ **op** *may apply*
 ∧ **current-robot** has **size**
 ∧ **op** has door-status(**door**, open by **width**)
⇒ delete(door-status(**door**, open by **width**)) from state and **op**
 add(door-status(**door**, new-status(**width**,**size**))) to state and **op**

operator **op** is *go-through-door*(**current-robot,robot-seq,door,new-room**)
 ∧ **op** has **current-robot, new-room, robot-seq**
 ∧ **op** *may apply*
 ∧ state has inroom(**current-robot, new-room**)
⇒ change(**current-robot**=pop(**robot-seq**)) for **op**

operator **op** is *go-through-door*
 ∧ **op** has **robot-seq**=*empty*
⇒ terminate application of **op**

operator **op** is *go-through-door*
 ∧ **op** has **current-robot, new-room**
 ∧ the operator *may apply*
 ∧ state has inroom(**current-robot, old-room** ≠ **new-room**)
⇒ delete(inroom(**current-robot,old-room**)) from state and **op**

operator is *go-through-door*
 ∧ **op** has **current-robot, new-room**
 ∧ the operator *may apply*
⇒ add(inroom(**current-robot,new-room**)) to state and **op**

operator **op** is *go-through-door*
 ∧ **op** has **current-robot**
 ∧ **op** *may apply*
 ∧ **op** has inroom(**current-robot, roomx**)
 ∧ **op** has #-of-robots-in-room(**roomx, n**)
⇒ delete(#-of-robots-in-room(**new-room, n**)) from state and **op**
 add(#-of-robots-in-room(**new-room, n-1**)) to state and **op**

operator **op** is *go-through-door*
 ∧ **op** has **new-room**
 ∧ **op** *may apply*
 ∧ **op** has #-of-robots-in-room(**new-room, n**)
⇒ delete(#-of-robots-in-room(**new-room, n**)) from state and **op**
 add(#-of-robots-in-room(**new-room, n+1**)) to state and **op**

Figure 3.11: Representation of an extension to the `go-through-door` operator application, in which a sequence of operators is moved through the door. (After each robot in the sequence is moved through the door, the next one is "popped" from the list, and made to be the operator's current robot). Here, the extended operator is represented using the information-access guideline, such that the operator application is facilitated in an abstract space.

factored, regardless of whether or not the "robot count" information is incorrect. The operators in the domains described in Chapter 6 are in fact of this form, and thus for those domains, it did not matter whether or not the operators were designed according to the information-access guideline.

3. If the operator is partially (or completely) applied via sub-operators applying in a subgoal, and information contained in some *Set X* is to be added to the initial state, the initial state creation rules for the subgoal must access this information only through the operator — if such augmentations exist. This requirement is not illustrated in our examples, which do not show operators implemented in subgoals, but is included for completeness.

3.5.2.3 Discussion

With the information-access guideline, we have presented a method for addressing a situation which may potentially arise when using precondition-relaxation abstractions. Information which is incorrect as a result of precondition abstractions may prevent an operator from applying partially when there is in fact sufficient information for it to do so. One example of a class of preconditions for which this problem may arise are those which require that a resource be obtained. Our approach requires the problem-space designer to make a syntactic connection between two parts of the operator: precondition testing and operator application. Using this syntactic link, the process of precondition-testing determines the information used to apply the operator.

Note that the information-access guideline does not in itself always provide a guarantee that abstract application difficulties will be avoided. As a rather unlikely example, the "door-open" precondition test in the scenario above could have been represented implicitly such that to check whether or not the door was open, the problem-solver consulted a "door-status" log sheet posted by the door. With this implicit check, the information accessed in the precondition test (the log sheet) would not be the information which was used during operator application (the physical status of the door). Thus, the implementation of the information-access guideline with

respect to the log sheet precondition test would not prevent problems from occurring when the actual door status was accessed during abstract operator application. To be fully effective, the information-access guideline must be used in conjunction with an explicit representation of the precondition-test requirements. Section 3.5.3 further addresses this issue.

Our approach may be viewed as analogous to the "Strips assumption", which addresses the *frame problem* [McCarthy and Hayes, 1969] — the problem of describing how a situation changes after an operator has applied – by explicitly describing a set of objects and relations which have changed, and assuming that everything else about the situation has stayed the the same. With the information-access guideline, we take a similar approach in determining what state information to ignore when applying an operator. We make note of the information contributing to those preconditions which were met, and assume that all such information not explicitly noted is to be ignored. Incorrect information is not removed from the state, but if it causes any preconditions of subsequent operators to be unmet, these subsequent operators will also be designed such that the incorrect information does not prevent partial operator application.

In contrast, an alternative approach could have attempted to invert the process by explicitly determining all objects or relations in the state whose current values could have *prevented* abstracted preconditions from being achieved, and then deleting those objects from the abstract state so that they did not impact the operator application. However, there would be several problems with such an approach. First, it would require extra work to determine those objects and relations which could have prevented a precondition from being achieved, in addition to performing the precondition tests. The motivation behind abstraction is to reduce the work required during search. Second, there may be many different ways to achieve a precondition. E.g., the "door open" precondition of the go-through-door operator could conceivably be achieved by either removing the door from its hinges or pushing it ajar. Since the problem solver could not be expected to have a priori knowledge of which way the precondition would have been achieved if it had not been ignored, it would (if the problem space designer took this alternative approach) need to remove from the state

all objects and relations which could contribute to an unmet precondition. E.g, if the "door open" precondition was abstracted, the problem solver would then need to remove from the abstract state any information which could have lead to an unmet door-open precondition. This could include the information that the door was attached to its hinges, as well as the information that the door was in the closed position, as well as any other information which could lead to an unmet door-open precondition. The problem of enumeration of these possibilities is similar to that of the *qualification problem* [McCarthy, 1977]. Removal of all such information from the state could in general produce a state which was too abstract to be useful during further problem solving. For these reasons, this alternative approach to implementation of the information-access guideline was rejected.

3.5.3 Avoidance of Implicit Assumptions about Problem State

Thus far we have presented two problem-space design guidelines, factorization and the information-access guideline. These techniques provide support for useful search in dynamically created abstract spaces, by allowing abstract operators to apply with partial information. Thus, they allow the problem solver to progress towards an abstract solution via precondition relaxation. However, to ensure that the structure of the abstract search is as useful as that of the non-abstract, an additional design guideline must be addressed as well; that of avoidance of implicit assumptions about problem state.

As discussed earlier, problem-solving in the abstract and non-abstract spaces will not necessarily follow the same problem-solving steps. Therefore, domain knowledge can be expected to translate most usefully between non-abstract and abstract spaces if it does not depend upon implicit assumptions about what should have happened during past problem-solving. This section addresses this issue, and discusses three aspects of problem design with respect to this guideline: representation of search control, goal tests, and precondition tests.

3.5.3.1 Search Control

A domain's search-control knowledge can affect abstract search efficiency. If the search control developed for a non-abstract domain does not transfer to a corresponding abstract search, then it may not be more efficient than the non-abstract problem-solving. This issue is relevant for any abstraction technique in which ground-level search control is used in an abstract space, and is not specific to SPATULA.

Section 3.4 gave an example which illustrated the way in which search control based on assumptions about what should have happened can backfire. The less the extent to which arbitrary search-control knowledge depends upon assumptions about what should have happened, the more reliably it will transfer to an abstracted search.

If a domain's search control includes complete goal-directed search-control knowledge — where "goal-directed" means that, given an unmet task goal or subgoal, the system has knowledge about what set of operators to suggest — then this search control will be independent of previous problem-solving, and thus will be equally effective during both abstract and non-abstract search. For this reason, goal-directed search control (such as MEA knowledge) can play a useful backup role as *weak* search control, alone or in conjunction with other control knowledge.

3.5.3.2 Goal Tests

Goal tests should be represented such that they test directly for all required goals. The system should not make any assumptions about the achievement of one goal conjunct via the achievement of another, since an implicit goal achievement which takes place in the non-abstract space may not take place during a corresponding abstract search. Again, this issue is not specific to SPATULA.

3.5.3.3 Precondition Tests

Operator precondition tests should be represented such that they test explicitly for any conditions of the state which contribute to the achievement of the operator's preconditions, rather than testing for implicit signals that the precondition has been

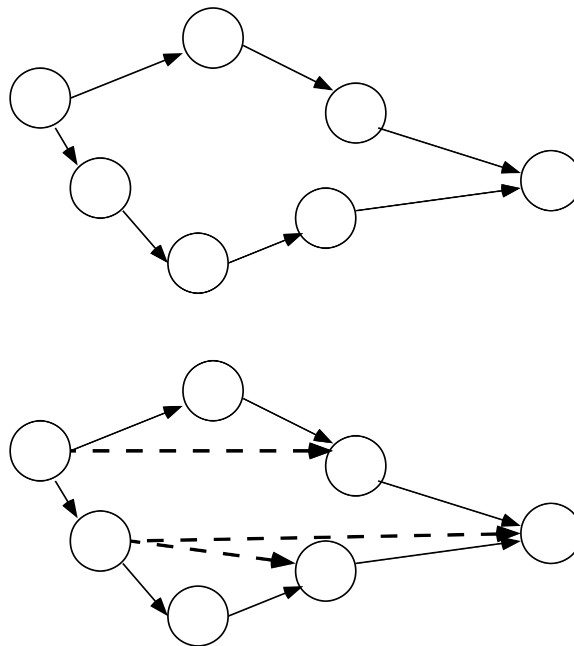


Figure 3.12: Branching factor of state-space search without and with precondition abstraction.

achieved. The "log sheet" precondition test described in Section 3.5.2 was an example of an (undesirable) implicit test of whether a door was open. As discussed in Section 3.5.2, explicit testing for such information allows syntactic detection of that precondition information which was *not* ignored during the precondition achievement process, and which hence may be safely accessed during operator application.

3.6 Impact of the SPATULA Abstraction Techniques on Search Characteristics

Changes in branching factor, search depth, and solution length can occur when a non-abstract problem space is dynamically mapped to an abstract one. Such changes are discussed in this section, with respect to the abstraction properties provided by SPATULA. As has been illustrated in the previous sections, problem-space representation can impact the abstract problem-solving which occurs when SPATULA is applied to a

domain. Therefore, no blanket statements can be made about the way in which the solutions and/or search space will always change within the abstract search. However, if the design guidelines described in Sections 3.4 and 3.5 are adhered to, the following changes will occur.

3.6.1 Branching Factor

During abstract problem solving, more operators can potentially apply directly at each new situation, since it is no longer necessary for all of their preconditions to be met before they apply. This will be true of any precondition-relaxation technique. Figure 3.12 illustrates this change. The figure shows a search space, where circles indicate states and arrows indicate operators. The top figure shows a non-abstract search. The bottom figure shows the same space, where abstraction has enabled additional operators to be applicable (as shown by the dashed lines).

However, the increased applicability of the operators in the abstract space does not in general increase the relative branching factor of abstract search. With SPATULA, unmet operator preconditions are not abstracted until the operator is selected. This means that operator selection criteria, even that which is based upon whether or not the operator's preconditions are met, will remain equally applicable during abstract and non-abstract search. Thus, if domain search control is designed according to the guidelines described earlier — such that suggestions about what to do next are based upon the current state and goals and are independent of previous problem-solving — then the branching factor from a given state will not be affected by abstraction.

3.6.2 Solution Length

In [Kibler, 1985; Pearl, 1983; Valtorta, 1984], it is shown that with the use of precondition relaxation, the length of the shortest abstract solution to a problem will always be less than or equal to the length of the shortest non-abstract solution. This result in fact extends to the class of all PI-Abstractions, and allows PI-Abstract solutions to be viewed as a source of *admissible* search heuristics, to be used by search algorithms such as A*. Chapter 8 further discusses examples of systems which take

this approach.

This theoretical result will always hold for a problem solver using SPATULA, given that the domain problem spaces are designed according to the guidelines described above. If this is the case, then 1) the operators will be able to apply abstractly without their preconditions met, thus allowing shorter solutions by effectively removing precondition subtrees from the solution trace; and 2) overly restrictive search control will not prevent the shorter solutions from being found.

However, for all precondition-relaxation methods, the fact that a shorter abstract solution exists does not necessarily mean it will be found (given arbitrary search control knowledge), unless only admissible search methods are used.

Consider a task for which a system has only incomplete MEA knowledge about its domain. Suppose that Operator A achieves a precondition of an Operator B as its primary effect, and causes secondary effects as well. Suppose that these secondary effects opportunistically achieve various task goals, but that the domain's search control is not "aware" of these opportunistic effects, and only includes the knowledge that A achieves B's precondition. For this task, the shortest abstract solution might involve first applying Operator A, and then applying Operator B. However, the problem solver might not ever construct this solution during its abstract search if it abstracted B's preconditions (and did not perform exhaustive search of the abstract space). A corresponding non-abstract search might in fact be shorter than the abstract, if it serendipitously discovered the opportunistic effects of A while searching to achieve B's preconditions.

Although we have presented this example in terms of comparative solution lengths, this issue is in fact more general; the problem-solving biases present during abstract search can work in conjunction with abstraction to affect the range of non-abstract solutions which the system will produce. This issue is discussed further in Chapter 5, after we have presented the way in which the problem solver uses the abstract search capability provided by SPATULA.

3.6.3 Search Expense

With SPATULA's abstraction method, abstraction removes subtrees from the non-abstract search tree. Those branches of the ground-level search in which the problem solver achieved operator preconditions will no longer be explored. Thus, the search depth will be reduced. If the branching factor of the search stays the same from non-abstract to abstract search, then — since search expense will increase exponentially with search depth — the abstract searches will be exponentially cheaper than the non-abstract.

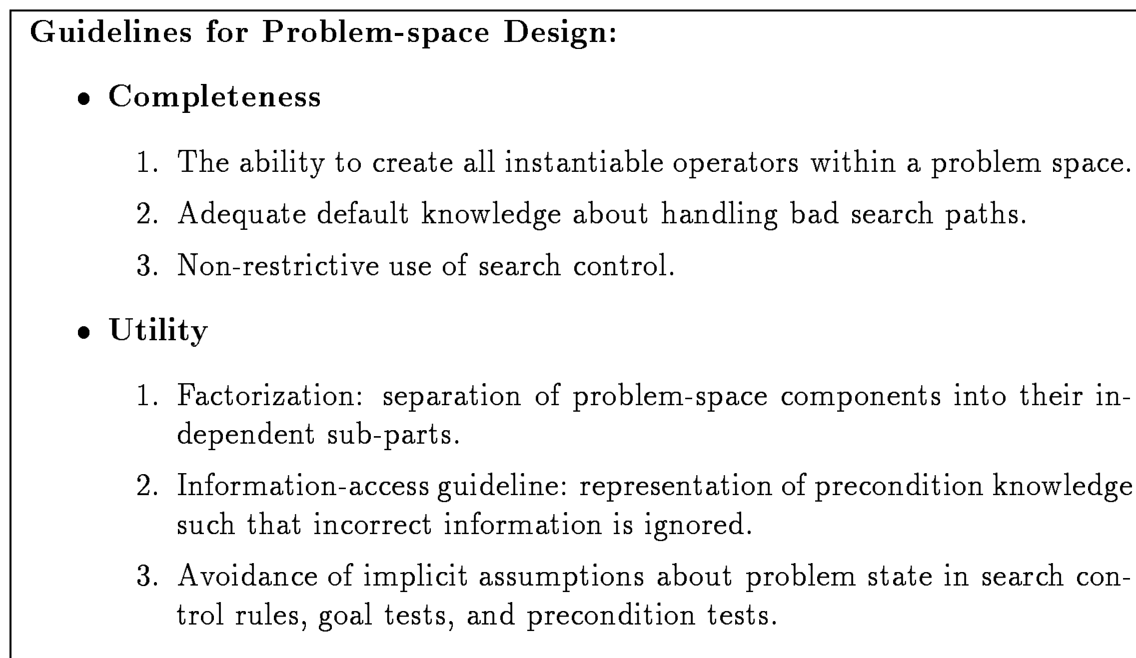
3.7 Discussion and Summary

3.7.1 Use of Problem Space Design Guidelines

The problem-space design guidelines presented in this chapter are summarized in Figure 3.13. The utility of the guidelines is independent of what preconditions in particular are abstracted — an abstraction method which performs dynamic precondition abstraction using different criteria than SPATULA would still need to deal with these same issues. In addition, the ideas implemented by the guidelines are independent of the particular problem-space representations used by a problem solver; e.g., a system need not represent operators using production rules for the factorization guidelines to apply.

The problem-space design guidelines need not be strictly followed for SPATULA to provide useful abstractions. However, the greater the extent to which they are adhered to in an arbitrary domain, the more useful the abstraction will be.

In addition, the guidelines may actually have a wider applicability than their usefulness for performing deliberately abstracted search. They may also increase the robustness of search which is not deliberately abstracted, but in which — because of noise or an incomplete theory — complete task information is not always available. For example, even if a search is not being deliberately abstracted, noise can cause discrepancies, and operators selected during search may not be performable exactly as foreseen. If a domain is factored, then partial application of operators can occur,



Guidelines for Problem-space Design:

- **Completeness**
  1. The ability to create all instantiable operators within a problem space.
  2. Adequate default knowledge about handling bad search paths.
  3. Non-restrictive use of search control.
- **Utility**
  1. Factorization: separation of problem-space components into their independent sub-parts.
  2. Information-access guideline: representation of precondition knowledge such that incorrect information is ignored.
  3. Avoidance of implicit assumptions about problem state in search control rules, goal tests, and precondition tests.

Figure 3.13: Summary of guidelines for design of a problem space so that abstraction may effectively occur using SPATULA.

and still allow useful results to be derived during search.

Similarly, a system should be more robust in general if its search control does not depend upon a particular sequence of operators having been applied in the past. That is, the less the problem-solving sequence is hard-wired, the more able the problem solver will be to respond to unanticipated situations. This is in fact a commonly-discussed issue in the expert systems literature.

3.7.2 Discussion: Operator Representations

Given the basic mechanism of precondition abstraction, a wide variety of potential abstractions are possible. Clearly, with SPATULA, a domain's abstract search behavior will be affected by the way in which domain knowledge has been conceptualized.

Soar allows a fair amount of variability in the way its operators are represented. Thus, the problem-space designer must make representational decisions during the design phase, some of which can impact the system's behavior during abstraction.

In Section 3.3 we discussed one such type of decision: whether to represent operator preconditions explicitly so that they may potentially be abstracted, or to represent them as *critical* (unachievable if not already met) so that they may not abstracted. This may be done by deciding to retain the critical preconditions as conditions in the operator application rules, rather than performing the default factorization into separate precondition tests. Alternatively, preconditions may be implicitly represented as critical by including them in the operator creation rules. If this is done, it means that such conditions must already be met before an operator is proposed, and thus the system does not have the opportunity to subgoal upon or abstract the conditions.

The latter technique (that of including critical conditions in operator creation rules) is an example of an aspect of operator representation in Soar which has a large impact on the range of abstractions produced in a domain: the abstractions which can occur in a domain are dependent upon which subtasks are considered operator preconditions and which are considered part of operator implementations. As discussed in Chapter 2, in Soar (as well as other problem solvers which set up subgoals to address subtasks), operators may be completely or partially implemented in subgoals. For many domains, it is possible to conceive of task representations in which a subtask can be represented either as a complex operator implemented in a subgoal, or as a simple operator plus a set of preconditions which must be accomplished before the operator can apply, or some combination of the two (that is, a complex operator implemented in a subgoal can also have some preconditions that must be achieved before the work is done in the subgoal).

As an example, imagine an operator from the robot domain in the example above, a `push-box` operator which pushes a box to a given location. It can be represented as a simple operator which instructs a robot to move a box, with the precondition that the robot is next to the box. This is the way the operation was represented in the ABStrips work. However, the operation could also be represented as a more complex (non-"primitive") operator which accomplishes in an operator implementation subgoal the tasks of both maneuvering the robot over to the box and pushing the box[6]. Such

[6]SPATULA abstracts operator preconditions only (previous work did involve abstraction of operator implementations; this work is discussed in [Unruh and Rosenbloom, 1989]).

an operator may not have any preconditions which need to be met before subgoal implementation, although the operators in the implementation subgoal may have preconditions of their own (thus allowing the implementation-subgoal problem solving to be abstract).

Thus, the representation of a problem domain, and the designer's conceptualization of operators and their preconditions, have a large impact on the abstractions which may be produced using SPATULA. This impact is independent of whether or not the domain is structured according to the design guidelines presented in this chapter. The effect of operator representations on abstraction refinement will be further discussed in later chapters, after we present the way in which the capability for abstract search may be used by the problem solver.

3.7.3 Summary

This chapter has presented a method for dynamic reformulation of a problem space using precondition abstraction. The abstractions are driven by unmet precondition impasses encountered during problem-solving. As an abstract search proceeds, the initial unmet precondition abstractions can propagate via partial operator applications. These subsequent abstractions as well as the initial deliberate abstractions define the new abstract space. The abstract searches will in general be shallower than their non-abstract counterparts, and thus allow a solution to be found more easily.

Because the abstractions are generated dynamically, the abstract problem spaces (i.e., the abstract states and operators) do not need to be created ahead of time– they are produced during problem-solving from the original problem spaces.

However, again because the abstractions are generated dynamically, the non-abstract problem spaces should be designed so that they may be abstracted in a useful manner. Towards this end, problem-space design guidelines were presented; when followed, they allow a complete and more efficient abstract search, from which subtrees of the corresponding non-abstract search will have been pruned.

The next chapters will build on this basic abstraction method, and show how it may be augmented and used by the problem solver to increase the tractability of

search for a ground-space solution.

Chapter 4

Using Abstract Plans for Problem Solving

The previous chapter has described the way in which an abstract problem space can be dynamically created from a non-abstract one during problem solving, via deliberate precondition abstraction and subsequent propagation of the abstractions; and discussed issues of efficiency and completeness related to creating abstract solutions. However, there has not yet been extensive discussion of the way in which, once abstract problem-solving is able to occur, it can be used to solve the original problem.

As discussed in Chapter 1, one goal of this thesis is to develop SPATULA as a *general weak* problem-solving method for abstraction. SPATULA, as a general weak method, has been designed to:

- increase problem-solving efficiency;
- produce good solutions;
- allow a system to learn more easily from its problem solving — that is, require less effort to build new knowledge about its tasks;
- and increase the transfer of learned knowledge to new situations.

Towards this end, SPATULA is implemented by providing the problem solver not only with knowledge about how to create an abstract space, but also with knowledge

about when and how to use the results of abstract search to guide more detailed problem-solving. With the use of SPATULA, the problem solver has an integrated framework for learning, using, refining, and repairing abstract plans, and accomplishes the goals above via this framework. SPATULA does not require any architectural changes to provide this framework, but rather is implemented by providing the problem solver with additional default knowledge about how to abstract.

This chapter presents this integrated model for abstraction, and describes the way in which the abstract problem-space creation techniques of the previous chapter are used by it. That is, we build on the abstraction method knowledge described in the previous chapter, by providing additional method knowledge which tells the problem solver how to use its abstract search capability. We then discuss the impact of the abstraction method on the problem solver's abilities, and the way it provides the integrated model for abstraction. We will refer to this integrated model as SPATULA's *basic* abstraction technique.

Then, given this framework, Chapter 5 will describe domain-independent modifications to the basic abstraction method, called *method increments*. The method increments — which take the form of additional knowledge provided to the system — build on and improve the basic method, and use the problem-solving context to provide heuristics for selecting useful abstractions.

4.1 The Abstraction Context Revisited: Reducing Decision Time

As briefly discussed in Chapter 3, SPATULA's abstraction techniques are only used within the context of searches to decide what to do at evaluation and selection *control impasses* — points in the problem solving at which the problem solver does not know what to do next because more than one operation seems equally good.

In Soar, a control impasse generates a subgoal in which the problem solver works to resolve the impasse; the impasse will be resolved when there is sufficient search control knowledge generated about the objects involved in the impasse to allow a decision to

be made. Recall from Chapter 2 that the type of search performed in the subgoal towards generation of these preferences is not predetermined by the architecture, and depends upon the kind of knowledge the problem solver has about problem-solving methods in general and about its domain in particular. However, within such a subgoal the problem solver will always be performing internal *lookahead search*, or *planning*, rather than executing actions in the "real" world. The lookahead searches can be recursive; one can be generated within another.

We will refer to the problem-solver's goals of interacting with the world as *top-level*, or *execution-space* goals (in contrast with subgoals generated to plan and reason about the real-world interactions). SPATULA does not use abstraction for the top-level goals of the problem-solver; any actions taken in the context of these goals are real actions, output to the world. Because the actions in the execution space are meant to be real, it does not make sense, conceptually, to try to abstract them. E.g., there is no way to abstract the actual action of walking through a door. What does make sense is to abstract the work that the problem solver needs to do to decide which of these top-level operations to perform.

SPATULA is then used to reduce the amount of effort required to perform lookahead searches and resolve control impasses. More specifically, SPATULA includes knowledge which matches against the current global problem-solving state and tells the problem-solver when it is in a context in which search may be abstract. The lookahead searches are then dynamically abstracted from the original problem spaces, using the techniques described in Chapter 3.

The abstract lookahead searches will produce abstract evaluations of the candidate options causing the control impasse. The evaluations are abstract because they depend upon fewer task details. These abstract evaluations then are used to resolve the control impasse. If an abstract search takes less time than the non-abstract one would have — which will be the case if the design guidelines of the previous chapter are followed — then the impasse will be resolved more quickly. Abstraction allows the problem solver to make decisions about its domain operations more easily.

Once the impasse is resolved, problem-solving continues from that point. If any new impasses are encountered, the process of abstract lookahead search is repeated to

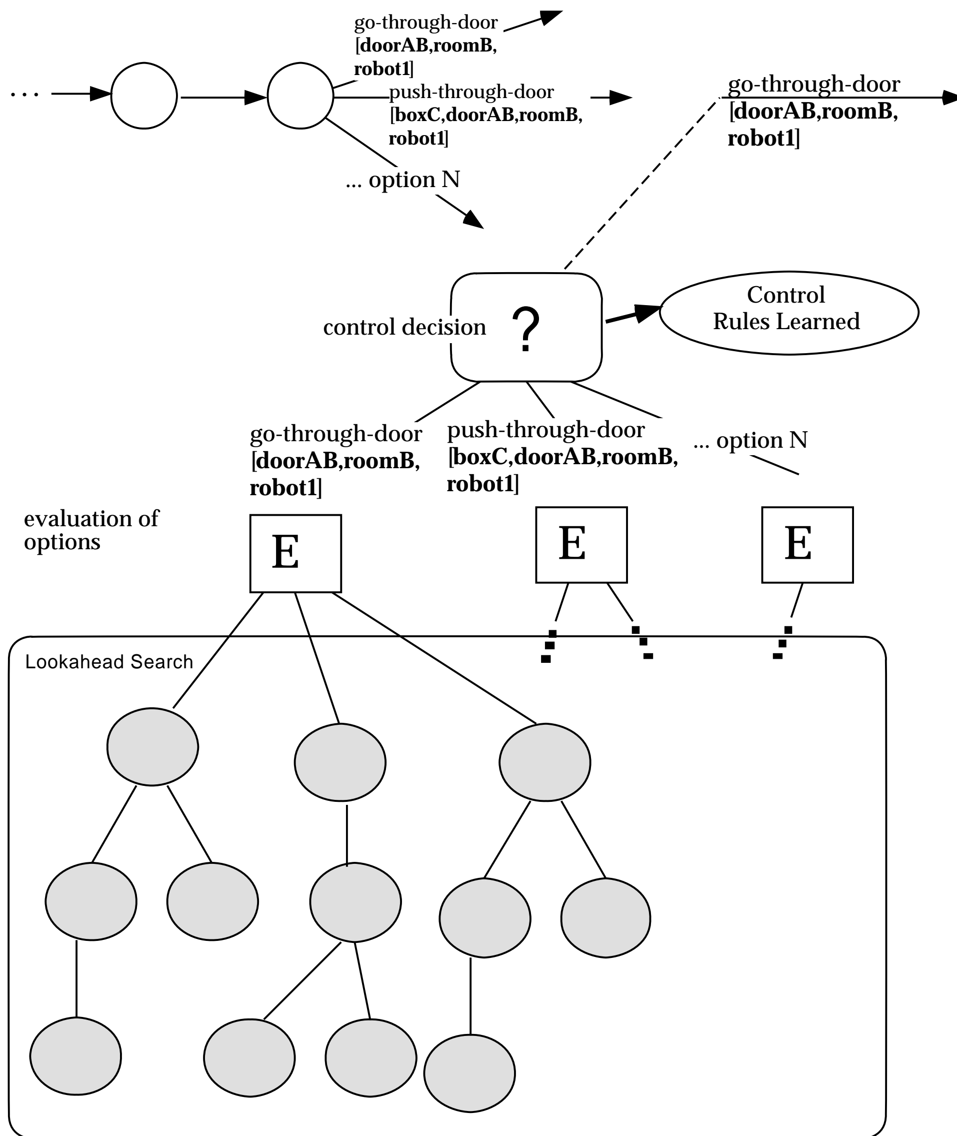
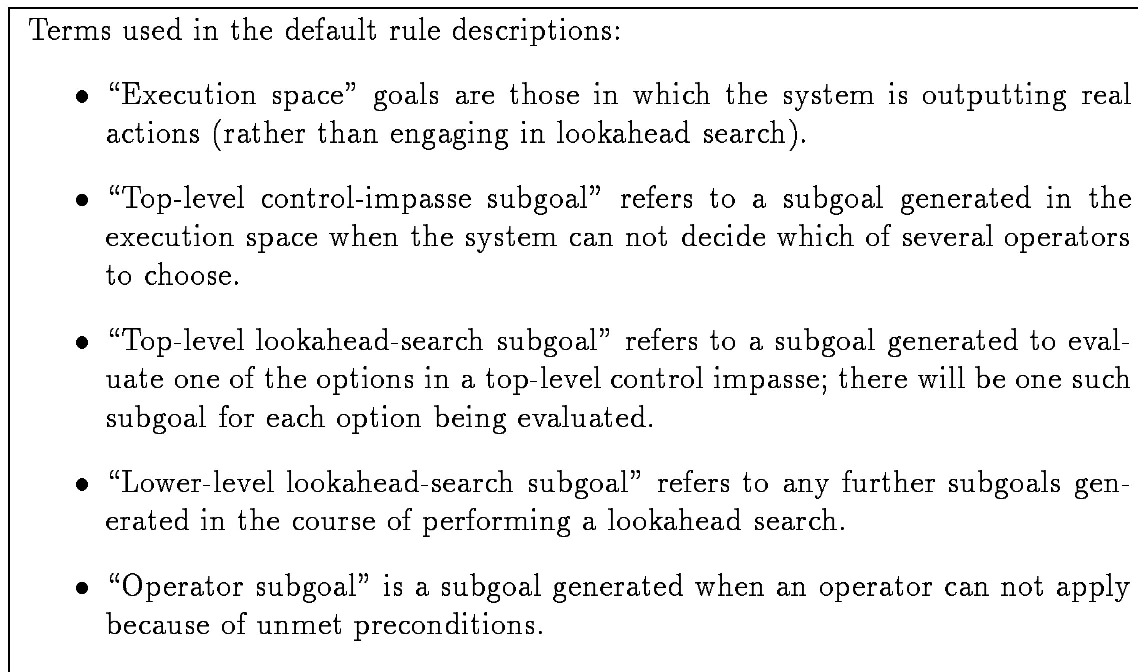


Figure 4.1: The context in which abstraction takes place during problem-solving.



Terms used in the default rule descriptions:

- "Execution space" goals are those in which the system is outputting real actions (rather than engaging in lookahead search).
- "Top-level control-impasse subgoal" refers to a subgoal generated in the execution space when the system can not decide which of several operators to choose.
- "Top-level lookahead-search subgoal" refers to a subgoal generated to evaluate one of the options in a top-level control impasse; there will be one such subgoal for each option being evaluated.
- "Lower-level lookahead-search subgoal" refers to any further subgoals generated in the course of performing a lookahead search.
- "Operator subgoal" is a subgoal generated when an operator can not apply because of unmet preconditions.

Figure 4.2: Terms used in the default rule descriptions of Figures 4.3 and 4.4.

resolve these impasses. Thus, the problem solver *interleaves* (abstract) planning and execution. Figure 4.1 reproduces Figure 1.5, and illustrates this process. Note that in this model, the problem-solver is using the results of its abstract search directly in the execution space, rather than using the abstract information to guide further, less abstract, lookahead search. This usage is independent of the particular methods used to generate the abstractions, and is further discussed in Section 4.4.2.

Figure 4.2 defines the terms used in the abstraction-rule descriptions of Figures 4.3 and 4.4. Then, Figures 4.3 and 4.4 summarize the contents of those default rules which provide to the system the knowledge about when and how to initiate abstract search[1]. That is, they allow the system to remember when it is within a lookahead search context, and to deliberately abstract unmet preconditions as described in Chapter 3. In these rules, the `abstract-at-level` for all lookahead search subgoals is set to

[1]The rules in the figures give an overview only; Appendix A provides more detail. Some of the rules in the figures may correspond to more than one production rule in the appendix. The order of the rules does not imply a chronological ordering; they fire when their conditions match the current problem-solving context.

1. When a top-level control-impasse subgoal is initiated, set the **abstract-at-level** variable for that impasse to 1.
2. When the top-level lookahead-search subgoal for each option in a control impasse is initiated, set the **level-count** variable for that search to 1.
3. When the top-level lookahead-search subgoal for each option in a control impasse is initiated, set the **current-abstract-at-level** variable for that lookahead search subgoal to the value of **abstract-at-level**.
4. For each new lower-level subgoal within a lookahead search, create a copy of the parent subgoal's **current-abstract-at-level** variable, local to the new subgoal.
5. Within lookahead search, for each subgoal, compare the **level-count** value with the **current-abstract-at-level** value for that subgoal. If **level-count** is $\geq$ **current-abstract-at-level** set an **in-abstraction-context** flag in that subgoal.
6. If a subgoal is in an abstraction context (that is, flag **in-abstraction-context** is set), then copy and set the flag in any child subgoals which are generated.

Figure 4.3: Default knowledge for detection of abstraction context.

7. If a subgoal is in an abstraction context (that is, flag **in-abstraction-context** is set), and there is an operator application impasse and a precondition of that operator is not met, then add the flag which states that the precondition is in fact met.

Figure 4.4: Default knowledge for initial abstraction of preconditions. This figure is a generalization of the rule shown in Chapter 3, Figure 3.5.

1, which matches the top level of lookahead search. This simply means that given these rules alone, the problem solver will consider all search subgoals to be within the abstraction context, and will perform all lookahead searches abstractly, by abstracting all unmet preconditions of selected operators.

The rules in these two figures, in conjunction with the techniques of Chapter 3, provide the problem-solver with its basic abstraction framework: one which allows multi-level abstraction, as well as situated learning, use, and refinement of abstract plans. In the remainder of this chapter we will describe the way in which this occurs. In Chapter 5, we will then add to this set of default rules to build upon this framework and modify the context in which abstraction occurs.

4.2 Abstract Search Occurs Only When Necessary

An important aspect of SPATULA's approach is that since abstraction is only used when a control impasse is encountered, it is not employed unnecessarily. The problem solver will use abstraction only when existing search control is not sufficient; there is no need for it to abstract when it already knows what it will do next. In addition, since the abstractions are produced dynamically during lookahead search, the problem solver avoids generating abstractions which it will never use.

4.3 Learning From Search in the Abstract Space

If a system has the ability to learn from problem-solving, as does Soar, and it is able to conduct search in an abstract space, then it can learn from the abstract search. The abstract rules are more general, and may be of greater utility; non-abstract rules learned via EBL are often overly specific [Etzioni and Minton, 1992].

In Soar, whenever results are obtained from subgoal processing, new long-term-memory rules are learned which contain the results in their right-hand-sides and encode in their left-hand sides those conditions that existed in the goal context *before*

the impasse occurred, which were necessary for the result to be generated (multiple results produce multiple new rules). The learning is a variant of EBL [Rosenbloom and Laird, 1986]; the rule's conditions are determined by backtracing through the dependency structure of data produced by the problem-solving in the subgoal, with the result as the *goal concept*.

In the case of a control-impasse subgoal, new rules are learned as a result of adding to working memory any new search-control information generated in the subgoal to resolve the impasse. The left-hand-sides of the new rules contain those aspects of the situation in which the impasse was generated which were relevant to producing that search control knowledge; the right-hand side is the search control itself.

If the problem-solving in the subgoal has been abstract, the problem solver will have looked at less information to produce its results, since parts of the problem have been ignored. That is, information that would normally have been backtraced through to explain a result is abstracted, and some subtrees of the corresponding unabstracted explanation tree no longer need to be expanded for the goal to be explained. (Another way of looking at this is that during abstraction, some nodes in the explanation of the solution are effectively replaced with the value *true* [Keller, 1990]). Because of this change in the proof tree, the conditions of the new rules will be abstract as well; that is, they will test for fewer aspects of the current context before firing than would their non-abstract counterparts. They are also more general.

4.3.1 Examples of Abstract Learning

As an example of the abstract learning process, consider again a task in a simple ABStrips-like robot domain. Figure 4.5 shows the necessary operators, initial state, and goal state for the task. Figure 4.6 shows the explanation structure produced by the problem solver for this task; arrows represent explanation dependencies. Without abstraction, suppose that the problem solver first proposes the `push-box-to-box` operator to move B1 to B2, generates an operator subgoal (because the robot is not next to B1), does some further problem-solving (using the `goto-box` operator) to get near B1, and then is able to apply the operator. The leaves (leftmost nodes) of the explanation show the initial state and context information used to produce the

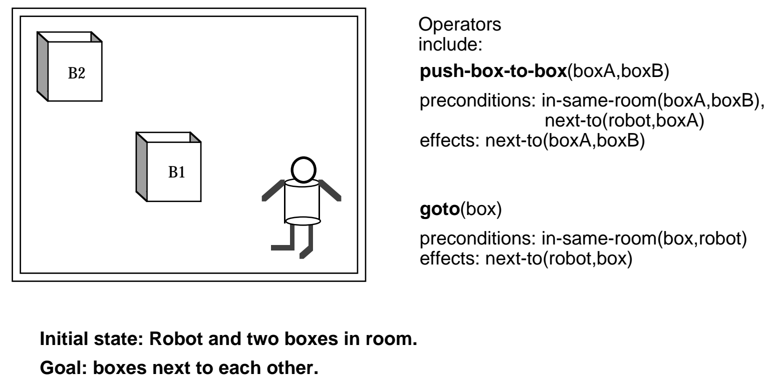


Figure 4.5: Example robot domain task and operators.

solution[2].

When the problem is solved abstractly using SPATULA, the encircled nodes (and the dashed dependency arrows) are no longer required for the explanation. The unmet precondition of the `push-box-to-box` operator is ignored, and since problem-solving is no longer done to achieve that precondition, some of the leaves of the explanation (e.g. `in-same-room(b1,robot)`) are no longer necessary, and will not be incorporated into the learned rule. Essentially, the rule no longer tests that the robot and box B1 are in the same room before suggesting that the `push-box-to-box` operator will be useful when the `next-to` goal is present.

For a more complex comparison of the difference between learning with and without SPATULA, Figure 4.7 shows another task in a more extensive Robot Domain. The domain shown here is used for some of the results presented in Chapter 6, and a complete listing of the operators is given in Appendix D.

Figures 4.8 and 4.9 show the rule learned for this task during non-abstract and abstract search, respectively, as a result of resolving the first execution-level operator tie. At this decision, the system compares the operator to close the door between Rooms 5 and 6 with the operator to push Box C into Room 7. In both cases (from both the abstract and non-abstract search), the system learns that it is easier to close the door first and then move the box, rather than to move the box and then come

[2]For simplicity, this example just shows the problem-solver reaching "goal success". In reality, for both the non-abstract and abstract searches, the search will produce an evaluation, which will consequently be used to produce search control rules.

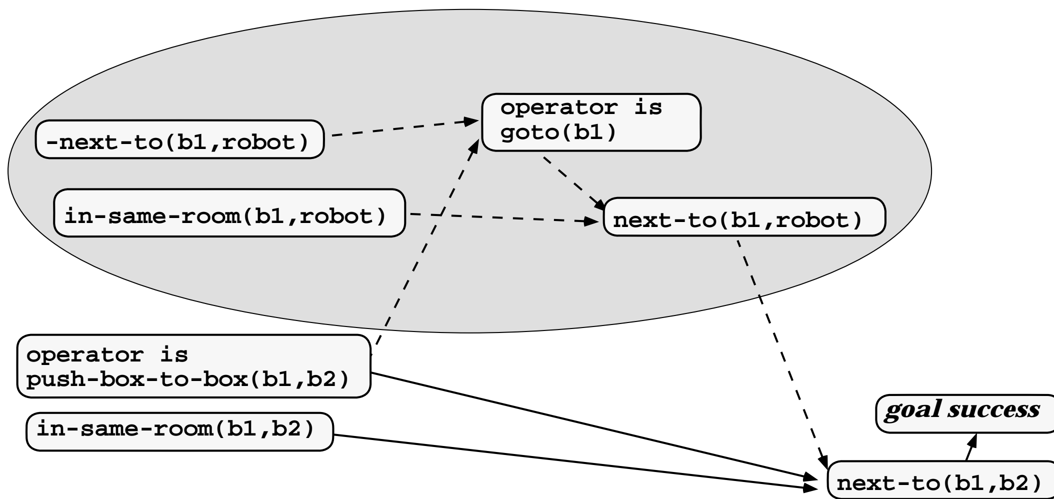


Figure 4.6: An explanation of goal success for the task of Figure 4.6. Arrows represent explanation dependencies. Circled area and dotted arrows represent information ignored during abstract problem-solving. With non-abstract problem-solving, all of the leaves (leftmost nodes) are conditions of the explanation. With abstract problem-solving, only the unshaded leaves are conditions of the explanation.

back to the door. In the case of the abstract search, it makes this assessment by noticing how close the robot is to the door, and does not work out the full details of moving the box into Room7.

In the rules, which are shown in Soar-like syntax, the variable names are enclosed in angle brackets (e.g., "<g6>"). In the examples, most of the Soar-generated variable names have been replaced by names indicating the values with which they were originally matched when the rule was formed. This facilitates comparison between the two examples. (The variables can of course still match with other appropriate values as well.)[3]

The non-abstract rule contains more "if", or left-hand-side, conditions than the abstract rule. Those conditions not contained in the abstract rule are marked with an asterisk on the non-abstract rule. Both rules check that doors Room-5/Room-6, Room-5/Room-7, and Room-4/Room-7 are open; that the robot is in Room 5;

[3]In addition, for readability, various tests which require that variables be different from each other (e.g., that two "room" variables are matched to different rooms) have been removed from the rules as well. For the purpose of the example, the reader may assume that if variables have different names, they must match different short-term memory objects (normally there is no such restriction).

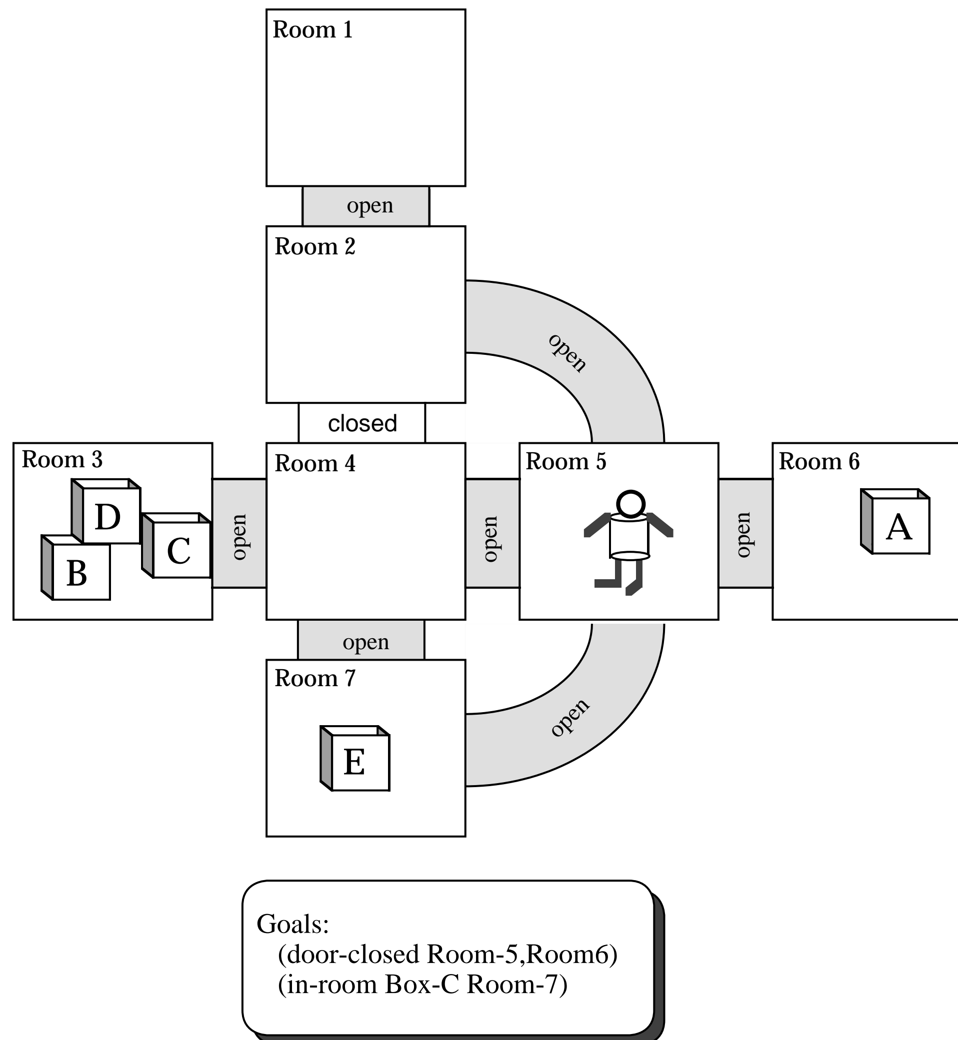


Figure 4.7: Example initial state and goal for Robot Domain task.

```
;; non-abstract rule
(sp st-2-15-p8615 elaborate
   ;; if:
   (goal <g6> ^problem-space <p1> ^state <s1> ^desired-state <d6>
              ^op-set <o1>
              ^desired-op <d5> ^desired <d6> -^operator <o2>)
   (problem-space <p1> ^name robot-domain )
   (op-set <o1> ^operator <o3> <o2>)
   (desired <d6> ^better lower ^goal-conjuncts <g3>)
   (goal-conjuncts <g3> ^door-closed <g2> ^in-room <g5>)
   (door-closed <g2>  ^door <room5/room6>)
   (in-room <g5>  ^box <b1> ^room <room7>)
   (state <s1>  ^door <room5/room6>
       *  <room5/room7> <room4/room7> <room3/room4> <room4/room5>
          ^connects <c1> <c9>
       *  ^connects  <c3> <c4> <c2>
       *  ^next-to <n1>
          ^box <b1> ^robot <robot>
          ^in-room <i1>
       *  ^in-room <i2>
          ^door-status <d1> <d4> <d7>
       *  ^door-status <d2> <d3>
  )
   (door <room5/room6> ^type door)
   (connects <c1> ^door <room5/room6> ^room <room5>)
   (box <b1> ^pushable t )
   (room <room7> ^type room)
   (connects <c9> ^room <room7>  <room4> ^door <room4/room7>)
   (door <room4/room7> ^type door)
   (in-room <i2> ^obj <robot> ^room <room5>)
   (door-status <d1> ^status open ^door <room5/room7>)
   (door-status <d4> ^status open ^door <room5/room6>)
   (door-status <d7> ^status open ^door <room4/room7>)
*  (room <room5> ^type room)
*  (connects <c3> ^room <room5> <room7> ^door  <room5/room7> )
*  (door <room5/room7> ^type door)
*  (room <room4> ^type room)
*  (connects <c4> ^room <room4> <room3> ^door <room3/room4>)
*  (door <room3/room4> ^type door)
*  (next-to <n1> ^obj2 <room3/room4> ^obj1 <b1>)
*  (connects <c2> ^room <room5> <room4> ^door <room4/room5>)
*  (door <room4/room5> ^type door)
*  (in-room <i1> ^obj <b1> ^room <room3>)
*  (door-status <d2> ^status open ^door <room3/room4>)
*  (door-status <d3> ^status open ^door <room4/room5>)
   (operator <o3> ^name close-door
                  ^type robot-domain-op ^door <room5/room6>)
   (operator <o2> ^name push-through-door
                  ^type robot-domain-op  ^into-room <room7>
                  ^from-room <room5> ^box <b1> ^door <room5/room7>)
 ;; then:
 -->
   ;; operator <o2> is worse than operator <o3>
   (goal <g6> ^operator <o2> < <o3>))
```

Figure 4.8: A non-abstract rule produced for the example task of Figure 4.7.

```
;;abstract rule
(sp st-2-15-p85 elaborate
   ;; if:
   (goal <g6> ^problem-space <p2>
              ^state <s1> ^desired-state <d4> ^op-set <o1> ^desired-op <d3>
              ^desired <d4> -^operator <o2>)
   (problem-space <p2> ^name robot-domain )
   (op-set <o1> ^operator <o3> <o2>)
   (desired <d4> ^better lower ^goal-conjuncts <g3>)
   (goal-conjuncts <g3> ^door-closed <g2> ^in-room <g5>)
   (door-closed <g2>  ^door <room5/room6>)
   (in-room <g5>  ^box <b1> ^room <room7>)
   (state <s1> ^door <room5/room6> ^connects <c1> <c3>  ^box <b1>
               ^robot <robot> ^in-room <i1>
               ^door-status <d1>  <d2>  <d5>)
   (door <room5/room6> ^type door)
   (connects <c1> ^door <room5/room6> ^room <room5> )
   (box <b1> ^pushable t)
   (room <room7> ^type room)
   (connects <c3> ^room <room7> <room4> ^door <room4/room7>)
   (door <room4/room7> ^type door)
   (in-room <i1> ^obj <robot> ^room <room5>)
   (door-status <d1> ^status open ^door <room5/room7>)
   (door-status <d2> ^status open ^door <room5/room6>)
   (door-status <d5> ^status open ^door <room4/room7>)
   (operator <o3> ^name close-door
                  ^type robot-domain-op ^door <room5/room6>)
   (operator <o2> ^name push-through-door
                  ^type robot-domain-op ^from-room <room5> ^into-room <room7>
                  ^box <b1> ^door <room5/room7>)
 ;; then:
 -->
   ;; operator <o2> is worse than operator <o3>
   (goal <g6> ^operator <o2> < <o3>))
```

Figure 4.9: The corresponding abstract rule produced for the example task.

and that Room-4 adjoins Room-7 and Room-5 adjoins Room-6. This information was therefore used in making both the abstract and non-abstract comparisons. For example, both abstract and non-abstract search used the information that the robot was in a room next to the door which needed to be opened. However, the location of Box C was not important to the abstract search — the system abstracted away the details of getting the box into a room adjacent to Room-7.

The non-abstract rule also tests for several other conditions in the state. It test that doors Room-3/Room-4 and Room-4/Room-5 are open; and that Room-5 adjoins Room-7, Room-3 adjoins Room-4, and Room-5 adjoins Room-4. In addition, it also tests that Box C is next to door Room-3/Room-4, and that Box C is in Room 3. This information was important in non-abstractly moving Box C into Room7. For example, since the problem-solver did not abstract away the details of getting Box C to a room adjacent to Room-7, its initial room location is now important, as is the fact that it was next to a door.

The additional conditions in the non-abstract rules may make them more accurate than the abstract rules — they may be more likely to be correct in new situations with which they match. However, the additional conditions constrain the future situations in which the non-abstract rule can be used. For example, the additional non-abstract specifications about room connectivity and door status reduce the chance that the rule will be applicable in a new situation with a slightly different room configuration. In contrast, the abstract rule will apply to most situations in which the robot is in a room adjacent to a door which must be opened. In Chapter 6, we report on experiments in which the transfer and accuracy of abstract and non-abstract rules were compared.

4.3.2 Learning MEA Knowledge

Experiments in a robot domain showed that it is possible to use SPATULA to learn means-end knowledge from abstract search, in a manner similar to that suggested by the examples above. In these experiments, the problem solver was not provided with any search control knowledge about its domain. When an operator tie was generated, the system searched abstractly through its space of legal operators until it found one

```
(sp p1644 elaborate
    (goal <g1> ^problem-space { <> undecided <s2> }
          ^state { <> undecided <s3> } ^desired <o1>)
    (problem-space <s2> ^name strips)
    (state <s3> ^robot <r1> ^box <b1> <b2> ^at <a1>)
    (desired <o1> ^task-goals <d1>)
    (task-goals <d1> ^next-to <n1>)
    (next-to <n1> ^box <b1> { <> <b1> <b2> })
    (box <b1> ^type object ^pushable t)
    (box <b2> ^type object ^pushable t)
    (preference <q1> ^role operator ^value acceptable ^goal <g1>
                ^problem-space <s2> ^state <s3>)
    (operator <q1> ^name push-box-to-box
              ^instantiation <i1>)
    (instantiation <i1> ^near-object <b2> ^robot <r1> ^far-object <b1>)
    (at <a1> ^box <b1>)
    -->
    (preference <q1> ^role operator ^value best ^goal <g1>
                ^problem-space <s2> ^state <s3>))
```

Figure 4.10: Example of MEA knowledge learned during abstract search.

which, when (abstractly) applied, could achieve a current task subgoal. The problem-solver then gave that operator a *best* preference, which generated a search control rule encoding the conditions under which the operator was successful.

For example, Figure 4.10 shows a Soar rule which was built when the system discovered that abstractly applying a `push-box-to-box` operator caused the achievement, in the abstract space, of a task goal that two boxes be next to each other.

Essentially, this rule says that:

if	a task goal is to have two boxes next to each other
	and the boxes are pushable
	and there is an operator called "`push-box-to-box`"
	instantiated with the boxes
then	suggest that the operator is "best".

Note that the rule does not test preconditions of the operator — specifically, it does not test whether or not the robot is next to the box it is going to push, or whether this box and the robot are in the same room as the target box. Because the rule was learned during abstract search, these conditions were not important to the result. Thus, using abstraction, the system is able to more easily learn which operators are likely to achieve a given task goal— in fact, the system is learning

MEA knowledge. It would have been much more time-intensive for the system to have learned the rules which would have been produced by a corresponding non-abstract search — though these rules would have been more accurate and tested for the operator's preconditions — since such a search would have involved a blind search to achieve all the operator's preconditions as well. Therefore, such abstract rules can help a knowledge-poor system "bootstrap" itself, by providing information about which operators are likely to reduce a difference. (If more than one operator is suggested as "best", then further problem-solving will be necessary to resolve the tie between such operators.)

Although this approach has some interesting possibilities, it was not pursued to any great extent in our experiments, since knowledge-free searches quickly become unmanageable in domains with a reasonably large set of operators. However, the experiments suggest that the weaker the previously existing domain search control, the greater the relative payoff from the use of abstraction. That is, the less knowledge the system has about its choices, the more effective becomes what is learned from exploring the primary effects of the operators. Thus, in conjunction with other *weak* sources of search control knowledge to help focus the search, this may be a promising area for further research.

4.3.3 Inductive Learning using Abstraction

The abstract rules learned using SPATULA are constructed without modifying Soar's chunking mechanism in any way. Rather, the abstraction process has inductively modified the domain theory so that it allows simpler explanations; *deductive* explanation-based learning over the abstract theory then produces rules which are *inductively generalized* with respect to the original domain theory. Hence, the example demonstrates that deductive learning mechanisms such as EBL can provide inductive concept learning; the theory and operationality criteria used for the explanation can be seen to constitute the system's learning *bias*[4]. SPATULA thus provides the system with the capacity for *knowledge-level* learning [Rosenbloom *et al.*, 1987; Rosenbloom *et al.*, 1991b;

[4]See [Rosenbloom *et al.*, 1992] for an extended discussion of the role of bias in explanation-based learning.

Dietterich, 1986]. Other systems which use this general approach include [Ellman, 1990; Bennett, 1990b; Knoblock *et al.*, 1991] (see Chapter 8).

SPATULA's approach to inductive learning of explanations contrasts with approaches which first produce full problem-solving traces, and then abstract from them (see Chapter 8). Each approach can have advantages. With SPATULA, by abstracting the domain theory *before* learning, the generalization process as well as the problem-solving becomes more tractable.

4.4 Using Abstract Control Rules: Plans

Once a control impasse is resolved using abstract lookahead search, the problem solver is able to choose one of the actions involved in the impasse and continue execution from that point. New abstract search control rules will have been learned in the process, not only for the initial control impasse, but for any sub-searches carried out in service of control impasses which were recursively generated *during* the lookahead searches. The abstract rules — as is the case for all Soar rules — are used in a situated manner. When a rule matches the current situation, it will fire. Hence, abstract and non-abstract search control rules may be applied together if both types of rules are relevant to a situation.

The accumulation of new search control rules may be thought of as the incremental construction and storage of an implicit *plan*. The rules define a sequence of actions for a problem, which may be accessed in situations relevantly similar to the one in which they were learned. If the problem solver encounters a new situation which is somewhat but not entirely similar to a learned-about situation, then some of the rules from the previous situation may apply even if others are not relevant; in this case the plan will be partially utilized. This can occur, for example, if after initial planning a situation changes somewhat. The plan in its entirety would no longer be applicable, but parts of the plan are still likely to apply. In fact, a problem-solver's plan for a task can be considered to be its accumulated search control knowledge relevant to that task, which may have been acquired by learning about many different situations rather than just one. Of course, if two or more sources of applicable knowledge

contradict each other, the system will need to do further search to determine what action to take[5]. Examples of other systems which take a similar approach to plan use include [Minton *et al.*, 1989; Drummond *et al.*, 1992].

If the search control rules for a task are abstract, then the implicit plan used by the problem solver is an abstract one. Since an abstract plan is more general than a corresponding non-abstract plan would be, it is apt to be relevant for a wider range of potential new situations. Because of the way in which SPATULA is incorporated as part of Soar's integrated problem-solving approach, the abstract plans learned from the lookahead searches may potentially be used in any relevant context in future problem-solving — both in the execution space, or during future lookahead search[6].

4.4.1 Multi-Level Refinement and Repair of Abstract Plans

The problem solver's use of SPATULA's abstract plans produces not simply a mapping from one abstract level to the ground space, but an emergent impasse-driven *multi-level* plan refinement behavior, as the planning and execution phases of problem solving are interleaved. Suppose that an operator is selected as a result of a decision during abstract search. Since the search was abstract, it is likely that the problem solver has not yet developed a plan which specifies how all of the operator's preconditions are to be met — that is, its abstract plan has not been completely expanded to the level of executable operators. If this is in fact the case, then an impasse will be generated, and the system will search to achieve the preconditions. It will perform *these* searches using abstraction as well. But this time, its abstractions will be lower-level abstractions, in the sense that they are lower in the subgoaling hierarchy; this time, the abstractions will affect preconditions of operators proposed to achieve other, higher-level, preconditions. The lower-level preconditions often address less important aspects of the task, but this need not necessarily be the case. (This issue is further discussed in the chapters to follow.) With these new searches,

[5]See [Rosenbloom *et al.*, 1992] for a more detailed general discussion of plans and planning in Soar.

[6]The abstract rules learned from lookahead sub-searches can match in the execution space as long as the "single-representation trick" [Dietterich, 1980] is used, such that the task representations are similar in both spaces. We in fact assume here that this will be the case.

more of the plan will be refined, or filled in.

The refinement process is iterative; once the new control impasses are resolved abstractly, and lower-level operators are selected for execution, any unexpanded portions of the plan will again generate impasses, which will again be resolved by searching abstractly. Thus the plan is successively refined; at each refinement it becomes less abstract, and attends to more aspects of the task. The multiple levels of abstraction defined by this process are determined *dynamically*, during problem solving, and are driven by the task context as well as the search control knowledge the problem solver has previously acquired about its task and domain.

Figure 4.11 illustrates the use of SPATULA to refine a plan, with a simple example in which the task goal is to move a robot into a given room, and to push a box into a corner of that room. The initial state and goal for the task are shown in Figure 4.12. In Figure 4.11, the problem-solver first performs an abstract search to determine which operators it will apply to achieve its task goal conjuncts. The search may include different ways to achieve a task goal conjunct (e.g., the robot may be moved to Room 3 while pushing a box, or alone), as well as different orders in which to achieve the goal conjuncts[7].

Suppose that as the result of this abstract search, Plan (1) is formed. It does not include information about how to achieve the preconditions of the `push-through-door` and `move-box` operators, since the search did not address these subproblems. (Although the problem-solver's plans in actuality consist of situated search control knowledge — used to generate partial orders for operator application — they are shown here as operator sequences for ease of illustration). The `push-through-door` operator is selected for execution. This operator can not apply directly, because its preconditions are not met (the robot and box are not at the door, and the door is not open). An operator subgoal is generated (indicated by a triangle), and an abstract search is performed within the operator subgoal to determine how best to achieve its preconditions. As a result of this search, the system's plan for the task is refined as shown in Plan (2). The `push-box-to-door` operator is selected for application, and again it

[7]For this example, we assume that SPATULA's abstract search abstracts all unmet operator preconditions. As will be seen in Chapter 5, this will not always be the case when extensions of the basic abstraction method are used.

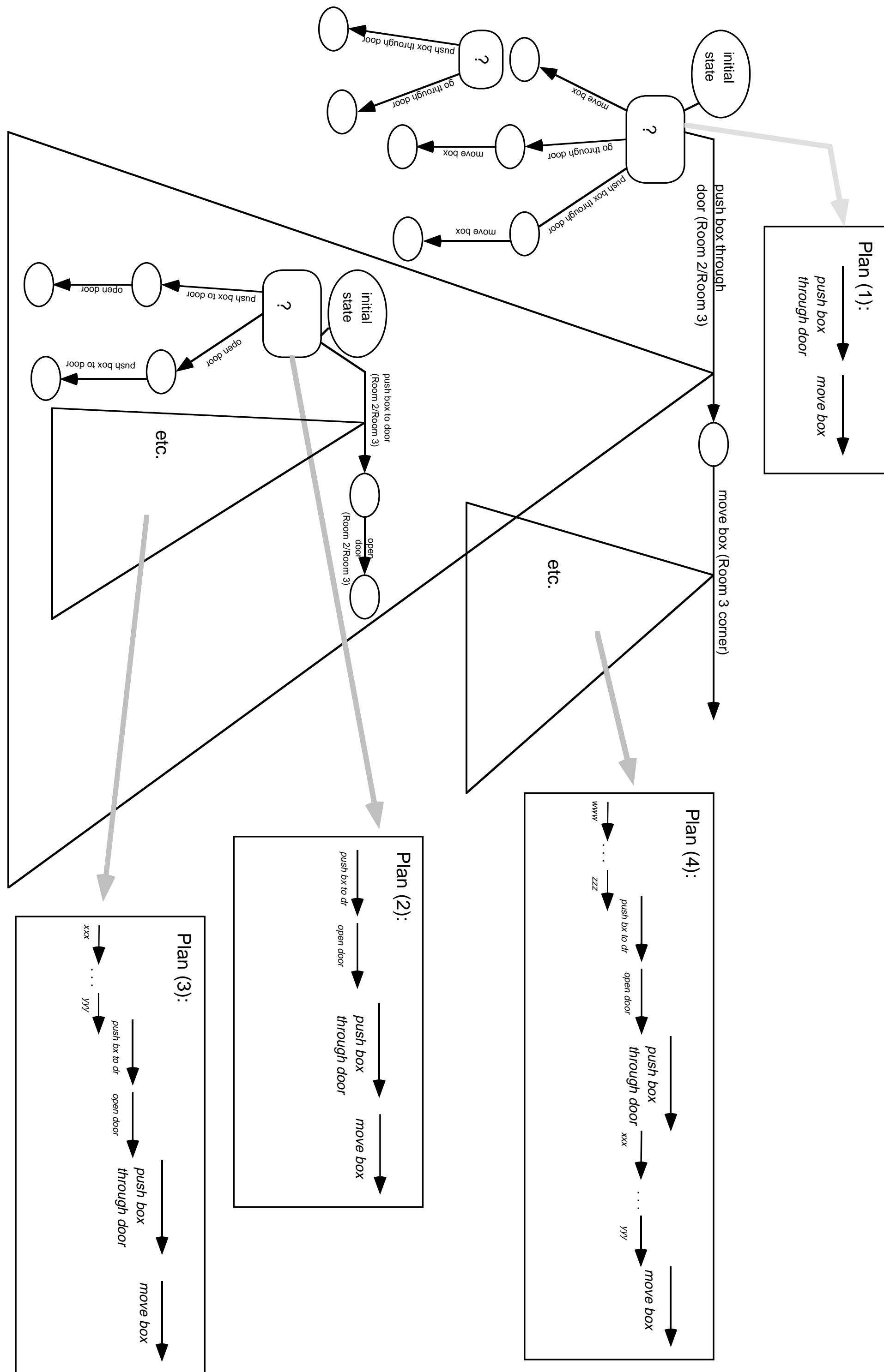


Figure 4.11: Successive refinement of a plan for a simple task in a robot domain.

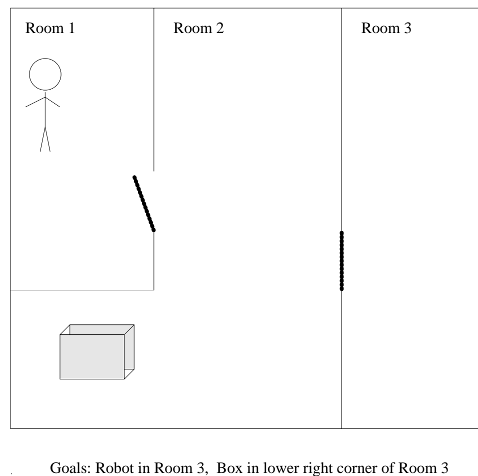


Figure 4.12: Initial state and goal for the robot domain example of Figure 4.11.

can not apply directly. Abstract search is performed within the new operator subgoal to determine how best to achieve the preconditions of the `push-box-to-door` operator, and as a result, the plan is further refined as shown in Plan (3). (The operators used to achieve the preconditions of the `push-box-to-door` operator are suggested by "xxx" and "yyy").

Similar refinements occur as necessary to continue application of the `go-through-door` operator. After it has been applied, the problem-solver's plan suggests that the `move-box` operator be applied next. If this operator can not be applied directly because of unmet preconditions, the plan will be further augmented as shown in Plan (4). (The operators used to achieve the preconditions of the `close-door` operator are suggested by "www" and "zzz"). The plan refinement process continues until the system has acquired sufficient information to achieve its task goals without further search.

By using abstract plans, the problem solver can reduce its search effort. The abstract plans — created with less effort than would have been required for a more detailed search — provide heuristics to constrain search at the less abstract problem levels. If an abstract plan is used to define an ordered set of independent subgoals at a more detailed level, then abstraction can provide an exponential reduction in search [Korf, 1987]. In the next chapter, we will discuss search reduction using SPATULA in greater detail.

The plan/execute cycle of plan refinements described above allows plan *repair* to occur implicitly as the abstract plan is expanded, in the same way that refinement occurs. The system need not distinguish between the two processes. For example, when an abstract plan is executed in detail, it may be that unexpected results of the non-abstract execution will render part of the abstract plan inapplicable. This may occur both because 1) no plans match; or 2) because existing plans are overridden by the recognition of an undesirable situation which must be repaired. In either case, an impasse will be reached, and additional abstract search will be performed to fill in the gaps in the system's knowledge with new rules (more will be said about plan repair in the following section). If the plan gets back on track later on, previously learned search control rules will again apply. In the case of 2), if the problem solver recognizes an operator in the abstract plan as contributing to the unfavorable results, then new search control can be learned to reject that operator in the future when a relevantly similar situation is encountered. These new rules override the previously learned rules [Laird, 1988].

If there is no way to apply an operator selected from the abstract plan during non-abstract execution — for example, because there is no way to achieve the operator's preconditions within an operator subgoal — then it will be rejected (using Soar's default problem-solving knowledge) and a new operator will be selected in its place. The selection of the replacement operator will bring to bear all of the problem solver's additional search control knowledge about the situation; for example, it may have knowledge to suggest that some operator is next best once the original operator has been rejected. The selection process may or may not require additional abstract search.

Therefore, no explicit reasoning needs to be done about gaps or failures of a plan when it is represented as incremental search control knowledge in this way. If existing search control (abstract or non-abstract) does not apply at some point, additional search will be done to acquire new knowledge. If existing search control does not apply when it was expected to, but is applicable at some other point during the problem execution, it will be opportunistically used then. The problem solver can still effect

repairs regardless of whether existing search control is inadequate because a new situation has been encountered, or because a previously learned-about situation was not completely explored or has unexpected facets. Note that Soar's integrated response to abstract plan refinement and repair does not preclude any explicit reasoning about why an abstraction has generated difficulties. If such information is available, it can aid the repair process.

4.4.2 Context of the Abstraction Refinement

The context in which abstraction takes place affects the ways in which abstract plan repair may occur. In the version of SPATULA implemented here, the problem solver is using abstract search to select an *executable* action. In contrast, some other planners use abstraction to guide the construction of a full ground-level plan within a planning search, and then output the full plan for execution (see Chapter 8). In both cases, abstraction is used to constrain search and provide heuristics about what to do next. However, the difference lies in the degree to which a plan is refined before it is used to actually select an action.

This difference is independent of the *particular* abstraction techniques used by a planner (except to the degree to which the abstractions are driven by actual rather than expected results of operator applications). Therefore, SPATULA's methods for automatically generating abstractions may be disjoined from the context in which the abstract plans are used. For example, both the basic abstraction method described in this chapter and Chapter 3, and the method increments to be described in Chapter 5, would work equally well within a method framework which used abstraction to construct a more detailed plan. (Such a framework could also be implemented in Soar, using a different set of default abstraction rules). Nevertheless, the context in which an abstract plan is used impacts the problem-solver's behavior, and hence a description of SPATULA as implemented here includes a discussion of this impact.

As stated previously, SPATULA falls into the class of abstractions called "precondition relaxations", which are themselves a subset of the class of "PI-Abstractions" [Giunchiglia and Walsh, 1990a; Giunchiglia and Walsh, 1990b], or "monotonic abstractions" [Knoblock, 1991]. As described in Chapter 1, it has been proven that

using PI-Abstractions, for every ground-level solution, there is guaranteed to exist an abstract solution such that a monotonic refinement from the abstract to the ground solution may be constructed. This is the case for abstraction hierarchies as well as single-level abstractions.

These results hold for Soar using SPATULA if the task domains are designed according to the guidelines presented in the previous chapter. However, for PI-Abstractions, it is not guaranteed that *every* abstract solution will have such a monotonic refinement. The system's problem solving methods then determine how it handles — or even whether it explicitly detects — such a situation. The non-monotonicity of a refinement is not necessarily problematic. The system may very well find a solution more easily by patching its existing plan — e.g., re-doing an undone precondition — than by backtracking to find an abstract solution which affords monotonic refinement.

However, regardless of the problem solver's approach to monotonicity violations, it may at times be necessary to undo the effects of previous plan steps and take a new approach if a solution is to be reached. The way in which the abstract plans are used determine the system's options in such a situation. If the abstract plan is refined during additional, more detailed, planning search, then the system can backtrack within the planning search. If the problem solver uses the results of its abstract search to select *executable* actions, then it can not backtrack in this sense, since it is executing actions rather than doing hypothetical planning. It may still be able to back up, but it must do this by applying new operators which undo the effects of its previous actions. Both usages of abstract plans have their advantages; domain and task characteristics can determine which is most appropriate.

If a plan is completely refined before it is used (that is, if a ground-level plan is output from the planning process), then, given that the planning domain theory is correct and the current state of the world stable, this plan will be executable. The advantage of this approach is that if the abstract heuristics used during the planning phase lead the planner into difficulties developing the plan, the planner can in fact back up during its planning search and find a different, more successful plan.

If on the other hand, heuristics developed from abstract plans are used to select the problem solver's next executable operation (as with SPATULA's use here), and

potential problems with the plans are not patchable, then the task may fail. As was the case with the guidelines of Chapter 3, problem-space design can influence the extent to which the problem solver is able to patch problems in execution-time plan refinement. During execution, the problem-solver:

- should be able to potentially propose all legal domain operators;
- should have default problem-solving knowledge about how to reject operators which have been selected but can not be applied;
- and should not employ search control equivalent to Soar's *prohibit* and *require* preferences.

With these capabilities, the system is not restricted in its actions by what the problem-space designer thought would happen, and thus is not at a loss in unanticipated situations encountered when using abstract heuristics.

However, execution-space problem solving is still not guaranteed to be complete with respect to the use of abstraction —that is, the problem solver is not guaranteed to be able to find a solution using abstraction in every case for which it can find a solution without abstraction — unless any effects which prevent a solution from being reached can always be undone. If some important mistakes can not be recovered from, then the domain may not be an appropriate one in which to use approximate heuristics in the execution space, whether derived from abstraction or from some other source. (An example of such a domain is intensive-care management, where *deadlines* are exactly that.) In such domains, it may be more suitable to use SPATULA's abstraction techniques to guide further planning search, rather than to guide executable actions as is done here.

However, there are several advantages to using abstract heuristics directly to select executable actions, and such advantages have influenced the implementation of SPATULA described here.

- The abstract heuristics are easier to learn than would be the corresponding non-abstract rules.

- Because the abstract rules are more general, they may be used in a wider range of new situations.

- Domain theories may not be completely correct, and/or the world may change because of inputs beyond the problem solver's control. In these cases, the effort needed to produce a complete and executable plan is often wasted because the plan is not usable in its entirety once it is produced. Abstract plans are more likely to remain usable given small situational changes.

- It may be the case that some lower-level plan information is not available during the planning process. As a simple example, when planning a bus trip, it may be the case that the bus number and boarding location is not yet known. Nevertheless, an abstract plan to board a bus can be constructed, with the actions for finding and boarding the specific bus filled in later.

- Even if the domain theory is correct, it may be a waste of time to fully expand the plan. It may be more efficient to deal with lower-level plan details in a reactive manner. (See Chapter 8).

Under such conditions, an abstract plan can be used as a guide to the actions the problem solver should take; the plan will be refined as execution proceeds, and this refinement will be able to take into account any unanticipated developments in the external environment. As discussed above, the Soar architecture is well suited to such an interleaved planning/execution approach.

4.5 Summary

In this chapter, we have described the way in which SPATULA's default knowledge about how to perform abstract search is used by an impasse-driven problem solver such as Soar – in conjunction with abstract-problem-space creation techniques such as those described in Chapter 3 — to produce a basic framework for the use of abstraction as a general weak problem-solving method.

Within this framework, abstraction is not employed unnecessarily — abstract searches are performed only when the problem solver does not have access to other search control knowledge which would tell it what to do next. The abstract searches allow control impasses to be resolved more efficiently, and produce abstract search heuristics. The abstract rules are inductively generalized as a result of learning from search over an abstract theory; because the abstract search is simpler, the rules are easier to learn. The learned abstract rules serve as abstract plans; the plans constrain more detailed search. Plan refinement is interleaved with execution and driven by the needs of the problem solver and its acquired knowledge. As the abstract plan is refined, a multi-level abstraction behavior emerges. The abstraction levels are determined dynamically by the problem-solving context. Any necessary patching or abstract plan repair is integrated with the plan refinement process.

Although SPATULA's basic abstraction framework was presented in conjunction with a particular method for abstracting a problem space, we expect that the framework will extend to use with other problem-space abstraction methods as well. The general approach — that of using abstract search to learn and refine abstract plans — is not dependent upon the use of a specific class of abstractions.

Chapter 5

Abstraction Method Increments

5.1 Introduction

In Chapters 3 and 4 we have presented an integrated abstraction method which allows an impasse-driven problem solver to automatically create and use abstract plans. This basic abstraction technique can be useful as is. For example, it can be helpful if the problem solver must choose among a large number of operators and does not know which ones advance it towards its goal. In this case, abstraction permits the problem solver to more easily guess which operators are likely to achieve an aspect of the current goal and which are likely to be unrelated. An example of this use of abstraction — in which the problem solver learned new MEA knowledge — was presented in Section 4.3.2.

Using this basic abstraction method, one abstract search path will be chosen over another if it has a higher domain evaluation. For a domain with goal-driven search control knowledge, in which operators are proposed in response to unachieved subgoals, and which has a domain evaluation based on solution length, the basic abstraction method will prefer one search path over another when:

- one operator has a net achievement of more goals than another operator (this includes situations in which opportunistic achievement or clobbering of a second goal occurs while applying an operator in service of a first).

- one way of achieving a goal requires a longer series of operators than another.

These conditions may not arise very often using SPATULA's basic abstraction method. Conditional or opportunistic effects may not be observable for abstractions of the high-level operators proposed to achieve task subgoals, and often the sequences of these high level operators proposed to achieve a goal are of the same length. Therefore, if a problem-solver is using goal-driven search control, it may not be the case that abstract search, using SPATULA's basic abstraction method, will provide much new information about which path is best. Too much information will have been abstracted away for the purposes of the task.

To address the problem of making more discriminate abstractions without requiring domain-specific knowledge, we have added a set of abstraction *method increments* to SPATULA. Method increments augment and modify the behavior of previously existing problem-solving methods, by adding additional knowledge to the methods. With SPATULA, the method increments take the form of additional default rules. These method increment rules, when added to the basic abstraction method knowledge of the previous chapters, produce new abstraction methods. The new methods use more information about the global problem-solving state than does the basic method, though they still make only limited demands on knowledge about the problem domain.

Two of the abstraction method increments enable the problem solver to obtain more information about its problem-solving context, and make better situated judgments about what would be useful abstractions for a particular task or subtask. As will be seen, both make these judgments by using relative — rather than absolute — information about the control decision options and problem-solving context. These two method increments are called *iterative abstraction* and *assumption counting.*

A third method increment is called *extended plan use.* It allows the system to deliberate about the extent to which it wishes to use its abstract plans during execution, using a heuristic which states that plans learned under certain circumstances are likely to be less useful than others. This method increment is used in conjunction with iterative abstraction. The last two method increments are driven by efficiency motivations; they provide heuristics for restricting the abstract search, yet still producing useful

8. For each new lookahead search (including sub-searches of top-level lookahead search) initialize the **assumption-count** for that search to 0.
9. If a subgoal is **in-abstraction-context**, and there is an impasse at an operator application and a precondition is not met, then add a flag that the precondition is abstracted. (This rule will fire at the same time as Rule 7 of Figure 4.4).
10. Each time a precondition is noted as abstracted, increment the **assumption-count** for that lookahead search.

Figure 5.1: Default knowledge for the *assumption counting* method increment.

decisions. These are called the *abstraction-gradient* and *goal-achievement-iteration* method increments.

The method increments may be used singly or in conjunction with each other, although some are null methods if not used with others. As will be shown below, they can interact synergistically with each other to produce better abstractions than any would have produced alone.

5.2 Assumption counting

The *assumption counting* method increment combines existing domain evaluation criteria with a new meta-evaluation based on the number of assumptions, or abstracted preconditions, required to complete an abstract search. The new combined evaluation criteria are used instead of the domain evaluation to compare the results of the abstract lookahead searches for each option.

In the current implementation of SPATULA, the evaluation criteria are combined *lexicographically*. To compare two options, the problem-solver looks first at the number of assumptions made during abstract search and compares the domain evaluations only if the number of assumptions are equal.

The method increment is implemented by adding knowledge to the system about

how to note and count abstracted preconditions, and how to combine this information with the domain evaluation. Figures 5.1 and 5.2 summarize the long-term-memory rules in Appendix A which provide this knowledge to Soar. The rules in Figure 5.1 provide the knowledge to allow the system to keep track of the assumptions made during a lookahead search; the rules in Figure 5.2 then tell the system how to perform evaluations given that knowledge. (Figure 4.2 may again be referred to for a definition of some of the terms used for the rules). These rules build on the rules listed in Section 4.1, by making use of the information about whether the system is in an abstraction context. Note that if abstraction is not used for an option's search, or if no assumptions are made during abstract search, then the number of assumptions for that search will be zero. Thus, the method increment is robust with respect to whether or not abstraction is used for a given part of the task; in a comparison of two options for which abstraction was not used, the system will simply base its preferences upon its domain evaluations.

Although assumption counting is simple, it provides the problem-solver with a surprising amount of leverage. It provides a measure (though not an exact one) of the difficulty of instantiating the abstract plan discovered during the search. In addition, as is shown below, it allows the system to detect goal interdependencies, by letting it observe the relative number of assumptions required for various operator sequences.

5.2.1 Example of Assumption Counting

Consider an example, again from an ABStrips-like robot domain, in which two boxes need to be moved to new locations. The diagram of Figure 5.3 shows the initial state and desired goal state — the goal locations of the boxes are shown in gray. The preconditions and effects of the relevant operators — `push-to-loc` and `push-to-room` — are listed. We assume the domain problem space is designed according to the guidelines presented in the previous chapter. To simplify the example, we also assume that the problem solver has been provided with MEA search control knowledge about its domain. However, the efficacy of the assumption counting method increment is not linked to any particular type of search control, as further discussed in Section 5.5.

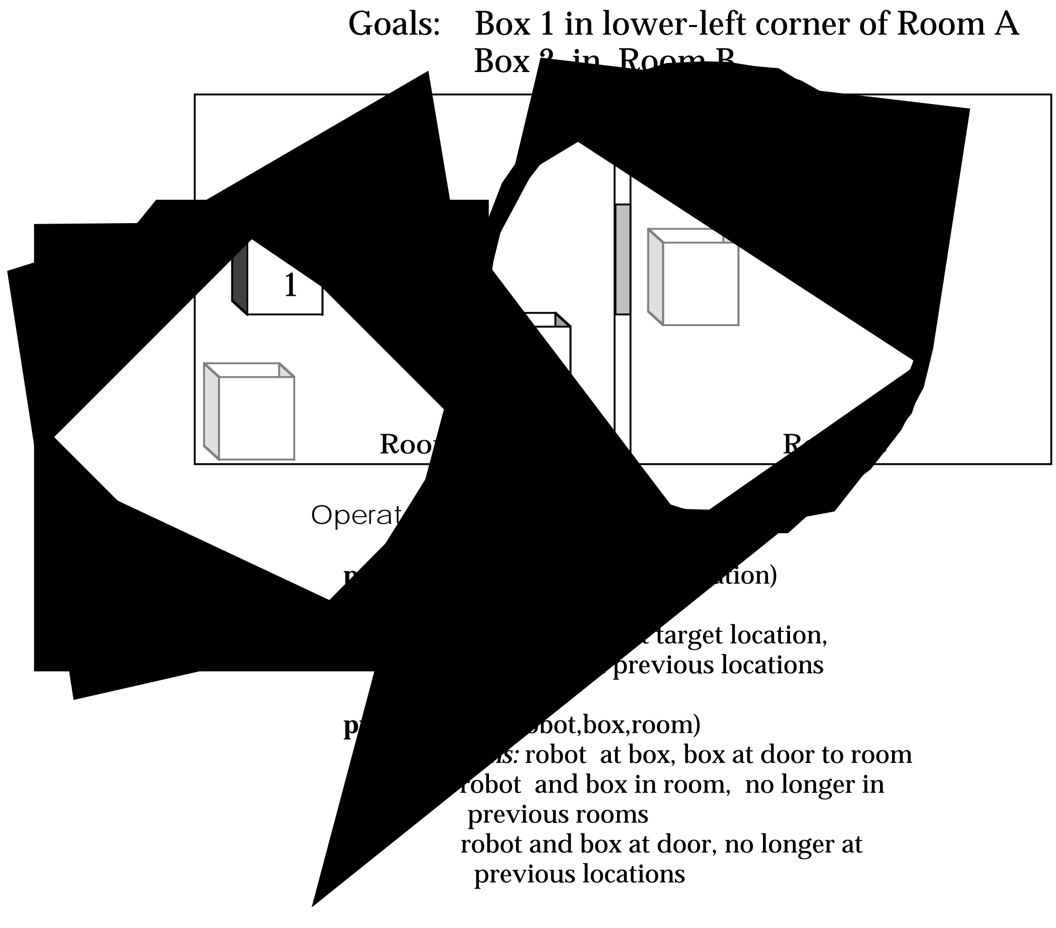
Replace those system default rules which tell the system how to give preferences to operators when their lookahead search domain evaluations are different from each other, with the following rules. The new rules implement the lexicographic ordering of the current method of combining assumption counts with domain evaluations; if the combination function is changed, these rules must be changed.

11. When lookahead search success is detected (by whatever domain criteria), create a combined evaluation which includes both the domain evaluation and the **assumption-count**. This must be done for both operator-subgoaling success (i.e., the system has found a state from which a subgoaled-upon operator can apply) and more general lookahead search success, in which the system has reached a state from which it can evaluate one of the options in its top-level control impasse.

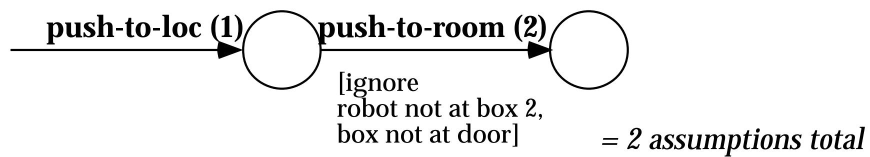
12. If the combined evaluation for one operator search has a lower **assumption-count** than the combined evaluation for another operator search, create a *better* preference for the first operator.

13. If two combined evaluations have the same **assumption-count**, but different domain evaluations, then prefer the operator with the best domain evaluation (with respect to the system's knowledge about domain evaluations).

Figure 5.2: Modifications of existing default evaluation knowledge for *assumption counting* method increment.



Lookahead Searches:



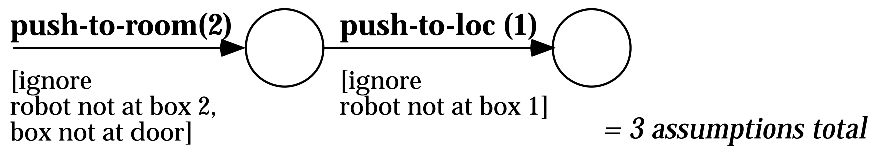


Figure 5.3: Example of utility of assumption counting.

Given this task, the problem solver will reach an impasse when it tries to determine which goal conjunct to work on first. Using SPATULA, it will perform abstract lookahead searches to decide how best to achieve the task.

The problem solver's two lookahead searches are shown at the bottom of Figure 5.3. Unmet preconditions are shown in brackets. When all unmet preconditions are abstracted, the searches simply involve trying the two operators in either order. If domain evaluations are based on solution length, then without assumption counting, both operator sequences will look equally good.

However, when assumption counting is used, the system is able to decide that the first of the two sequences, in which it pushes `Box 1` first, is best. It can guess by its comparison (without knowing the semantics of the precondition tests) that if the operators are applied in the reverse order, the `push-to-room` operator undoes a previously achieved precondition of the `push-to-loc` operator, and hence it estimates that less work is required to push `Box 1` first.

Thus, assumption counting can give the problem solver information about the interactions occurring between the task operators[1], as well as an estimate of how difficult each operator will be to achieve. It does not require semantic knowledge about the operator preconditions to make these estimates. Note that the utility of the assumption counts comes from the *comparisons* between the operators; as an absolute measure, the counts would not provide much information. Assumption counting implicitly uses the heuristic that the abstracted preconditions of the options being evaluated are of approximately the same importance with respect to difficulty of achievement and/or impact on other parts of the task (so that it makes sense to compare them). This heuristic is further discussed in Section 5.5.1.

With the assumption counting method increment, if no assumptions were made during an evaluation search, then the problem solver effectively uses the domain evaluation function by itself, since the assumption count is zero. Therefore, this method may be used consistently by the problem solver regardless of whether or not a lookahead search is abstract, or whether or not the problem space is represented

[1] Here, "interaction" is used not just to mean the cases in which one operator *undoes* a previously achieved subgoal, but also cases in which the application of one operator in service of one subgoal makes another subgoal harder or easier to achieve.

such that assumptions can be counted.

We are not convinced that the lexicographic ordering is always the most useful way to combine the assumption count with the domain evaluation. For example, if one abstract solution path is twenty steps longer than another (to pick an arbitrary number), but has one fewer assumption, then it may not be useful to pick the path with fewer assumptions unless the ignored preconditions are expected to take at least twenty steps on average to achieve. However — as will be seen from the empirical tests described in Chapter 6 — the lexicographic ordering proved to be a useful domain-independent measure. Our investigations suggested other potentially useful combination policies as well; these will be discussed later and are topics for future work.

5.2.2 Factorization: Operator Precondition Testing

To make use of the assumption counting method, the problem solver must be able to detect whether or not an operator's preconditions are met, for each precondition individually. Therefore, the *factorization* guidelines of Chapter 3 must be followed. In particular, precondition testing must not only be factored from operator application, but the test for each precondition must be made separately; that is, must be contained in a separate rule. Then, the problem solver is able to discern which preconditions are met and which are not, and thus keep a count. (In contrast, one big test for all preconditions would, if it failed, only indicate that at least one of the preconditions was not met, and would only provide information about the number of operators along each path for which assumptions were made.)

5.3 Iterative Abstraction

SPATULA's next method increment is called *iterative abstraction.* Iterative abstraction is based on the heuristic that if the problem-solver can't distinguish between the results of evaluating two or more options at a control impasse — if it can't tell which is best — then it's operating at too high a level of abstraction. To state this another

To resolve a control decision impasse using iterative abstraction:

- Set the variable `abstraction-level` to its most abstract value.
- Search to evaluate each control decision option at `abstraction-level`.
- As the result of search, can any options be distinguished as worse than others? If so, create preferences to that effect, thus removing those worse options from consideration in the control impasse.
- As the result of search, are there any options whose evaluation searches required no assumptions, and which appear equally good? If so, give them preferences which state that they are *indifferent* to each other.
- If only one option is left, or all remaining options are indifferently preferred, the control impasse will be resolved.
- Else, decrease `abstraction-level`, and go to 2.

Figure 5.4: The general algorithm for iterative abstraction.

way, iterative abstraction implements the heuristic that a useful level of abstraction for a situation is that at which the system can discriminate among the situation's options.

The iterative abstraction technique uses the general algorithm shown in Figure 5.4. It may be summarized as follows:

> The problem solver first tries to resolve a control decision by evaluating the options involved in the control impasse at a high level of abstraction. If the evaluations provide insufficient information to completely discriminate between options, then — rather than making a random choice — the problem solver iterates by re-evaluating those options which looked the best at an increased level of detail (it does not reconsider those options, if any, which looked worse). The iteration cycle is continued, with level of detail increasing at each iteration, until the problem solver is able to distinguish between the remaining options or ascertain that they are equivalent for its purposes. At this point a decision is made and the control impasse is resolved.

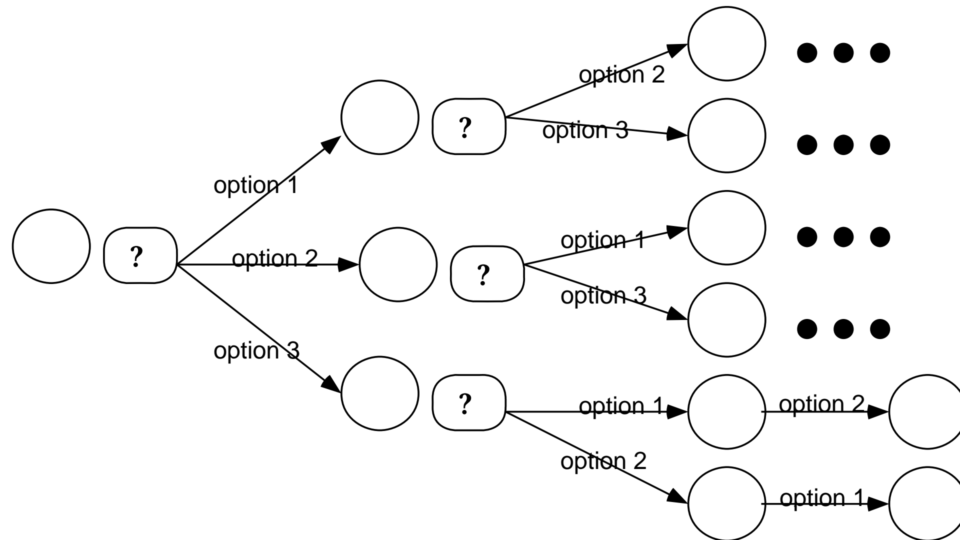


Figure 5.5: Iterative abstraction; first iteration level lookahead searches for each of three options.

Note that as stated in general terms this algorithm makes no assumptions about the abstraction method that the problem solver is using, and does not require that the problem solver use the method described in Chapter 3. For example, if the problem solver has domain knowledge of any type about what preconditions to abstract and how to increase the level of problem solving detail, the iteration algorithm can be used. If the problem solver has analogous knowledge about abstracting operator *implementations*, rather than (or in addition to) operator preconditions, again the algorithm can apply.

However, for the purpose of this research, the concept of "increased level of detail" is operationalized with respect to the domain-independent precondition abstraction of Chapter 3 in the following way. The problem solver first attempts to evaluate the options of a control decision by abstracting away all operator preconditions in the evaluation searches for that control decision. Figure 5.5 shows an example of such lookahead searches. If each operator being considered (Options 1–3) achieves a different goal conjunct, then the system still must search to determine how best to order the operators (it may have other search control or methods which obviate the

need for an exhaustive search). Although it is not shown in the figure, it may require more than one operator to achieve a goal conjunct, and/or the system may also need to search to discover the best way to achieve a goal conjunct as well. However, the system does not need to search to achieve the operators' preconditions, since they are abstracted. This first-iteration-level search may suggest that some candidate options are worse than others, particularly if other method increments, such as assumption counting, are used. If one option is not clearly best (or good enough by some domain criteria), the problem solver iterates with the best candidates.

At the next iteration, the abstract look-ahead searches are performed in more detail by requiring the problem solver to achieve those operator preconditions (call them "set A") which it abstracted away before. This is done for all operators being considered in the control decision. Thus the task is considered in more detail. However, in this new iteration the problem solver will still abstract away the preconditions of the operators which are applied to achieve the preconditions in set A. (Call these newly abstracted preconditions "set B"). Because set A was abstracted in the previous iteration, there was no possibility of even considering set B previously.

Figure 5.6 shows the second-iteration-level lookahead search which begins by applying Option 3 of Figure 5.5 (the searches which begin by applying Options 1 and 2 would be analogous). At this iteration level, the system must now achieve the preconditions of Option 3. Suppose that its preconditions are achieved by applying Options 4 and 5. The problem-solver may first need to search to determine the best sequence for Options 4 and 5. For any operator ties generated in the process of these searches, equal evaluations will in fact generate preferences stating that the operators are equal to each other. That is, within a search at a given iteration, sub-searches do not iterate recursively — the search stays at the same iteration level all the way through. This way, any larger-grained differences between options may be discovered without spending a lot of time on more detailed distinctions.

Once it has selected an ordering for precondition achievement, the problem solver applies Option 5, then 4. However, these operators are themselves applied *abstractly*; their unmet preconditions are assumed met. After Options 4 and 5 are abstractly applied, then Option 3's preconditions will be met and it may be applied. (Though its

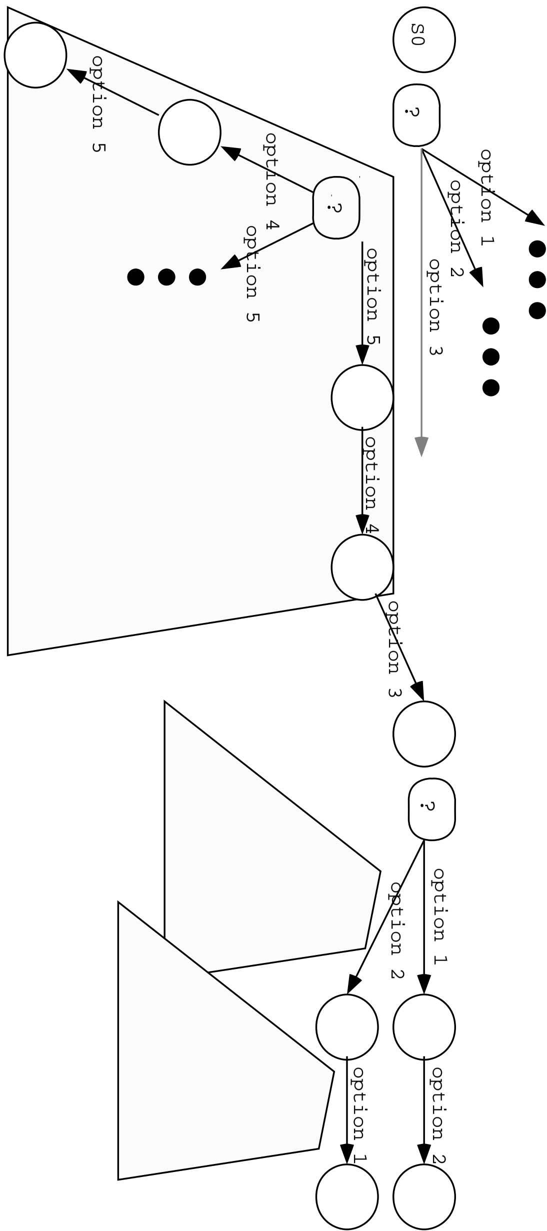


Figure 5.6: Iterative abstraction, second iteration level.

preconditions are met, its *application* may be abstract as a result of the abstractions of Options 4 and 5). The remainder of the searches proceed in the same way, with the preconditions of Options 1 and 2 now being considered.

If the problem solver can not yet make a decision at this second level of abstraction, the iteration process continues. However, if any of Options 1–3 looks worse than the others, it will be eliminated from further iterations at this point. In the next iteration, the preconditions of Options 4 and 5 must now be considered, and so on.

An important effect of the iteration process is that at each iteration, those options which are unrefinable at that level of abstraction (e.g., because there is no way to achieve a necessary precondition) will fail and be rejected. Thus, at each iteration level, the task context's *critical* preconditions for that level are discovered. The system is able to determine these critical preconditions without searching all paths in full detail.

5.3.1 Implementation

Implementation of the iterative-abstraction method increment consists of providing the problem solver with the following new abilities: counting abstraction iterations, counting operator subgoal levels, specifying when it is appropriate to abstract a search based on this information (i.e., specifying when the problem solver is in an abstraction context), and specifying how to eliminate options from the iteration process via their evaluations. As implemented for our experiments, the iteration process will halt when one option has a better evaluation than all others, or when options are shown to be non-abstractly equal. Other halting conditions are possible, and are discussed later.

Figures 5.7 and 5.8 summarize those default rules of Appendix A which provide this knowledge to the system. The first rule of Figure 5.7 tells the system how to count the operator-subgoaling levels generated by the system. The next two rules tell the system when to iterate, and how to keep track of the iteration level. As with the assumption counting rules, these rules build on the basic abstraction method knowledge of Section 4.1, and are described using the terms of Figure 4.2. They allow the system to change the values of the variables which tell it when it is in an abstraction context — with each iteration, the system must generate an additional

14. For each operator subgoal initiated during lookahead search, increment the **level-count** variable of the parent goal to create a new **level-count** for the new subgoal. For each non-operator-subgoal generated, copy the **level-count** value unchanged.

15. Note when all options in a top-level operator control impasse have been evaluated; if they have, then iterate, by setting up the state in the control impasse subgoal for a new round of evaluations. (If the system *was* able to discriminate among all options then the control-impasse subgoal would be architecturally resolved and this rule would not match. Those options judged worse will be removed from the impasse, and the next iteration will only evaluate those remaining.)

16. At each iteration within a control impasse, increment the **abstract-at-level** value. (This then influences when Rule 5 of Section 4.1 may fire).

Figure 5.7: Default knowledge for the *iterative abstraction* method increment.

Replace the existing Soar default knowledge about handling equal evaluations with the following rules.

17. If the evaluations for two operators are the same, and the search is a *lower-level* lookahead search, then add a preference which states that the two operators are *indifferent* to each other.

18. If the evaluations for two operators are the same, and no abstractions were made during search, then add a preference which states that the two operators are *indifferent* to each other.

19. If the evaluations for two operators are the same, and the search is a top-level lookahead search, and abstractions *were* made during search, then create *no* new preferences between the operators.

Figure 5.8: Modification of default long-term-memory evaluation knowledge for *iterative abstraction* method increment.

operator subgoal level before it can begin to abstract.

The rules of Figure 5.8 modify the system's set of evaluation rules so that it now requires that control decision options be discriminable (or non-abstractly equal) before it generates preferences to resolve the control impasse.

5.3.2 Examples of Iterative Abstraction

The iterative abstraction process may be further instantiated by considering the example of Figure 5.3, but with a slightly different initial state, as shown in Figure 5.9. The robot has the same goals, but it is no longer next to either box. Therefore, its first-iteration-level search (abstracting all unmet preconditions) does not provide enough information (even with assumption counting) for the system to distinguish between its proposed operator sequences. This first-level search is shown in Figure 5.10. Hence, the system iterates and considers the problem in more detail.

Figure 5.11 shows the lookahead searches produced with iterative abstraction at

Goals: Box 1 in lower-left corner of Room A
Box 2 in Room B

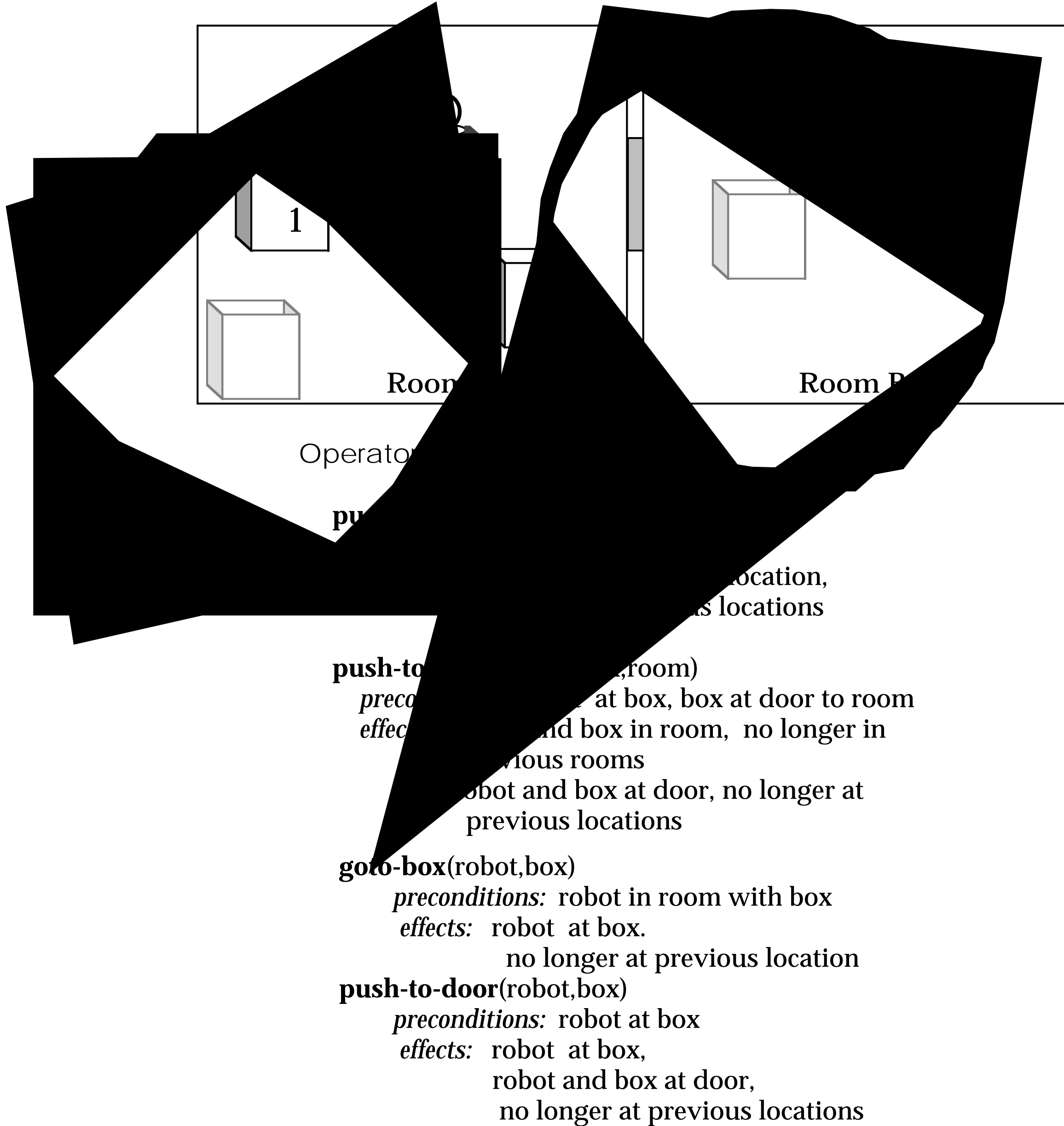


Figure 5.9: Example of iterative abstraction.

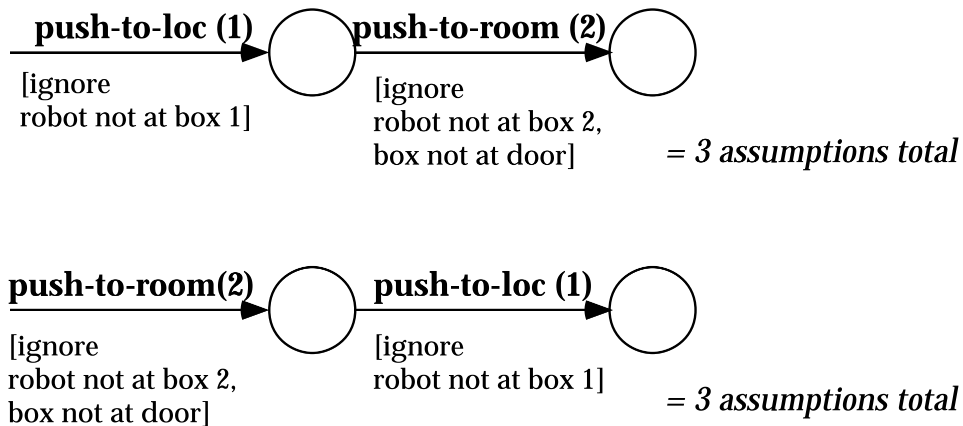


Figure 5.10: First-level abstract lookahead search for iterative abstraction example.

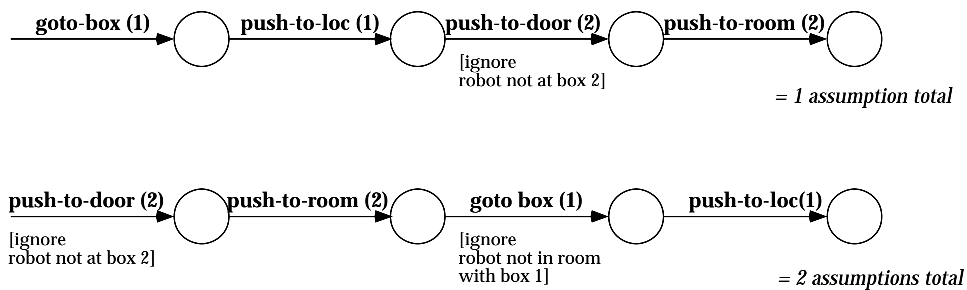


Figure 5.11: Second-level abstract lookahead search for iterative abstraction example.

the second level, using the additional operators described in Figure 5.9. At this iteration, one level of preconditions is achieved by the `goto-box` and `push-to-door` operators, but *their* unmet preconditions are now abstracted. In this example, each second-level search still produces solutions of equal length (this need not always be the case), but if the number of assumptions is compared, the system is able to determine that the upper solution appears easier; by first pushing `Box 1`, it does not need to perform the extra work of coming back from the adjacent room.

Thus, this task provides an illustration of both the way in which the iteration process can be used to discover a useful level of detail for a particular context, and the way in which assumption counting and iterative abstraction can work together; without assumption counting, the system would require further iterations to discriminate between options.

5.3.2.1 Eight-Puzzle

A second example shows the utility of iterative abstraction in a different domain — the Eight-Puzzle, as described in Appendix B. In this simpler domain, there is one operator, which moves a tile to an adjacent cell. The operator's precondition is that the adjacent cell be empty. For this example, the problem solver uses search control knowledge which suggests that tiles be moved to adjacent cells in a direction towards their goal position, and that unobstructed moves onto the blank be preferred before moves onto currently occupied cells.

For the example task shown in Figure 5.12, abstract searches at the first iteration level (that is, using the "basic" abstraction method) do not provide any information about which initial move — the 4 tile or the 8 tile — is better[2]. Both require nine more abstract moves to reach the goal. Unmet preconditions that a cell be clear are ignored, so some tiles are placed in occupied cells, e.g. the 8 and 2 tiles in the second search.

Figure 5.13 shows the searches at the second iteration level. Now, for the search

[2]An initial move of tile 2 is not shown in the example, but would produce the same evaluation as the tile 8 move.

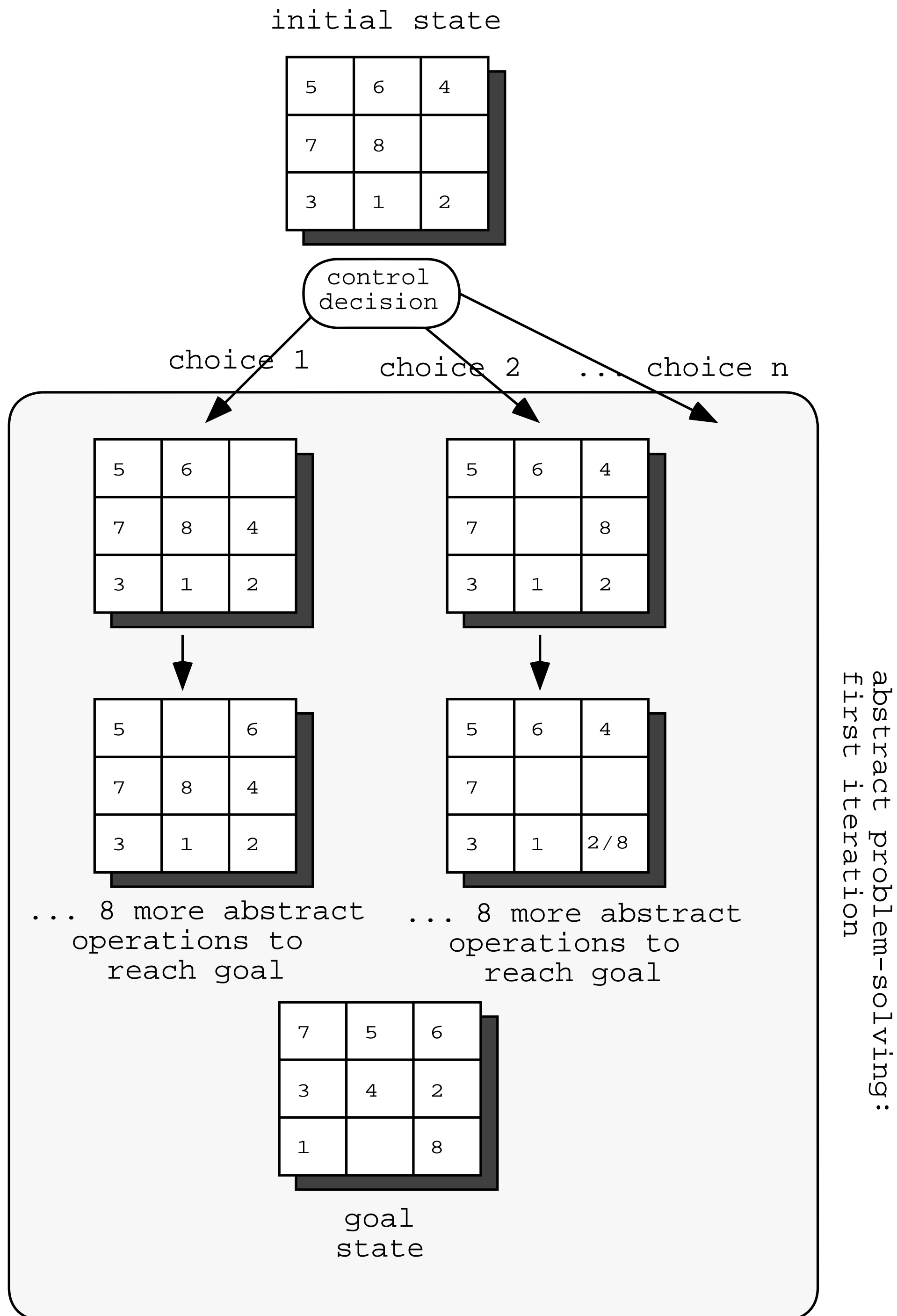


Figure 5.12: Eight-Puzzle lookahead search using iterative abstraction: first level.

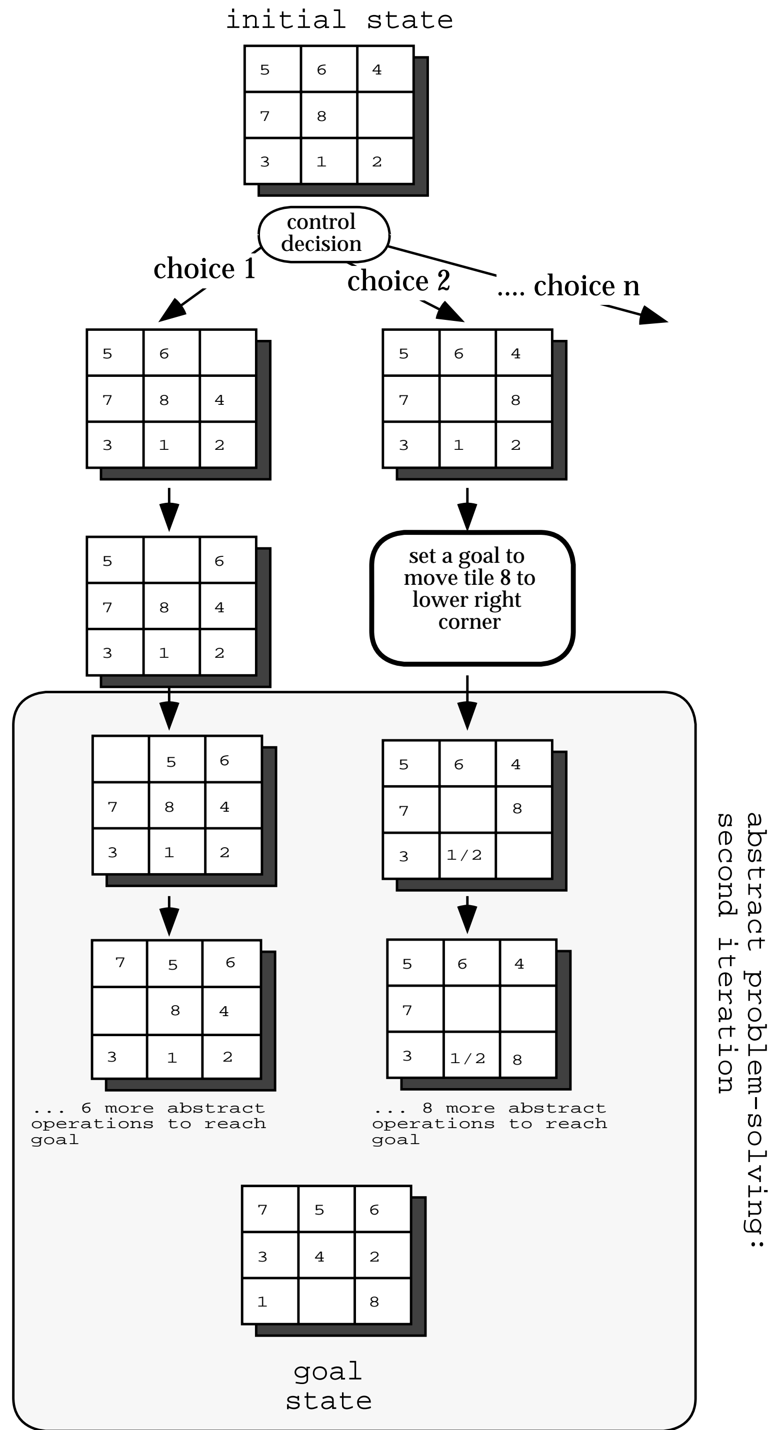


Figure 5.13: Eight-Puzzle lookahead search using iterative abstraction: second level.

on the right, the problem solver must achieve the first level of unmet operator preconditions and clear the lower-right cell before moving the 8 tile there. However, any unmet preconditions encountered while doing the clearing are themselves ignored, e.g. the 2 tile is placed with the 1 tile.

At this level of abstraction, the system is able to distinguish between the two searches– the second one now takes a greater number of steps, since the necessary displacement for tile 8 moved another tile further from its goal. Using iterative abstraction, the system can estimate that this second search will take more work without needing to expand the search to its full detail (that is, without needing to perform the cascade of moves necessary to clear the bottom middle cell for the 2 tile move.)

Although it was not used in the example, the assumption counting method increment would also have suggested that the first search was the easier of the two.

5.3.3 Learning Multi-Level Abstract Plans With Iterative Abstraction

The use of iterative abstraction impacts not only the choices made at control decision impasses, but the information learned from those impasses, and allows the system to draw on a source of multi-level abstraction different from that described in Section 4.4.1.

With iterative abstraction, the abstraction level required to discriminate between a pair of options is context-sensitive. As discussed in Section 4.3, a search control rule is learned each time such discrimination takes place. With iterative abstraction, the learned rules reflect the level of abstraction used to make a particular decision. Since at each iteration at a control decision some options may be ruled out, search control rules may be learned at varying levels of abstraction during the process of making a single control decision. Thus, the use of iterative abstraction may generate a plan which is more abstract in some portions than in others. Then, once a decision is made using iterative abstraction, multi-level refinement occurs as described in Section 4.4.1 to fill out, or refine, the plan acquired so far. At the points in which the plan is less

abstract, less refinement will be necessary.

5.3.4 Discussion

The iterative abstraction method lets the system invest more effort in the resolution of a control impasse — to increase the chance that it may then be effectively and efficiently refined — while attempting to keep the search as abstract as possible to increase tractability. That is, the system uses iteration to try to find a useful balance between cost and accuracy. As was the case with the assumption counting method increment, it is the *comparison* of the different operators being evaluated which provides the problem solver with its leverage for guessing which precondition level is the best; relative rather than absolute information is important. Several iteration levels may be useful within the context of a single control impasse– one level of abstraction may indicate that a particular set of operators is worse than another set; but further iterations may be necessary to distinguish between the operators of the better set. The iteration process also provides the system with the ability to detect unrefinable search paths (e.g., paths for which there is no way to achieve a necessary precondition) without exploring all paths at full detail. At each iteration, it is able to determine the "critical" preconditions for that level.

As with assumption counting, iterative abstraction is a heuristic only, and will not always produce useful decisions. Results are influenced by the levels of abstraction created during iterative abstraction, and domain search control as well as problem representations can affect these levels. Recall that Soar is not restricted to using any specific search control; its default operator subgoaling behavior is independent of any particular search control used within the operator subgoal. Therefore, it need not use pure MEA knowledge to achieve unmet operator preconditions as was done with the robot domain examples — or any MEA knowledge at all. So, for example, given an unmet precondition, domain search control might suggest a single higher-level operator to achieve that precondition, where the suggested operator has further unmet preconditions of its own. Alternatively, the search control could instead suggest the application of a series of lower-level operators to achieve the precondition, where the lower-level operators have no unmet preconditions themselves. These two sets

of search control will produce very different abstraction hierarchies when iterative abstraction is used. This issue is further discussed in Section 5.5.

Iterative abstraction bears some similarities to the technique of iterative depth-first search, or Iterative-Deepening-A*[Korf, 1985a]. With iterative deepening, a sequence of depth-first searches are performed, with the depth of the search increasing at each iteration until a solution is found or resources are exhausted. If resources are exhausted, an evaluation can be made based on the deepest search completed. The idea behind iterative deepening is that the cost of each new search will tend to be exponentially more expensive than the cost of the previous, and thus relatively little extra effort is expended — iterative deepening expands the same number of nodes, asymptotically, as A* on an exponential tree. However, it uses only linear space, thus avoiding the storage cost required for breadth-first search.

With iterative abstraction, the iteration is on abstraction level rather than search depth, and the termination conditions for the iteration process use different criteria — the system is attempting not to find the most detailed solution with the resources available, but to find the abstraction level that allows a good tradeoff between accuracy and efficiency. However, it shares the motivation that each abstract iteration is of relatively small cost compared with the next more detailed search. In addition, iterative abstraction has the potential to provide a capability for resource-limited planning. At each iteration, the system tends to generate more accurate preferences about its options, and may eliminate some of the candidates. If it must stop planning before it is able to completely discriminate among its choices, it can pick randomly among those options which currently look the best, and make use of the abstract plans available at that point. This capability has not yet been tested empirically, but is an important area for future research.

5.4 Extended Example: Iterative Abstraction and Assumption Counting

Iterative abstraction and assumption counting can work together to allow more accurate and efficient abstract evaluations than either method increment would have provided by itself. This need not necessarily be the case; however, our experiments suggest empirically that the two method increments provide better search control when used together rather than singly, since they provide complementary sources of information. The following example illustrates such an interaction, and shows in more detail the structure of an abstract search.

Consider a task with four goal conjuncts, again from an ABStrips-like robot domain. The particular domain presented in this example is used for some of the empirical tests described in Chapter 6. The task takes place in a series of rooms with connecting doors, with several boxes and a robot at various locations in the rooms. A complete description of the domain operators is given in Appendix D, but the description will not be necessary to understand this example. The initial state and goals for the task are shown in Figure 5.14.

In this example, the problem solver possesses MEA knowledge about its domain such that it suggests operators which it believes will achieve the goal conjuncts. Since all such operators will appear equally good given only the MEA search control, the operator tie shown in Figure 5.15 is generated. The problem solver then performs abstract lookahead search, using both iterative abstraction and assumption counting, to decide which operator to select. The domain evaluation function prefers solutions with fewer steps.

The problem solver begins by performing lookahead searches for the tied operators at the first abstraction iteration level, in which it abstracts away all unmet preconditions of the operators in the tie. This proves to be too abstract; although it is not shown here, all operators will look equally good at this highest level of abstraction. (Therefore, for this task, use of the assumption counting method increment alone would have been no more useful than choosing randomly.) Because it can not make a decision, the problem solver iterates.

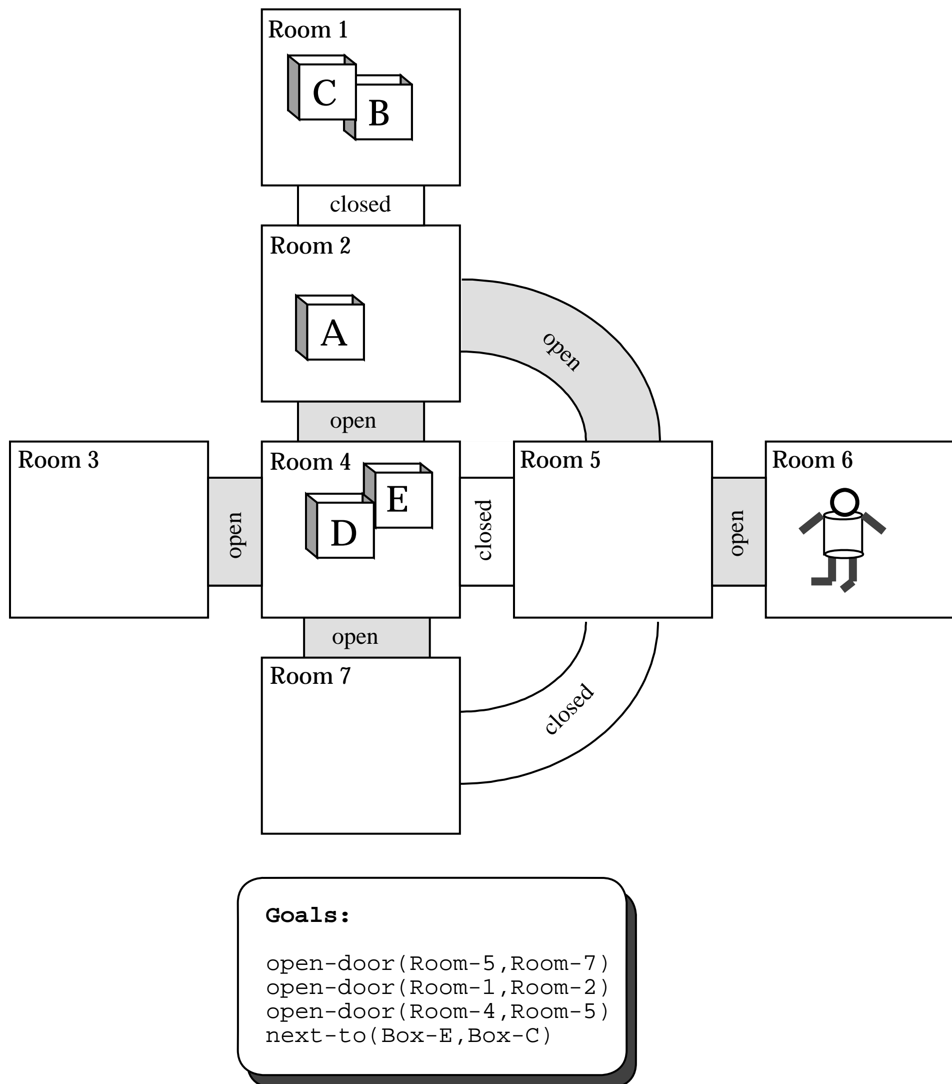


Figure 5.14: Initial state for a 4-goal-conjunct task in a robot domain.

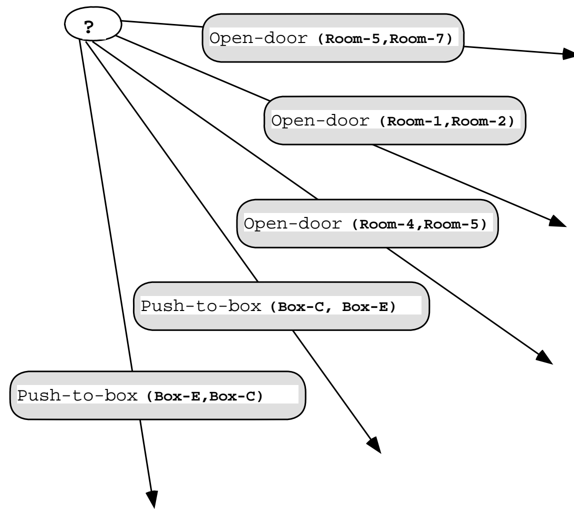


Figure 5.15: Initial operator tie generated for the task of Figure 5.14.

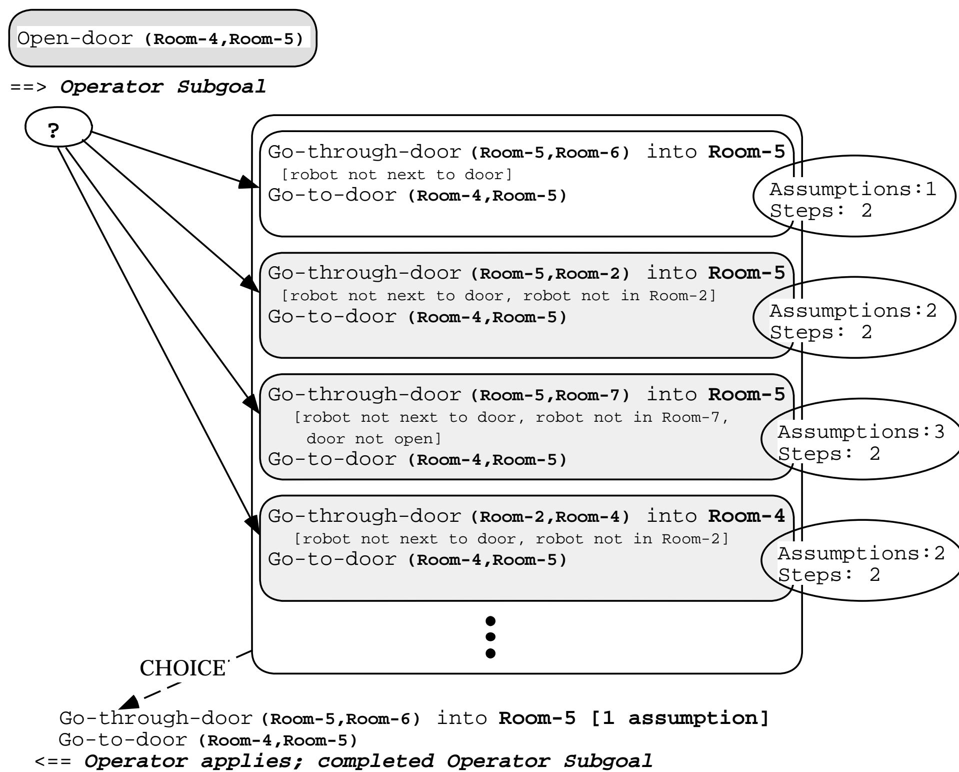


Figure 5.16: Beginning of second-level iteration for the `Open-door(Room-4,Room-5)` operator: achieving the `Open-door` preconditions. The top unshaded box proves to be the best operator sequence.

At the second iteration level, the searches expand to the achievement of one more level of preconditions. Consider the lookahead search which occurs for the `Open-door(Room-4,Room-5)` operator at this level of abstraction — that is, the search to evaluate the plan that is produced by applying the `Open-door(Room-4,Room-5)` operator first. The problem solver must first plan to achieve the preconditions of this initial `Open-door` operator, which are that it be must be in an adjacent room and next to the door. Figure 5.16 shows a portion of the operator tie generated by the possibilities. The robot can approach the door between **Room-4** and **Room-5** from either **Room-4** or **Room-5**. To get into these rooms, it has several choices. E.g., **Room-4** can be approached from **Room-3, Room-2**, or **Room-7**. Each box in the figure shows an operator sequence explored by the system to achieve the preconditions.

When evaluating the choices, the problem solver now abstracts any unmet preconditions of the operators in Figure 5.16. Unmet and abstracted preconditions are shown in brackets. For each sequence of operators, the system discovers that all of the choices take the same number of steps, i.e., have the same domain evaluation. Thus, at this point, iterative abstraction alone would not have allowed the system to make a decision. However, the choice of going from **Room-6** (in the unshaded box) requires the fewest assumptions; therefore it looks the easiest. (And, it is in fact true that it is the easiest). Therefore the problem solver chooses this operator sequence and, still within the context of the abstract lookahead search, applies it abstractly. The room layout of Figure 5.17 shows the result.

Now, still within the lookahead search which began with the `Open-door(Room-4, Room-5)` operator, the problem solver must decide which goal is best to work on next. Figure 5.18 illustrates this process. Since the system is searching at the second iteration level, for each subgoal it explores next it must achieve one level of preconditions before abstracting. (The operator ties generated during these precondition searches are not shown here).

For ease of explication, suppose that the problem solver stops the search when two goal conjuncts have been achieved. Then, based on the evaluations, the problem solver next chooses to apply the `Open-door(Room-5,Room-7)` operator. This operator, in the top unshaded box, has *no* assumptions — the operator was easy enough that no

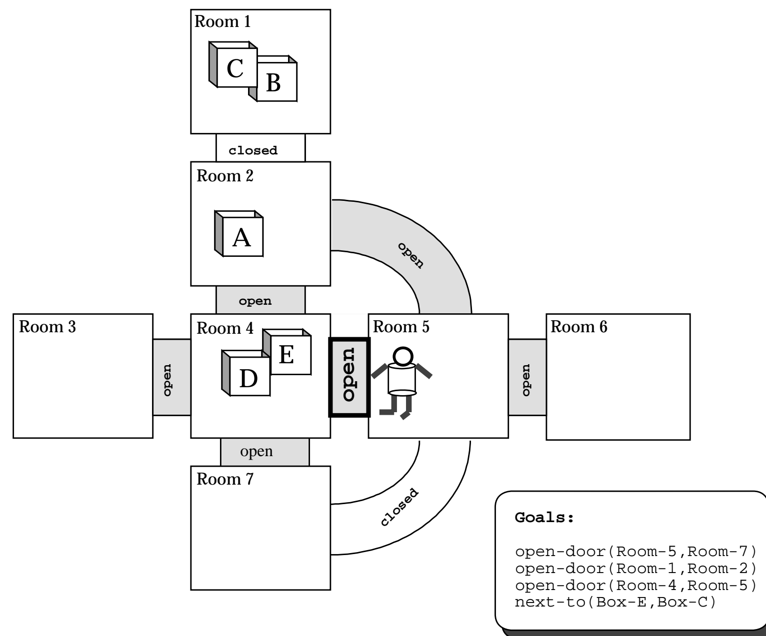


Figure 5.17: Result of applying the `Open-door(Room-4,Room-5)` operator at the second level.

abstraction was used to apply it. This is the case because (in the abstract space) the robot is now in a room adjoining the door. Using abstraction, the problem solver is able to determine this easily without exploring the other choices in detail.

The abstract plan constructed thus far at the second iteration level for the `Open-door(Room-4,Room-5)` operator is shown in the top box of Figure 5.19. It takes five steps, and requires one assumption (shown in square brackets). The abstract plan segment achieves two goal conjuncts.

The abstract plan segments generated for all of the operators in the original tie are shown in Figure 5.19 (again, the ties generated during the precondition searches are not shown here). For each operator in the tie, a search is done to find the best plan segment which begins with that operator (and achieves two goal conjuncts). For example, if the robot first pushes `Box-C` to `Box-E`, then the goal conjunct which appears easiest to achieve next is that which opens door `Room-4/Room-5`.

The top two unshaded abstract plan sequences for the `Open-door ( Room-4,`

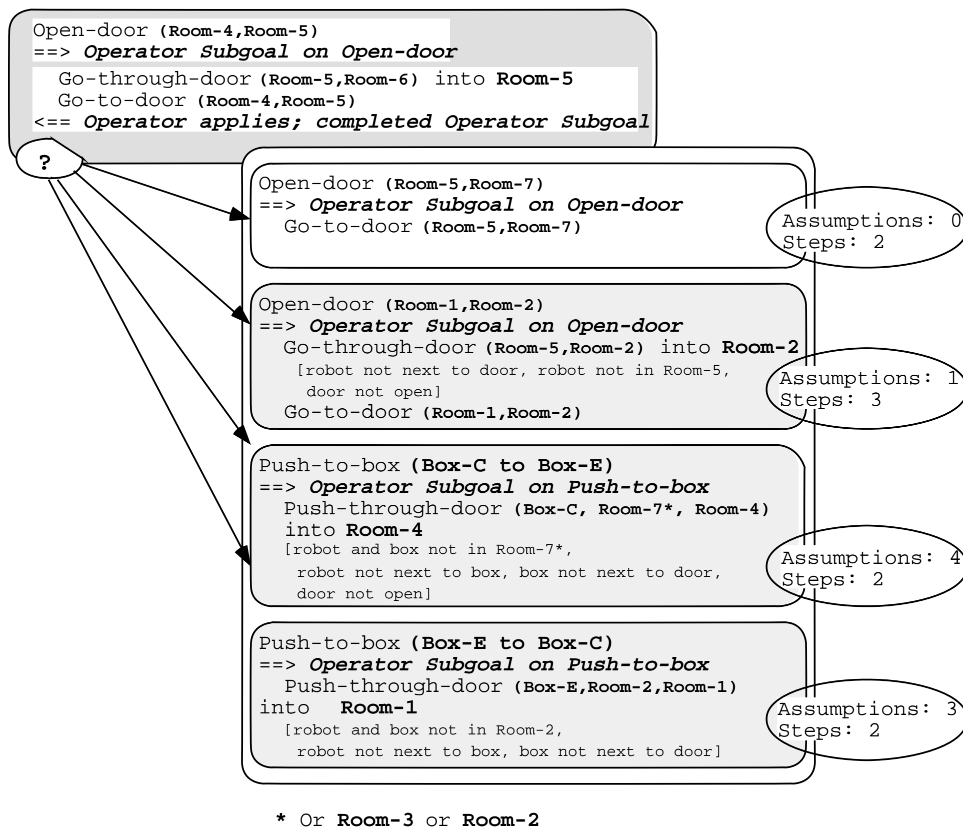


Figure 5.18: Continuance of the second level search starting with the `Open-door(Room-4,Room-5)` operator. Top unshaded box indicates best choice.

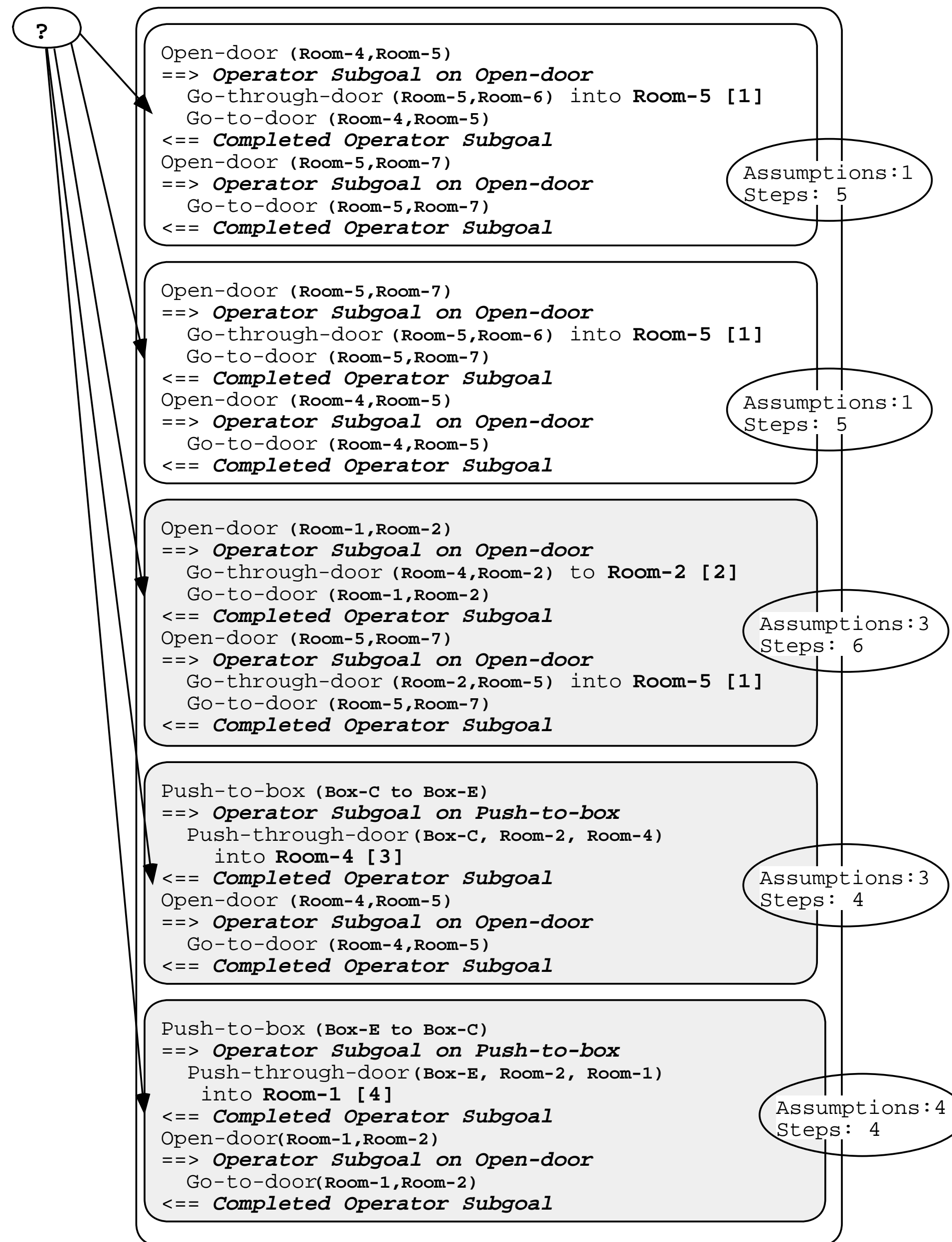


Figure 5.19: Second level abstract plan segments for all operators in initial tie. Top unshaded boxes indicate best choices.

Room-5) and Open-door (Room-5, Room-7) operators have the best evaluations[3], and to resolve the control impasse the problem solver will pick randomly between the two operators (after another iteration).

In this example, the two operators which received the best evaluations are in fact the best operators to apply first. From its abstract plans, the system detects that if it applies these two operators after any of the others, they will not be as easy to do; however, the same is not true of the other operators. The system's information is relative and comes from comparison of its choices.

Note that some of the other options produce abstract plans which have fewer steps. If assumption counting had not been used, then the problem solver would have picked one of these operators instead, which would have generated a longer final ground-level plan. Conversely, if iterative abstraction had not been used, then an essentially random decision would have been made at the most abstract plan level, using assumption counting alone. Hence, the two method increments interact synergistically.

5.5 Problem Representation and SPATULA's Method-Increment Abstractions

Because SPATULA's abstractions are driven by the problem-solving context, its performance is affected by the problem-space representations and search control used by the system for a given domain. When a problem solver uses operator subgoaling to achieve unmet preconditions, its solutions implicitly define a hierarchy of preconditions, with each level of the hierarchy associated with an operator subgoaling level. As discussed in Section 3.7.2, the shape of such a hierarchy is constrained by the operator representations. In addition, the hierarchy is influenced by the system's search control, including the search control knowledge used with respect to precondition achievement (that is, which operators are proposed to achieve which unmet

[3]If the abstract searches had extended to achieve all four goal conjuncts, these two operators would still have been ranked best.

preconditions). This knowledge need not be traditional MEA search control knowledge, with which for each unmet precondition, operators are proposed whose primary effects achieve that precondition. Rather, this knowledge may be of arbitrary form. (For example, the precondition hierarchy would be flatter than with traditional MEA knowledge if produced by a predominately forward-chaining search strategy). This search control knowledge may change dynamically as new rules are learned. Although MEA knowledge was used, e.g., for the robot domain described in this chapter, this need not be the case, and thus SPATULA's abstraction hierarchy need not take on a "MEA-refinement" shape. The operators selected during operator subgoaling determine what is abstracted and how abstract the search becomes.

SPATULA, as a weak method, can be viewed as using the general heuristic that, on average, problem spaces will be designed such that the conceptually more important preconditions, with respect to their role in making useful abstract decisions, will be towards the top of this hierarchy. Clearly, an entire task or even problem space is not likely to be organized in this fashion. However, the greater the extent to which this is the case, the more likely that important problem interactions will be visible at high levels of abstraction, and thus the more useful SPATULA's general approach to abstraction will be.

Assumption counting and iterative abstraction compensate in part for those situations in which the implicit precondition hierarchy is not thus optimally organized. However, they themselves make certain assumptions about the structure of the hierarchy. The assumptions may be reflected in the particular method-increment parameters used. As discussed below, there is latitude for adjustment of these parameters beyond the values used for our experiments; this remains for further work.

With assumption counting, the method increment parameter is the "combination" evaluation function, used to combine the domain evaluation with the assumption count. The combination function may vary in the model used (i.e., the function may be lexicographic, polynomial, etc.), as well as the weight given to each input.

With iterative abstraction, the method increment parameters are:

- The number of iterations, showing a difference between two options, that are required before one option is preferred. (For example, only one iteration, or

several). A related parameter is how much of a difference is required at each of these iterations before a decision is made. (For example, the system may require an *increasing* difference between two options for n iterations before one is preferred).

- the initial iteration level.
- the halting criteria for iteration. (For example, the system may halt and choose randomly if more two or more options looks equally good for n iterations without change. Alternatively, it may halt only if one option appears clearly better, or if search reaches its most detailed level).

5.5.1 Problem Representation and Assumption Counting

Assumption counting allows the system to estimate the degree to which an abstract solution still needs to be filled in. By preferring those solutions which appear to need less refinement, the system is recognizing that unperceived interactions can cause problems. The weight given to the assumption count as compared to the domain evaluation information indicates the degree to which the system believes that the unperceived interactions will cause difficulties.

However, assumption counting itself uses the heuristic that the abstracted preconditions of the options being evaluated are of approximately the same importance with respect to difficulty of achievement and/or impact on other parts of the task (so that it makes sense to compare them). The less the extent to which this is true, the less credence should be given to an exact assumption count. For example, in some domains the method increment parameters might work better if adjusted so that the system only judges one solution better than another if the first solution uses n fewer assumptions.

In the future, a decision-theoretic approach might provide one means of adjusting the assumption-counting method increment parameters. Suppose that the importance (with respect to difficulty and/or task impact) of the unmet preconditions in a domain is observed over time to have a normal distribution around some average. If the variance of the distribution were known, this would have two implications:

1. Given two search paths, each with some number of assumptions, it would be possible to calculate the probability that the assumptions made along one search path represent more important work than the assumptions of the other search path, by at least some number of "importance" units. This estimated difference in importance might then be a more useful contribution to the combination evaluation function than just the assumption counts. The probability of the difference in importance might also provide a more principled means of combining assumption counting information with the domain evaluation.

2. If the potential range of importance of the assumptions remains the same at each iteration level when iterative abstraction is used, and if the number of assumptions tends to increase as the iteration level increases, then the probability that the assumptions along two paths are of the same average importance, would increase with iteration level. Thus, the higher the iteration level, the more certain it would become that a difference in assumption counts actually does indicate a difference in total unmet precondition importance.

5.5.2 Problem Representation and Iterative Abstraction

With the use of iterative abstraction, the system recognizes that the abstract evaluations produced at the highest levels of abstraction may not be an accurate reflection of the abstract plans' utility. Iterative abstraction uses the heuristic that unimportant differences between two search paths will tend to average out until a key difference is found. These unimportant differences may be considered to be "noise", in the sense that they are not salient problem details with respect to picking a good abstraction. The less the extent to which differences average out, the more conservative the iterative abstraction algorithm should be about eliminating options at an iteration. For example, the system may wish to only eliminate a candidate if it remains worse for some number of iterations, or if its evaluation is worse by some significant amount.

Again, a decision-theoretic approach might provide one means for adjusting the parameters used for iterative abstraction. Suppose that for each problem aspect along a search path — where, e.g., a problem aspect is some aspect of the state which must

be changed or assumed done to reach the search goal — the probability that it is "noise" is observed over time to follow a normal distribution. If the variance of this normal distribution was known, then, given two search paths with some number of "problem aspects" each, it would be possible to calculate the probability that the paths were distinguishable (beyond noise) by some amount. Given evaluations for two paths, this information would then provide a means of adjusting the comparisons between evaluations to generate preferences. E.g., if a large amount of noise is suspected, then one search path with a better evaluation might not be given a better preference and a further iteration would be required.

The more problem aspects in the search paths being compared, the greater the probability that the noise for each path approaches the average of the distribution. So, the more detailed the search, the more likely that any perceived distinguishability is not spurious. If the average branching factor of the search trees for a domain were known (so that it was possible to estimate the increase in problem aspects considered at each new precondition level), then it might be possible to calculate a likely level of iteration for which there was a high probability of considering the options distinguishable. The search could then start at this level of iteration, rather than a more abstract level.

For both iterative abstraction and assumption counting, the less the extent to which important problem features are at the top of the implicit precondition hierarchy, the more expensive the abstract searches must become to produce useful results. (Conversely, if the method increment parameters are not set conservatively enough, the abstract decisions will not be as useful under such circumstances.) This does not necessarily mean that SPATULA should not be used under such circumstances. As will be discussed in Section 5.9, under a certain set of assumptions abstraction will remain less expensive regardless of the number of abstraction iterations required, except for very easy problems.

An important area for future research is the exploration of domain-independent ways to determine the most useful settings for the method-increment parameters (and consequently, to determine more accurately when the abstraction method may be too

expensive to be useful). This will be further discussed below in Chapter 9.

5.6 Extended Plan Use Method Increment

The next method increment to be described is called *extended plan use*. The extended plan use method increment addresses the issue of when to use the abstract plans that are learned during lookahead search, rather than what abstractions to make, as was the case with iterative abstraction and assumption counting. There are two parts to the method. The first part provides the system a means to reason explicitly about the extent to which it will use abstract plans learned during lookahead *sub*-searches. To do this, the problem solver must have a means to discriminate between the plans learned at each abstraction iteration, and use only the most detailed sub-search plans available. The second part of the method increment provides a heuristic for guessing which rules from these sub-searches will be most useful, and uses only these more useful plan fragments as the plan is refined.

5.6.1 Extended Plan Use: Part 1

As described previously, search control is learned during the process of resolving control impasses. Such rules are learned from lower-level impasses generated during lookahead search, as well as from the execution-space impasse which originated the lookahead search. The search control rules learned at the end of the iteration process during resolution of the top-level execution-space impasse do not test for abstraction level (since the execution space has no abstraction level), and thus these top-level rules may potentially fire at any time during problem solving. However, each rule learned during a lower-level search will — because of the way that iterative abstraction is performed —contain on its left-hand-side a test for the abstraction level at which it was learned[4]. This means that it will not fire unless the problem-solver is again problem-solving at that same abstraction level. This representation is necessary because the system must not use the lower-level plans learned during one iteration for a

[4]Recall that the default abstract rules are such that if iterative abstraction is not used, the abstraction level is set by default at 1.

subsequent iteration; if it did so, then it would not learn anything new by performing the new iterations.

Once a lookahead search has been completed to resolve an execution-level impasse, we would like to provide the problem solver with the opportunity to use – in the execution space – the most detailed plan that it developed during the lookahead sub-searches. That is, we would like the problem solver to use only the rules learned at the highest available iteration level for that situation. Similarly, if plans learned in different situations simultaneously apply, we would like the problem solver to be able to use only the most detailed. The first part of the extended plan use method increment gives the problem solver these abilities, by allowing the problem solver to explicitly detect and reason about the abstraction level of its plans. Then, once it has determined the most detailed plans available for a given situation, it must then allow these plans to fire in the execution space.

The implementation of these abilities is driven by the way in which Soar reactively matches and uses its plans. (Other systems, which incorporate a declarative access to their own memory, might use different implementations). The default evaluation rules for lower-level impasses are modified such that the actions of the learned search control rules not only add a preference to working memory, but also add a tag stating the iteration level at which the rule was learned. Then, each time a new execution-space operator is selected, the following sequence takes place. Signals are first added to working memory which indicate that rules of *all* abstraction levels are appropriate in the execution space. Each applicable rule from any abstraction level will then match against the signal for the level at which it was created, and will fire and add to working memory both search control information and the tag indicating the rule's abstraction level. These abstraction-level tags are then compared to determine the highest level of detail for all the rules which were applicable in that situation. When the problem solver determines this highest level, it removes the execution-space signals for all other levels. The less detailed rules, without a signal to match against, will be retracted. Only the most detailed search control remains in working memory. This process is repeated each time a new execution-space operator is selected. The rules which implement the first part of the extended-plan-use method increment are shown

To use plans from sub-lookahead-searches within the execution space:

20. Modify the evaluation rules of Figures 5.2 and 5.8 such that the **current-abstract-at-level** (the iteration level) of the goal in which the operators generated the control impasse is added to working memory when the preferences are posted.

21. As each execution-space goal is created (i.e., the initial task goal as well as any operator-subgoals generated in the execution space), add ''iteration flags'' to the goal such that plans from all levels of search can potentially match and fire in the execution space. Add an iteration flag for each potential level of iteration, up to some maximum level (to which it is expected the search will not reach).

22. For execution-space goals, analyze the iteration level information posted by each learned search control rule when it fires, via the modification of Rule 20. Reject all but the highest-iteration-level iteration flags which were added to the execution space goals. (This causes any search control which matched a rejected flag to be retracted.)

23. For execution-space goals, after each operator has applied, replace the ''iteration flags'' for all iteration levels as in Rule 21 (so that the highest iteration level of search control for the next operator can again be reasoned about in the new situation).

Figure 5.20: Default knowledge for the first part of the extended plan use method increment. These rules allow the system to reason about the levels of detail encoded by its learned control rules from lookahead sub-searches.

To prevent just those search control rules which post *indifferent* preferences, and were learned in lower-level lookahead searches, from being used in the execution space:

```
 24. Modify Rule 17 of Figure 5.8 so that it additionally tests
     that it is in a lookahead search context before firing.  (This
     test will be incorporated into the search control learned as a
     results of firing the rule.)
```

Figure 5.21: Default knowledge for the second part of the extended plan use method increment.

in Figure 5.20.

5.6.2 Extended Plan Use: Part 2 (Conservative Version)

The second part of the extended plan use method increment makes more conservative use of which sub-search knowledge is transferred to the execution space. It is motivated by a variant of the same heuristic used to produce the iterative abstraction method increment, but this time cast in terms of what may be learned from a lookahead sub-search. As described in Section 5.3, with iterative abstraction, sub-searches within an iterative search do not iterate recursively — the search stays at the same iteration level all the way through. Thus, options which appear equally good at lower-level control impasses are given *indifferent* preferences, and lower-level plans are learned accordingly. The idea behind this more conservative approach to extended plan use is that these plan fragments, learned when the system was *not* able to discriminate between options during lookahead sub-searches, were not learned at a sufficient level of detail to be useful to execution-space problem solving. Only those lower-level search control rules which were learned when one option appeared clearly better should be transferred. Use of this heuristic provides a more conservative version of extended plan use.

The implementation of this heuristic as part of the extended plan use method increment is very simple, and requires modification only to one of the system's default evaluation rules. The change to this evaluation rule is shown in Figure 5.21; the

rule which creates indifferent preferences for abstract lower-level searches is modified so that it explicitly tests that the search is taking place within a lookahead search context. This test then becomes part of the explanation of the preference, and will be incorporated into the conditions of the learned search control rule. Thus, such rules will only be applicable within a planning context. In contrast, those plan fragments learned when one option appeared clearly better, transfer to the execution space.

5.6.3 Discussion: A Spectrum of Plan Use Methods

The first part of the method increment is effective with or without the iterative abstraction method increment; without iterative abstraction, all plans applicable in the execution space are at the same abstraction level, and will always be utilized. The second part of the extended plan use method increment, which produces a more conservative version of the method, will be null without the first (since if the problem solver does not transfer lower-level abstract plans to the execution space, it does not matter which subset of those plans are applicable outside the planning context).

The extended plan use method increment provides the problem solver with ability to discriminate between lookahead-sub-search plans at different levels of abstraction, and to reason about their use. However, use of these lower-level plans may not always be beneficial. If the system does not use the extended plan use method increment, then it will effectively be using only the first step of an abstract plan developed to resolve a top-level control decision impasse (since the sub-search plans will not transfer to the execution space). Without information about the later steps in the plan, control impasses will be re-generated as the abstract plan is refined, and these control impasses will need to be resolved again. However, this time the problem solver will be making the decisions with any new information obtained from the plan refinement that has occurred in the interim. This means that its re-resolution of these impasses, though requiring additional search time, may be more accurate. Similarly, if the problem solver uses the more conservative version of the extended plan use method increment, it must re-resolve those lower-level impasses in which two options looked equally good — again, the re-resolution of these impasses will be at greater cost, but increased accuracy.

Thus, the extended plan use method increment gives the problem solver the leverage to make a decision about how conservative it will be in the use of its lower-level abstract plans. In a domain in which mistakes are relatively costly to patch and/or cause poorer solutions, operation without the extended plan use method increment can provide the best cost/benefit tradeoff [Korf, 1990]. Conversely, in a domain in which mistakes are easy to patch and do not greatly impact solution quality, a less conservative approach to plan use is probably more appropriate. We have not yet automated the problem solver's selection of the appropriate plan use mode, but in Chapter 6, we will gives examples of domains for which different plans use modes proved to be most effective.

5.7 Efficiency-Driven Method Increments

The last two method increments provided by SPATULA reduce the computational cost of the problem solver's abstract searches, while at the same time employing heuristics to allow the most relevant information from the searches to be maintained. They are called the *abstraction-gradient* and *goal-achievement iteration* method increments. The two method increments both use *greedy*, or localized, evaluation approaches — they examine most closely the more immediate effects caused by the operator being evaluated. They share in common the idea that it is possible to estimate the effects of applying a particular operator (and thus to make a control decision) without determining the effects of that operator on the entire task, and that this can be done in such a way as to keep the cost of successive abstraction iterations from escalating as sharply. Thus, the more highly interdependent are the goals in a task, the worse may be the performance of these method increments. A rationale behind both method increments is that if there is uncertainty in the environment, it will become worse the further into the future the plan is projected.

The two method increments are orthogonal to each other and may be used in conjunction with any of the other method increments presented above.

5.7.1 The Abstraction-Gradient Method Increment

The abstraction-gradient method increment causes an abstract search to become more and more abstract the longer the search continues, until some most abstract level is reached. As with the method increments described above, this method can be considered both in general terms and as implemented in a particular way by the research described here. In general, the idea is that the most detailed problem solving during an abstract search will occur at the beginning of the search, and that problem solving then becomes more abstract (and hence involves less work) the longer the search continues. Using abstraction-gradient, the system pays the most attention to those aspects of the task which are the most direct consequences of the operator being evaluated; thus, the increment takes a greedy approach to search.

For SPATULA, the way in which we have operationalized the idea of performing increasingly abstract problem solving within a search, is to use the concept of levels of abstraction described in the iterative abstraction method above, and to apply it in reverse to an *individual* abstract search, where the search may or may not be part of an iteration series. Specifically, the problem solver begins a lookahead search at some given level of abstraction. However, after each operator application within the initial (highest) goal of the lookahead search, the method increment directs the problem solver to *reduce* the level of detail by decrementing the number of precondition levels which must be achieved before abstraction can occur. When the number of precondition levels reaches one (that is, when the operators in the top goal of the abstract search have their preconditions ignored), the problem solver remains at this level of abstraction for the remainder of the search.

If the abstraction-gradient method increment is used with iterative abstraction, then with each iteration, an increasingly larger amount of problem-solving will take place before the lookahead search reaches its most abstract. Figure 5.22 shows the abstraction-gradient method increment applied to a lookahead search for three subsequent iterations within a control impasse. In the figure, triangles indicate search to achieve an operator; the bigger the triangle, the more detailed the search. At the first iteration, search is initiated at the system's most abstract level and so does not

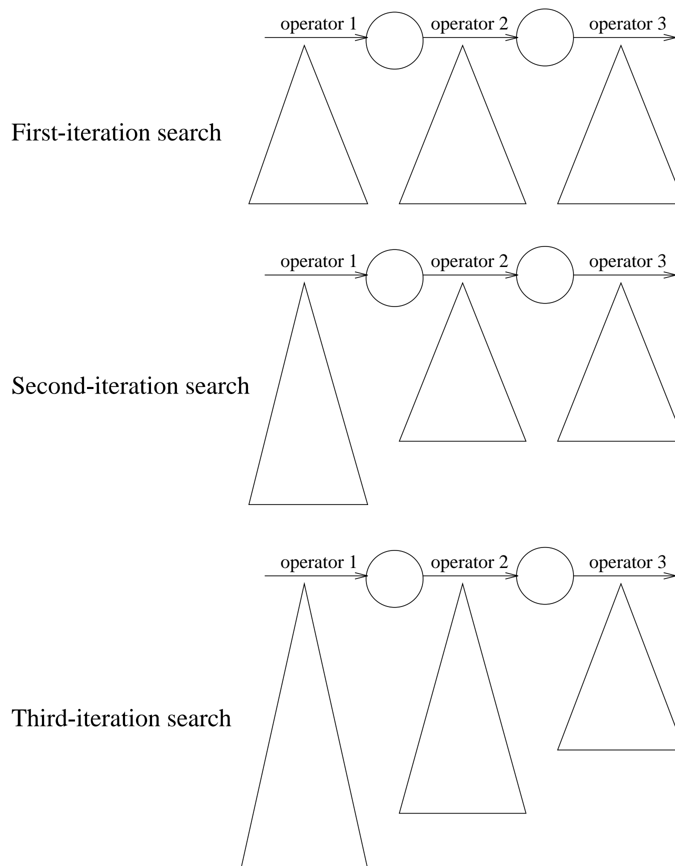


Figure 5.22: An illustration of the abstraction-gradient method increment. The initial abstraction level of a search determines how much more abstract it becomes as search becomes deeper.

> 25. `Within a top-level lookahead search subgoal (for each option in a top-level control impasse), each time an operator is applied, and the` **current-abstract-at-level** `variable for this top search subgoal is > 1, then decrement the` **current-abstract-at-level** `variable.`

Figure 5.23: Default knowledge for *abstraction-gradient* method increment.

become more abstract. At the second iteration, which begins in more detail, the system will reduce the detail of the search after the first operator is applied. However, the search is then at its most abstract and is not abstracted further after the second operator has applied. At the third iteration, the search begins at yet more detail, and becomes progressively more abstract the deeper the search.

Figure 5.23 summarizes the single rule from Appendix A used by the system to implement the abstraction-gradient method increment. It tests and modifies the same variable — `abstract-at-level` — that the system compares against the current subgoal level of search to determine if it should currently be abstracting, and which the iterative-abstraction method increment increases at each iteration. The abstraction-gradient method may be used with or without iterative abstraction, although if lookahead search is initiated at its most abstract level without iterative abstraction, abstraction-gradient will have no effect.

As an example of the abstraction-gradient method increment, Figure 5.24 revisits the searches of Figure 5.19, and shows the evaluations which would be produced for the operators at the second iteration level if this method increment was employed as well. For each search path, the problem solver begins by solving for one level of preconditions. However, as a goal conjunct is achieved, the level is decremented for that search. So, the remainders of the searches are performed with all operator preconditions abstracted.

As may be seen from the evaluations, by using the abstraction-gradient method increment, the problem solver is still able to estimate that it is easier to open one of the doors closer to the robot's initial position first, rather than achieve another subgoal and then return. It is able to make this distinction with less effort than

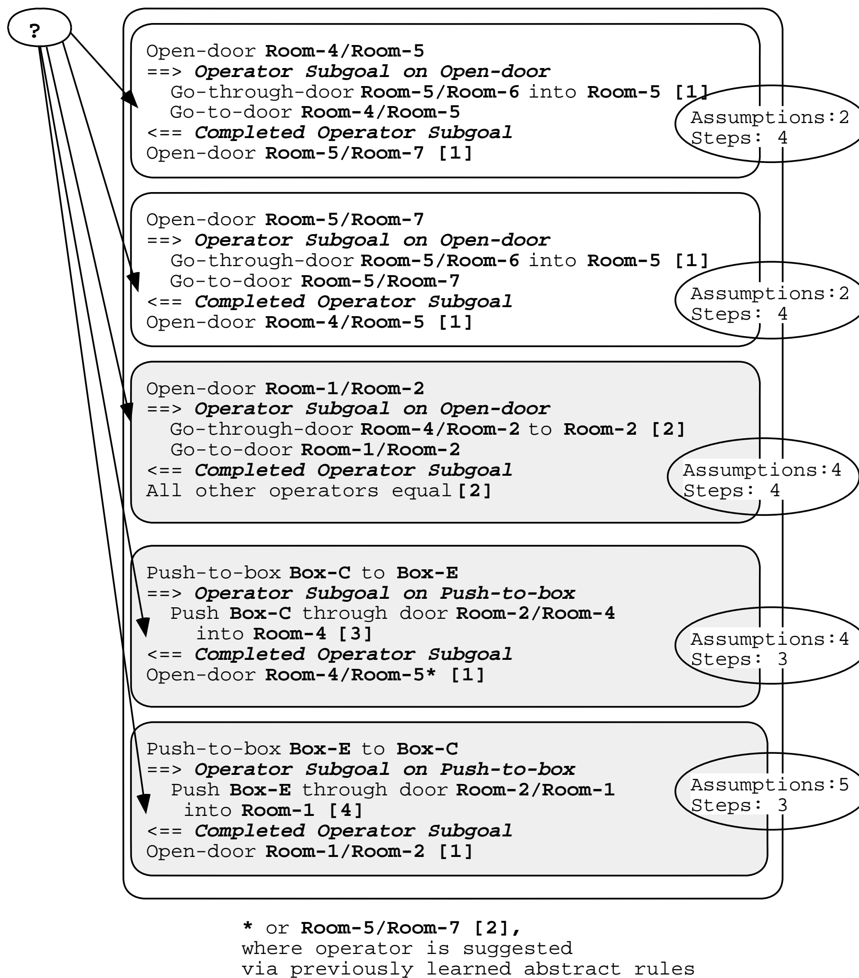


Figure 5.24: The searches of Figure 5.19, with the abstraction-gradient method increment used as well. The same evaluation is obtained with less effort.

required previously, by looking at the beginning of its search paths in the most detail. In this example, a greedy approach works well.

5.7.2 Iteration on Goal Conjunct Achievement

The second efficiency-driven method increment is called *goal-achievement iteration*. It also expresses, in a different way, the idea that it may be useful to reduce the effort required to perform an abstract search by using increasingly abstract problem-solving the longer the search proceeds.

Goal-achievement iteration is motivated by the observation that if some goal conjuncts in a situation may be achieved independently of each other, then the problem solver can work locally on the best ordering of the operators which achieve the dependent goal conjuncts, without needing to take the full task into consideration. This search control technique — that of only searching to achieve a subset of the task goals — may be considered a form of abstraction, since with its use, part of the task is abstracted away during search. E.g., [Lansky, 1992] provides a discussion of such a technique, called *localization*, and discusses its relationship to abstraction. However, the automatic determination of subgoal dependencies requires an assumption about the availability of declarative information for problem analysis. As an alternative approach, if the system does not have information about *which* goal conjuncts provide operator interactions, but were to know the size of the largest subset of interacting goal conjuncts (call this size k), then it could use this knowledge to reduce its evaluation effort at a control decision, by only searching to achieve k goal conjuncts together along each search path.

For example, consider a robot domain task whose goal conjuncts include moving a box to a room, and closing the only door to that same room. Suppose that for this task, the problem solver is given the information that the achievement of each goal conjunct can affect at most the achievement of one other conjunct, and uses this information to search to achieve only two goal conjuncts during its planning phases. At a control impasse which included the relevant `push-box` and `close-door` operators, the system would evaluate the results of pushing the box and then achieving another conjunct, as well as the results of closing the door and then achieving another

conjunct. Without searching to achieve the complete task, it would learn that pushing the box and then closing the door was more useful than closing the door first (since the door would only need to be opened again when pushing the box, thus undoing the goal conjunct achievement). A partial goal conjunct ordering would thus be developed. Using this approach, as problem-solving proceeds and more goal conjuncts are achieved, evaluation searches to k additional conjuncts will reach further towards the task goal state. In this way, an *incremental* refinement of the goal ordering for the task is generated. However, although this approach is effective if the problem solver is given a good estimate of k, it will not be provided with such information in general.

The goal-achievement-iteration method increment addresses this difficulty, by modifying the iterative-abstraction method increment. It reduces the effort spent during iterative-abstraction search, by having the problem solver look for interactions between many goal conjuncts at less detail than it uses to look for interactions between fewer goal conjuncts. The system does this within the context of iterative abstraction by first looking for any interactions between large sets of goal conjuncts at a high level of abstraction (and thus more cheaply)[5]. The highly abstract searches may indicate with relatively little effort that some of the options at a control decision impasse will lead to undesirable interactions. Such options will be eliminated from consideration. As search detail is increased using iterative abstraction, the remaining options will be evaluated at greater detail. At the same time, using goal-achievement iteration, the problem solver will decrease the size of the subsets of goal interactions that it considers, by reducing the number of goal conjuncts which must be achieved during search. At each iteration, more detailed interactions may be detected, but fewer potential interactions are considered. Therefore, goal-achievement iteration will be most appropriate for tasks in which goal conjunct interactions are either relatively localized, or easy to detect at a high level of abstraction. As with the abstraction-gradient method increment, goal-achievement iteration takes a greedy approach to search.

[5]The set of goal conjuncts under consideration need not necessarily consist of all the goals of a task; this method may be applied to a currently "in focus" subset of the task goals.

With goal-achievement iteration, a search within a control impasse subgoal may result in a partial rather than total ordering on task goal conjuncts. As task problem-solving proceeds and goal conjuncts are successively achieved, fewer additional goal conjuncts need to be achieved to reach the goal. Thus, at each new top-level control decision impasse, this partial ordering will be incrementally refined.

Note that goal-achievement iteration is not entirely independent of the domain evaluation used for the task, because it assumes that it is possible to perform an evaluation given that only some subset of the goal conjuncts have been achieved. For example, the method increment may be used with a domain evaluation based on a metric such as solution length or cumulative operator cost.

5.7.2.1 Implementation

Goal-achievement iteration is implemented in SPATULA by building on iterative abstraction as follows. At each iterative-abstraction iteration, the problem solver *decrements* a variable which holds the number of goal conjuncts to achieve at each iteration, at the same time that the abstraction level is *increased* using iterative abstraction. So at the first iteration for a control impasse, the abstract searches are performed very abstractly, but all required goal conjuncts are taken into consideration. If the problem solver is not able to resolve the impasse, then another abstraction iteration is performed, in which the searches are made in more detail but the number of goal conjuncts to solve for is decremented. The iteration process continues in this manner. Variations on this basic procedure are topics for future work. E.g., it may be useful for the results of the option evaluations at each iteration to influence whether the goal conjunct count is in fact decremented or remains the same.

Figure 5.25 summarizes the knowledge in Appendix A used by the system to produce the goal achievement iteration method increment. The current implementation of the method increment assumes that the system has knowledge of the total number of goal conjuncts for which it would normally solve when performing an evaluation for the operator in question. However, if this is not the case, then this information could be derived during the first iteration of problem solving, by noting the number of conjuncts achieved when the system's domain knowledge detects that an evaluation

26. When a top-level control-impasse subgoal is initiated, initialize **number-of-goals-to-achieve**. The initial value may be the total number of goal conjuncts in the task, or less.

27. When a top-level lookahead search subgoal is initiated, initialize **goal-conjuncts-achieved** to 0.

28. As each new lower-level subgoal within a lookahead search is initiated, copy down the value of **goal-conjuncts-achieved** from parent goal.

29. Note when a goal conjunct has been achieved; increment **goal-conjuncts-achieved**.

30. Note when a goal conjunct has been un-achieved; decrement **goal-conjuncts-achieved**.

31. For each iteration within a control impasse, decrement the **number-of-goals-to-achieve**. (This rule will fire synchronously with the iterative abstraction rule which increments the **abstract-at-level** variable for the control-impasse subgoal).

32. When **goal-conjuncts-achieved** $\geq$ **number-of-goals-to-achieve**, add a flag which signals that search success is detected. (This then indicates to other default rules that an evaluation of the search may then be performed).

Figure 5.25: Default knowledge for the *goal-achievement-iteration* method increment.

should be performed. As discussed above, the method increment is not appropriate if evaluations can not still be performed when some smaller number of goals conjuncts than this total are achieved. If used without iterative abstraction, this increment is a null method unless the initial **number-of goals-to-achieve** is set to a number different than the total number of goal conjuncts.

5.7.2.2 Example

The example presented in Section 5.4 illustrates the way in which a useful partial ordering for task subgoals can be constructed by using goal-achievement iteration in conjunction with iterative abstraction. The example illustrated search at the second iteration level. Recall that the system was able to choose to open the two doors nearest to the robot first, while only searching to achieve two goal conjuncts.

Goal-achievement iteration will produce good results for this task because it is not necessary to look at all sequences of *all* goal conjuncts to make a decision about the system's first move. By a relative comparison of the results of achieving all combinations of any two goal conjuncts, the problem solver was able to determine that it appeared easier to open the two closest doors first, rather than to achieve any of the other goal conjuncts first and then open one of the closest doors[6].

In this example, once the initial two `Open-door` operators have applied in the execution space, the problem solver will require additional search to determine which of the remaining two goal conjuncts (`Next-to(Box-E,Box-C)` and `Open-door(Room-1, Room-2)`) to work on next. If it is using goal-achievement iteration, then, as before, it will start its search by achieving a relatively large number of goal conjuncts at a high level of abstraction, and decrease the number of goal conjuncts with each iteration. However, now that there are only two goals left to achieve, the system will be able to search to task completion at the second abstraction iteration. In this example, the `Push(Box-E,Box-C)` operator will be applied next. This operator is chosen because the system detects that after the application of the first two operators, the robot is

[6]As described above, goal-achievement iteration would in fact have solved for three rather than two goal conjuncts at the second abstraction iteration, given that the system initially began by achieving all four task goals at its first iteration. Thus, for this example, even weaker conditions than the default are effective.

now near **Box-E**, and that once it has pushed **Box-E** into **Room-1**, it will be near the door between **Room-1** and **Room-2**. (It does not, however, detect that the door will be fortuitously opened in the process of pushing **Box-E**).

Thus, for this example, an initial partial ordering of the goals was useful, but a total ordering was not necessary — the remaining goals were ordered later in the task. In this way, goal-achievement iteration produces an incremental refinement of the partial ordering of the goal conjuncts.

5.7.2.3 Factorization of Goal Conjunct Tests

To be effective, the goal-achievement iteration method requires that — as suggested in Chapter 3 — tests for goal conjunct achievement be factored. In the same way that precondition tests must be factored to allow assumptions to be counted, goal tests must be factored to allow the problem solver to count the number of conjuncts achieved.

If the goal test is not factored, the problem solver will not be able to detect that individual goal conjuncts have been achieved. The evaluation search will terminate when the complete goal is reached (if no other terminating conditions are reached first). In such a case, goal achievement iteration will be a null method, and have no additional effect on the iterative abstraction method.

5.7.3 Discussion

The abstraction-gradient and goal-achievement-iteration method increments reduce abstract planning expense by expending less planning effort the deeper the search becomes. These method increments proved useful in most of our experimental domains, as will be described in Chapter 6. Their utility is related to the extent to which the task goal conjuncts are interdependent, and thus we do not expect them to be equally useful in every domain. Further research is required to characterize the cost-benefit tradeoffs of these method increments with respect to the potential goal interdependencies in a given domain.

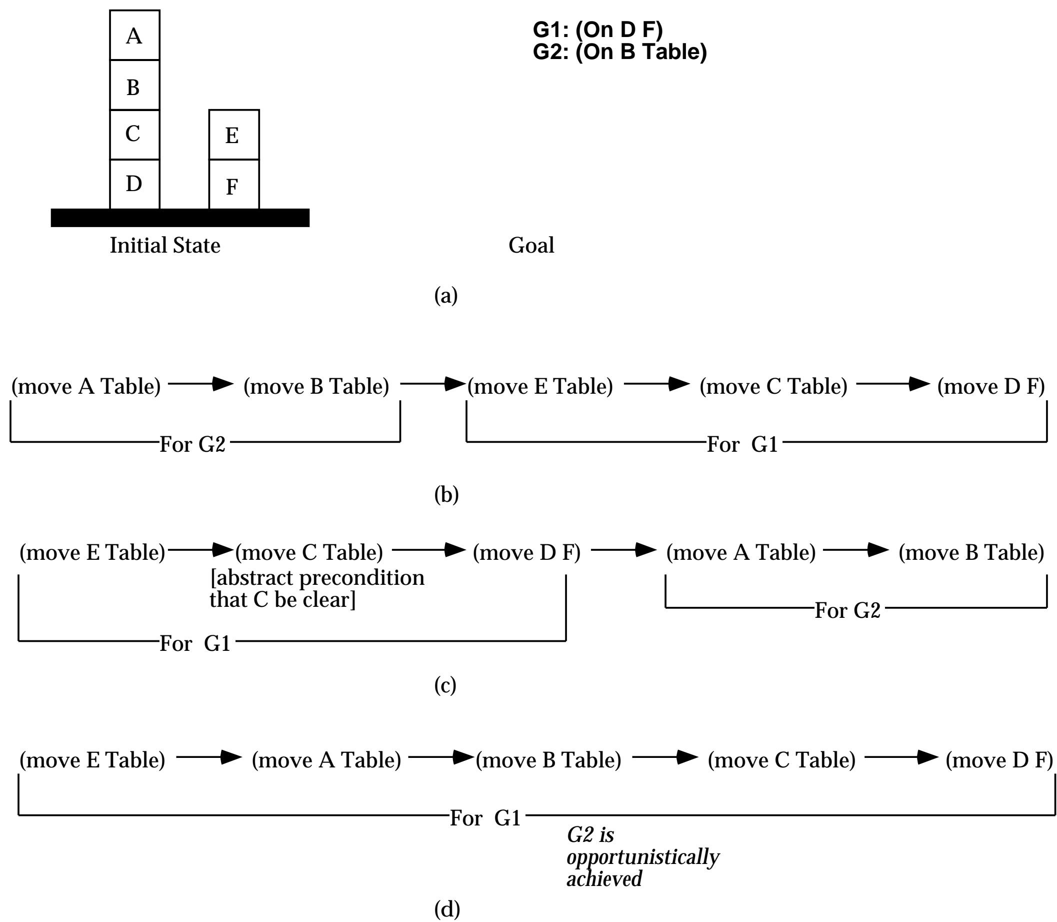


Figure 5.26: Example of bias in abstraction.

5.8 Discussion: Method Increment Biases

Use of SPATULA's method increments provides several *biases* to the solutions produced by the system when using abstraction, where a bias is defined as any basis for choosing one plan over another other than the task goal.

One notable bias is produced by the conjunction of iterative abstraction and assumption counting; if there are helpful interactions between subgoals, the system will tend to prefer to work on the easiest subgoals first. If the interactions are detrimental, it will tend to do the easiest subgoals last. The reason is that during lookahead search, the system is able to see the effects of easier subgoals more completely.

Consider the simple "Blocks World" example of Figure 5.26. Assume that the system is searching using a linear goal achievement strategy, to determine which of its two goals to achieve first. Assume also that the `move` operator in the example has the preconditions that both the block being moved and the target location are *clear*. The domain evaluation is solution length, and the system is using both iterative abstraction and assumption counting.

For this task, the abstract lookahead searches at the first, most abstract iteration level will not discriminate between the two goal orderings, and thus the system will iterate. Figures 5.26(b) and 5.26(c) show the abstract lookahead searches from the initial state to the goal, at the second iteration level. Figure 5.26(b) shows the lookahead search which takes place when Goal G2 is attempted first. It turns out that no abstraction is actually necessary during this search at the second iteration level, since the preconditions of `(move A Table)`, `(move E Table)`, and `(move C Table)` are all met. Next, 5.26(c) shows the lookahead search which takes place when Goal G1 is attempted first. Here, the unmet precondition of the `(move C Table)` operator is abstracted. Since the abstract operator application does not alter the state information that A is on top of B, A must still be moved to the table before B may be moved.

SPATULA considers search (c) more expensive than search (b), since it makes more assumptions. (Search (b) makes no assumptions). Therefore, the plan described by search (b) would be selected. This plan needs no further refinement.

If the plan described by search (c) *had* been selected, it would have been refined to the sequence shown in (d). Without abstraction, this solution would have appeared as good as (b). However, with the abstraction method increments, the system is biased towards the solution which works on the easier subgoal first; the system can ascertain that achievement of G2 does part of the work for G1, but at the second abstraction iteration level, it can not tell that the reverse is true as well.

Other primary method increment problem-solving biases are produced by the conditions which test whether an option should be eliminated from the iteration process, and whether all remaining options should be considered equally good (that is, when to halt). These decisions are affected by both the iterative abstraction

parameters and the assumption counting meta-evaluation function. These conditions bias the problem solver because it is making a guess that an option which does not look promising at a high level of abstraction will not look any better during later iterations and thus does not need to be considered further. These early guesses may then impact the choices made during later iterations of the same impasse, or during later impasses.

The other method increments bias the system as well. Abstract plans can transfer overgenerally and impact the space of possible solutions; thus, the extent to which the plans are used provide a bias. Clearly, the efficiency-driven method increments bias the problem solver, since they force the system to attend most closely to the more localized effects of an operator.

Therefore, changes in the method increment parameters change the system's bias, and consequently impact the space of inductively generalized search control *concepts* which may be learned during problem-solving.

5.9 Search Complexity using SPATULA

In this section, the complexity of search using SPATULA is analyzed with respect to a linear goal achievement strategy, where "linear" goal achievement is defined as that in which work on different subgoals is not interleaved. However, search may still include consideration of the different ways in which to achieve a subgoal — with respect to its impact on the achievement of other subgoals — as well as the order in which the subgoals are achieved. This strategy is relevant since it is the default control strategy used for our experimental domains[7].

The analysis makes several simplifying assumptions. It assumes that learning enables the problem-solver to remember the plans it has constructed, but, for purposes of simplification, that no transfer to new situations occurs. (As observed from empirical tests to be described in Chapter 6, in reality some transfer would be likely.) The analysis assumes no use of the efficiency-driven method increments of Section 5.7, but

[7] As will be discussed in Chapter 6, use of linear goal achievement as the default strategy did not mean that the problem solver was restricted to the *linearity assumption*; rather, if its linear strategy failed, other operator orderings could be utilized.

does assume the use of iterative abstraction. The analysis does not take into account any plan repair activity. In reality, with SPATULA, the utility of its abstractions varies with the domain and domain representation, and repair is necessary at times.

5.9.1 Search to apply an operator

First, consider the average cost of a search to apply one top-level operator (that is, an operator proposed to help achieve a task goal) non-abstractly. An operator with unmet preconditions will have some number of different orders in which it will try to achieve its preconditions (with respect to domain search control), and some number of different sub-operators with which it will try to achieve a precondition (again, with respect to domain search control). Call the average cost of enumerating the sub-operators in all the different potential paths for achieving the preconditions, B. That is, the search tree of possible sub-operator sequences to achieve an operator's preconditions has an average of B nodes. Because we are assuming here that goal achievement is linear, each of these B sub-operators will also require cost B to order the operators achieving their preconditions, and so on. Let d be the average number of operator-subgoaling levels required to refine a top-level operator. Then, the average non-abstract search expense for such an operator will be $O(B^d)$.

Now, consider the cost of an abstract search to apply a top-level operator, again using linear goal achievement. Call the average number of sub-operators required to achieve an operator's preconditions, l. So, B must always be $> l$. (If $B = l$, then there is no search necessary to apply the operator and abstraction would not be used). Suppose the system is performing its abstract searches at the kth abstraction level (that is, it searches to achieve k levels of preconditions). Given a top-level operator, the system will first perform an abstract search of cost $O(B^k)$. This will produce an abstract solution of length $O(l^k)$, in which a plan is produced for the achievement of k levels of preconditions.

Then, each of the $O(l^k)$ operators in the abstract solution must be refined. If the system were to refine these sub-operators non-abstractly, the cost of this search would be $O(B^{d-k}l^k)$. However, when k^{th}-level abstract search is used to refine each of the operators, an abstract plan will be produced of length l^{2k}, at cost $O(B^k l^k)$. The

cost of refining each of the l^{2k} operators may again be reduced using abstraction, and so on. With multi-level abstraction — that is, with abstraction used for each new sub-refinement — the total refinement cost with abstract search is

$$O(B^k(1 + l^k + l^{2k} + ... + l^{d-k})) = O(B^k l^{d-k})$$

when d divides k evenly. (When d does not divide k evenly, the cost is $O(B^k l^{k((d \text{ div } k)-1)} + B^{d \bmod k} l^{d-(d \bmod k)}))$.

With iterative abstraction, an abstract search starts by abstracting all preconditions of the operators in its search, and increases the levels of preconditions solved for until a choice is made. If for each refinement a decision is made at the kth level of precondition achievement, then the cost for multi-level abstraction over the d operator-subgoaling levels is is

$$O((B + B^2 + ... + B^k)(1 + l^k + l^{2k} + ... + l^{d-k}))$$

when d divides k evenly. As with iterative deepening [Korf, 1985a], the dominant complexity of the expression comes from the last iteration. Thus, the complexity of using iterative abstraction to apply an operator remains $O(B^k l^{d-k})$, when k is the number of iterations required for each refinement. (Analogously, the complexity remains the same for the case in which d is not an even multiple of k.)

5.9.2 N goal conjuncts

Now consider the work needed to solve a task with n goal conjuncts, again with linear goal achievement. With SPATULA, as with any relaxed-model abstraction method, the number of task goal conjuncts is not reduced. Thus, the problem-solver must search abstractly to order the operators which achieve its task goal conjuncts.

The search to find the best ordering for a group of n objects is of complexity $n!$, which approaches $O(n^n)$. If for each of n task goal conjuncts, there are c possible top-level operators which can achieve that conjunct, then the complexity of search to order a task's top-level operators can approach $O((cn)^n)$. (This is a worst-case estimate, since other sources of search control may obviate the need to search for all

orderings of all goal conjuncts). The cost of non-abstractly solving a task with n goal conjuncts is then $O((cn)^n B^d)$.

As above, suppose that using abstract search, k levels of precondition achievement are required to discriminate among options and make a decision at a choice point. Then, the abstract search to order the operators that achieve n task goal conjuncts will be of cost $O((cn)^n B^k)$. This search will produce an abstract solution of length nl^k. The remainder of the task plan refinement with abstraction will be of cost $O(nB^k l^{d-k})$, since the task goal conjunct order is now fixed, and there are nl^k operators with refinement cost $O(B^k l^{d-2k})$ each. Thus, the total task cost with abstraction is

$$O((cn)^n B^k + nB^k l^{d-k}).$$

(For simplicity we assume that one top-level operator is required to achieve each task conjunct; if a operators are required to achieve each conjunct, then the second term in the sum is multiplied by a.) The complexity of the first term in the sum is exponential in n, but the second is not. This is because task goals can vary in number, but we are assuming that the average cost of applying a top-level operator is independent of this number, and depends upon the domain representation and search control.

The ratio of task expense with abstraction to without is

$$O\left(\frac{1}{B^{d-k}} + \frac{l^{d-k}}{(cn)^{n-1} B^{d-k}}\right).$$

Several results may be observed from this expression.

- Abstraction is theoretically cheapest when $k = 1$, assuming that no backtracking or patching need occur. (In reality, the efficiency of fewer iterations must be balanced against the utility of the abstractions produced.)

- The complexity of search using SPATULA can not be less than the expense required to order the top-level goals. This expense will depend upon the domain search control, but may be exponential in the number of goals.

- For small n, the second term in the ratio is dominant, and the relative savings from abstraction depends upon the relative costs of B and l, and will increase

as n increases. However, as $(cn)^{n-1}$ becomes larger than l^{d-k}, the first term becomes dominant and the ratio approaches $O(B^{d-k})$. Thus, the *relative* savings from SPATULA using linear goal achievement is bounded by domain characteristics rather than task size. (Though not discussed here, this bound does not exist when achievement of subgoals is interleaved. In that case, though the expense of both abstract and non-abstract search increases, relative savings provided by abstraction increases with the number of task goal conjuncts).

- For a given n, the relative savings increases as B or d increases. Thus, an increase in domain complexity provides an increase in the relative gains provided by SPATULA.

5.10 Summary

The basic abstraction method described in Chapters 3 and 4, when used alone, will not always provide new information to the problem solver during search if strong sources of other search control knowledge about operator effects are already available. In this chapter, building on the framework of the basic abstraction technique, a repertoire of abstraction method increments was added to SPATULA to strengthen the method. The method increments provide heuristics for generating useful abstractions, while avoiding any assumptions about the extent of the declarative knowledge available to the problem solver about its domain. Figure 5.27 summarizes the abstraction method increments described in this chapter.

In the chapters thus far, we have described an abstraction method which addresses our research goals: it is domain-independent, allows the problem solver to make decisions more efficiently, learn about its domain more easily, and increase transfer of its plans to new situations. In the next chapter, we will present the results of empirically evaluating the utility of SPATULA's basic abstraction technique and associated method increments in several test domains.

- Method increments which allow the problem solver to obtain additional contextual information with which to estimate which abstractions will prove most useful:
  - Assumption counting: The number of assumptions required to complete an abstract search path is taken into account when estimating the relative merit of different control decision options.
  - Iterative abstraction: Makes use of the heuristic that a useful level of abstraction for a situation is that at which the system can discriminate among the situation's options. Iteratively increases search detail at a control decision until such a discrimination can be made.
- Method increment which allows the problem solver to reason about its abstract plans:
  - Extended plan use: allows the problem solver to control the circumstances under which its different levels of abstract plans are learned and used.
- Method increments to increase the efficiency of the abstract search, while providing heuristics which allow the problem solver to focus on important aspects of its task. These method increments tend to be less effective in domains for which many task goal conjuncts are highly interactive:
  - Abstraction-gradient: Abstraction level increases as an evaluation search progresses, thus focusing more attention on actions taken at the beginning of the search.
  - Goal-achievement iteration: As search detail increases, search is restricted to exploring interactions between smaller sets of task goal conjuncts. A null method without the concurrent use of iterative abstraction.

Figure 5.27: Summary of SPATULA's method increments.

Chapter 6

Basic Experimental Results

6.1 Introduction

The general weak abstraction methods which comprise SPATULA have been implemented in the Soar problem solver, and experimentally tested in three distinct domains: a Robot Domain, the Tower of Hanoi, and the Eight-Puzzle. In addition, some experiments were run using a less general method for abstracting operator implementations (rather than preconditions) in a fourth domain, computer configuration. In this chapter and Chapter 7, the results obtained with the SPATULA abstraction method are presented. The computer-configuration results are discussed in [Unruh and Rosenbloom, 1989].

Our experiments were performed to determine the extent to which the use of SPATULA's abstraction methods:

- produces good task solutions;
- increases problem-solving efficiency;
- allows the problem solver to learn new rules about its domains more easily;
- and increases transfer of the rules learned.

In this chapter, we focus on the results pertinent to the issues above. We will show that in our experimental domains:

- using SPATULA, solution quality is significantly better than that produced by choosing a solution based on pre-existing domain search control.
- using SPATULA, abstract search requires fewer problem-solving steps than the corresponding non-abstract search.
- the harder the domain, the larger the payoff — in efficiency and solution quality — to be obtained from tractably constructing a good plan with which to guide the construction of a solution. SPATULA provides a means for constructing such plans.
- using SPATULA, the cost of building learned rules is reduced. The more complex the non-abstract subgoal search, the greater the cost reduction.
- except in domains for which tasks are very similar to each other, learned abstract plans allow significantly better plan transfer than non-abstract plans.

In the remainder of this chapter, we first describe the experimental domains and methodology used for the experiments. Then, the results introduced above are described in the following manner. First, the impact of SPATULA's basic abstraction method alone is discussed. The next two sections focus on the results of using iterative abstraction and assumption counting. Then, abstract plan utility data is discussed, including issues of plan expense, generality, and impact on solution quality. Last, the results of this chapter are summarized.

Then, in Chapter 7, we discuss some of the additional results obtained from the use of SPATULA, including discussion of the additional method increments, and unexpected results both beneficial and problematic.

6.2 Description of Experimental Domains

The experiments to test SPATULA were performed in three domains: the Eight-Puzzle (EP), a Robot Domain, and the Tower of Hanoi (TOH). Appendices B, D, and C describe the domain operators and tasks used for each respectively. Each domain was created following the problem-space design guidelines of Chapter 3. These domains

were considered appropriate for SPATULA's interleaved approach to planning and execution as discussed in Section 4.4.2; in each domain, all mistakes are correctable eventually during execution if memory resources are not exhausted.

Below, we describe the search control used for each task, and any notable characteristics of problem-solving in the domain.

6.2.1 Eight-Puzzle

The Eight-Puzzle domain was chosen as a test domain because of its search-intensive characteristics and its history in previous abstraction work. For this domain, ten randomly generated tasks with ten-move solutions were tested. In the domain formulation used here, the EP domain has one operator, which moves a tile from square to square. The operator is instantiated with the tile and the source and destination squares. The operator has one precondition — the destination square must be blank. The condition that the tiles be adjacent was *critical*, in that it was represented as part of the conditions for proposing the operator. (There is no reason that experiments couldn't be run with this condition explicitly represented as a precondition as well. This is left for future work.) Thus, during abstract search with this representation, the tile may be moved to an adjacent square without regard to whether or not there is already a tile on the square. Measuring the number of steps in the shortest such search produces the "Manhattan Distance" heuristic [Pearl, 1983].

6.2.1.1 Search Control

The EP search control prefers operators which move tiles towards a goal; and of those, prefers operators whose preconditions are met. In addition, if an operator subgoal is generated because the operator's precondition is not met, search control proposes operators which move the tile blocking the destination square; one operator is proposed for each direction of movement other than back onto the source square.

6.2.1.2 Domain Problem-solving Characteristics

The EP domain and search control were such that if the problem solver managed to make good choices at its first few decisions, and started on a good search path, there were then relatively few subsequent choice points and it was easy to find a solution. (This may have been due in part to the relatively short optimal solution lengths — 10 moves — in the tasks tested.) In addition, these good solution paths tended to involve very little operator subgoaling, since search control preferred operators with preconditions met.

Of the ten tasks tested, about half of the operators in a task's initial top-level tie, if chosen, would lead easily to the solution. However, the other half of the choices in the top-level ties, which led down bad paths, generated searches in which it was very difficult to reach the solution — tiles would get further and further out of place, and the search control was not sufficient to reorder them.

6.2.2 Robot Domain

The Robot Domain was developed from the domain used in the ABStrips work [Sacerdoti, 1974]. Two versions of the robot's world were used for the tasks run in this domain. The first version based the tasks' initial states upon the original room layout used for ABStrips. The second version used initial states in which the complexity of the room layout and the connections between the rooms were increased along one dimension — that of the number of rooms and the connections between them.

Figure 6.1 shows a 2-goal-conjunct task in which the ABStrips layout is used. Figure 6.2 shows the same task, using the more complex room layout. For each task tested with both layouts, the initial configuration of the original rooms and doors remained the same. Thus, the optimal solution for each task remained the same as well — if the problem solver wasn't led astray by the extra information, it could solve the task using the complex layout in exactly the same way as in the original layout.

The tasks used for the Robot Domain experiments were drawn from randomly generated sets[1], grouped according to number of goal conjuncts. Tasks with 2, 3, and

[1] We would like to thank Craig Knoblock for providing these benchmark tasks.

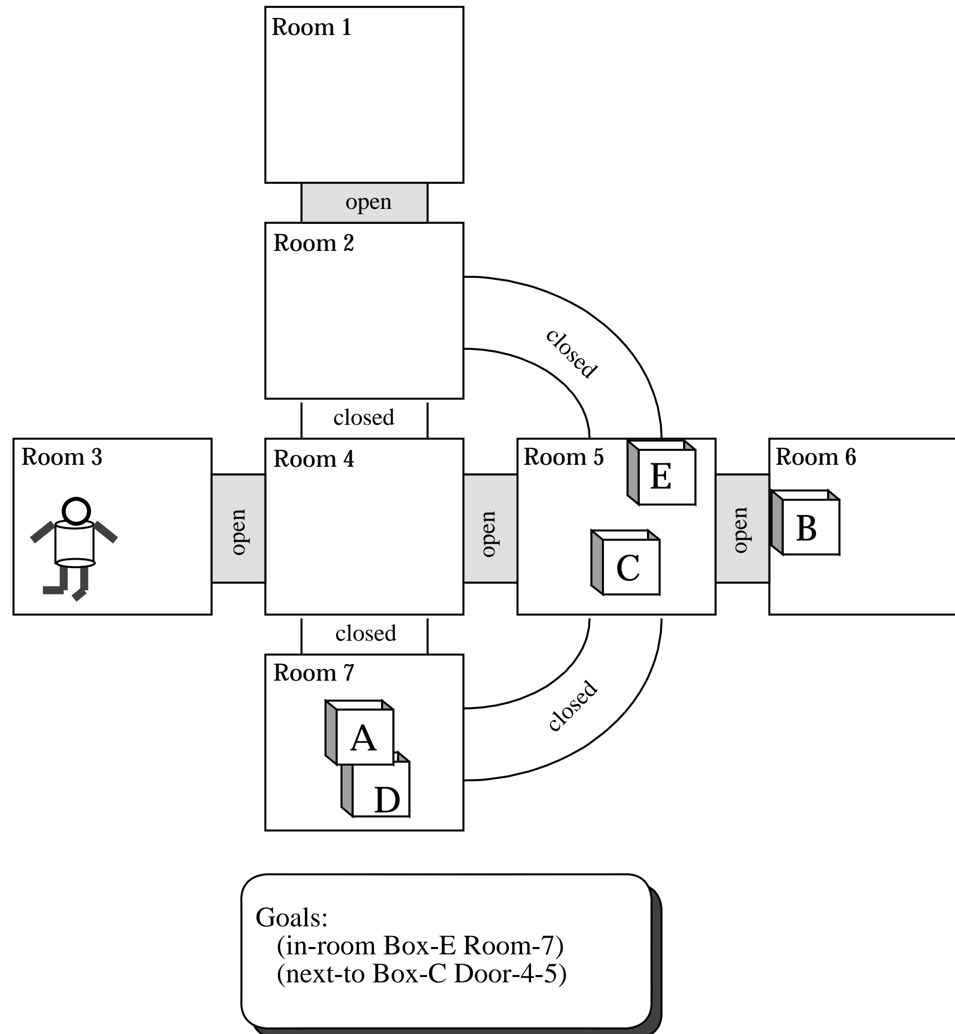


Figure 6.1: A typical task in the Robot Domain, using the "ABStrips" room layout.

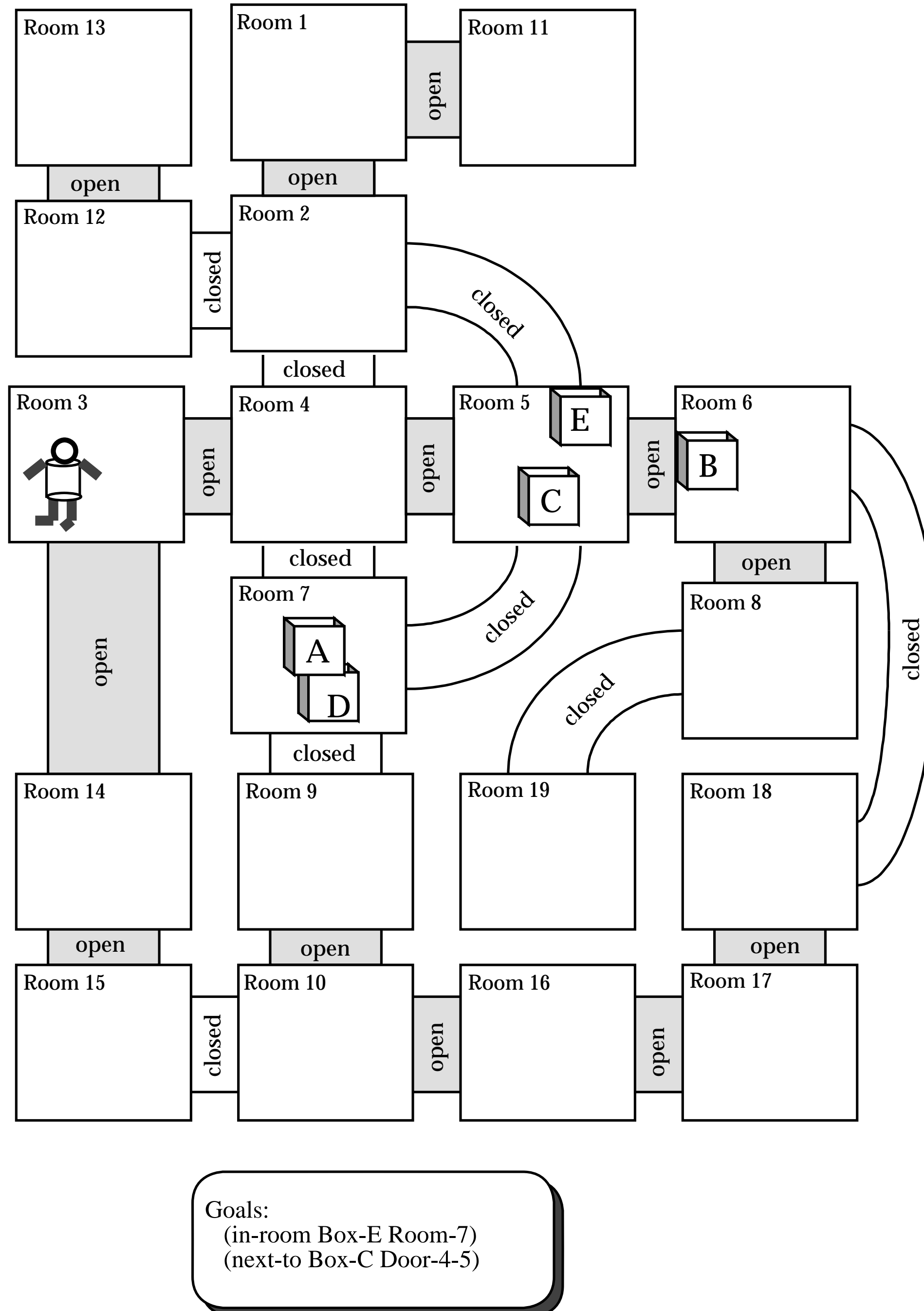


Figure 6.2: The same task as in Figure 6.1, using the more complex room layout.

4 goal conjuncts were tested.

6.2.2.1 Search Control

The problem solver was provided with MEA knowledge about the Robot Domain. The MEA knowledge took the form of search control which, given a goal conjunct, suggests operators which are likely (but not guaranteed) to help achieve the subgoal[2]. For example, if the robot has a goal to be in a particular room, operators will be proposed to move the robot into the room from adjoining rooms. However, in actuality it may not be possible to reach (all of) the adjoining rooms from the robot's current position.

6.2.2.2 Domain Problem-solving Characteristics

Using the original room layout, many tasks are fairly easy to solve in this Robot Domain; since the paths between rooms are fairly short, choices of bad paths have relatively minor repercussions. Exceptions are tasks in which there is a large amount of subgoal interaction. However, with the complex room layout, the subgoal search is more difficult; bad choices can cause a large increase in task difficulty and solution length.

6.2.3 Tower of Hanoi

The TOH domain is a favorite in abstraction research, since its recursive nature admits to nice hierarchical abstractions of the problem. For example, the top n disks may be aggregated into one or ignored, with n decreasing as search becomes less abstract. Because SPATULA does not perform the pre-search problem-space analyses which

[2]Although many systems, e.g. "Prolog-based" problem-solvers, use a single-representation trick such that the preconditions of an operator *are* the new task goals, it is not required in Soar that there be an explicit connection between knowing about operator preconditions and knowledge about how to achieve the preconditions. They may be described using different vocabulary. Therefore, in the Robot Domain, the problem solver was also provided with search control knowledge which suggested particular task subgoals given particular unmet operator preconditions. For example, an unmet precondition "`not(robot-next-to-door)`" will cause a suggestion that a task goal be posted to move the robot next to the door.

lead to these recursive approaches, we were interested in exploring the abstractions produced in such a domain.

In the formulation of the TOH domain used here, there is one operator, which moves a disk to a peg. Preconditions of the operator are that the disk be *clear* (no other disks on top of it) and that the top disk on the target peg not be smaller than the disk being moved. Thus, during abstract search, it is possible for a disk to be moved on top of a smaller disk, or for a disk to be moved without clearing out the disks on top of it.

Tasks with 3 and 4 disks were tested. Tests included the "canonical" task in which all disks start on one peg and end stacked on another, as well as other tasks with the disks in different randomly generated initial configurations and the same goal. Task goals were specified by giving *only* the target peg for each disk (and not specifying the order in which the disks should be placed on their pegs).

6.2.3.1 Search Control

The TOH Domain was provided with the following two search control rules. First, if a disk to be moved is not clear, then move the disk on top of it out of the way. Second, if a disk on a peg is smaller than the disk to be moved to that peg, then move the first disk to a different peg (note that this second rule does not specify the order in which the pegs on a disk should be moved off). With this search control, search was still required to determine which potential ordering of disks could create a legal stack.

6.2.3.2 Domain Problem-solving Characteristics

Given loop detection, the TOH is a relatively constrained domain. Although there are many possible non-optimal moves given the search control described above, the bad paths are not very long.

6.3 General Experimental Methodology

In the experiments, we first investigated the effects of the basic abstraction method alone. Then, more extensive tests were performed to include the method increments. For the tests which included the method increments, a series of tasks from each domain of Section 6.2 was run using variants of SPATULA (i.e., with different method increments), as well as two non-abstract search methods.

Empirical comparisons were made, across problem-solving methods, of task solution quality and problem-solving performance. In addition, the interaction of learning and abstraction — the expense, transfer, and utility of learned rules — was explored in the TOH and Robot Domain[3].

Analyses were also done of trends in these measurements across changes in domain characteristics. In addition to comparing results across domains, comparisons were made in the Robot Domain across tasks increasing in two different dimensions of difficulty: number of goal conjuncts[4] and complexity of domain. In addition, qualitative analyses were done of the types of abstractions which were produced in each domain, and the conditions under which they occurred.

6.3.1 Default *Weak* Problem-solving Methods

All test methods in all domains used the default *lookahead* method of depth-first search described in Chapter 2. (There exist more efficient (but perhaps less cognitively plausible) search methods, such as IDA* [Korf, 1985a], but since all tests in a domain used the same default search knowledge, comparisons show relative performance). For all domains tested, the problem solver was provided with default rules about *operator subgoaling*. Therefore, if an operator is selected but — because its preconditions are not all met — is not able to apply, then the problem solver knows how to set up a subgoal in which to search for a state from which the operator can apply. (Note that the operator subgoaling method does not assume the presence of any *particular*

[3]Although these tests were not made in the EP (the first domain tested), there is no technical reason why this could not be done as well. This is a topic for further work.

[4]Although tasks with a larger number of goal conjuncts weren't necessarily more complex on a task-by-task basis, this was the case overall.

search control knowledge guiding the problem solver to a state in which the operator can apply. The search control knowledge used by each domain was described above in Section 6.2).

The default goal achievement strategy used for all domains was linear, in that initially search control was employed to work on only one task goal conjunct at a time. If the current goal spawned task subgoals, then focus would shift to choosing an ordering on the subgoals and achieving each in turn. (See [Rosenbloom *et al.*, 1992] for a description of the implementation of other goal achievement strategies in Soar). The system's default knowledge about management of goal conjuncts was such that all in-focus goal conjuncts were considered equally for achievement (that is, no *a priori* goal ordering was used, as in Prolog). Therefore, even if the system was given domain knowledge about what actions were likely to achieve its subgoals, it might still need to search to determine which task subgoal to work on first, as well as how the subgoal might best be achieved. Often, characteristics of the other subgoals can affect how best to achieve a particular subgoal. Because the goal management knowledge was default, it could be overridden by more domain-specific knowledge about which goal to work on first.

The system noted when subgoals were achieved and clobbered, but did not backtrack at clobbered goals; instead it reachieved them. The initial goal achievement strategy was linear, but the problem-solver was not restricted to employing the linearity assumption; if its linear strategy failed, other operator orderings could be utilized, in which subgoals were achieved in an interleaved manner.

Detection of problem-solving loops is not provided architecturally in Soar. Knowledge to detect operator-subgoaling search loops was provided as part of the system's knowledge. However, in the Robot Domain and TOH, the problem solver did not have the knowledge to detect a particular kind of loop in which it had re-achieved a previously achieved state of the world within a single goal context. Runs which looped in this way were omitted from the data below; they do not fall into the same category as tasks which were unable to be completed for intractability reasons, and it is hard to estimate the statistics (number of problem-solving steps, etc.) they would have generated had their loops been detected.

6.3.2 Learning

It was taken as a "given" in our tests that the Soar architecture should learn from its problem-solving, be it abstract or non-abstract, since the integration of learning and problem solving is an important part of the Soar problem-solving architecture. However, the interaction between abstraction and learning is of particular interest. Since abstraction produces rules which are more general than their non-abstract counterparts, there is a greater possibility that the abstract rules will be over-general[5]. (This issue is distinct from that of whether the abstract lookahead search causes one operator to be mistakenly chosen over another; it may be that a good decision is made as a result of abstract lookahead search, but that the resultant rules apply in a non-useful future situation.) In addition, the abstract rules might be more expensive to use. Therefore, though learning and plan use was considered to be an integral part of the abstraction methods, analyses were done to determine the impact of the abstract plans.

6.3.3 Comparison with Non-abstract Problem-Solving Methods

Two non-abstract problem-solving methods were selected to provide comparative information about the abstraction methods: *non-abstract best-path search* and *non-abstract first-path search*.

Non-abstract best-path search requires that the problem solver compare all search paths — with respect to its existing search control, problem-solving, and evaluation methods[6] — before making a decision. Theoretically, such searches, since they use depth-first lookahead, can become infinitely deep. However, our search spaces were finite, and as discussed above, in each of our experimental domains the problem solver was provided with knowledge about cutting off most search loops. The exceptions were removed from the data set.

[5]It is possible for non-abstract rules to be over-general too [Laird *et al.*, 1986b], but this does not occur as often.

[6]Therefore, the problem solver will not necessarily search *all* legal paths, since it is biased by its other knowledge.

Best-path search provides a measure of how hard it is to come up with the best solution to a task, with respect to the system's problem-solving methods and biases. Therefore, since the abstraction methods we tested used the same lookahead search methods as the best-path search, a comparison allows analysis of the effects of abstraction on the search. In contrast, with first-path search, the problem solver just selects randomly from its set of possibilities (constrained by existing search control), at each new operator tie during lookahead search. The depth-first search terminates when some solution is found (again, though theoretically this search can be infinite, this was not a problem in our tests). The better the domain search control, the better the problem solver will perform when using this method. Such first-path searches indicate how tractably a domain responds to decisions made solely on the basis of existing search control, and what penalties are paid because of the need to repair random mistakes during search. In addition, the comparison of abstract and first-path solutions indicates the extent to which solutions produced using abstraction improve on existing search control.

6.3.4 Data Collection

Performance measurements were made of both the *solution quality* and *efficiency* of the tests run, since an analysis of the cost-effectiveness of an abstraction method includes both considerations of how good the solution is and how easily the solution is produced. Comparisons of these measurements then help indicate what gains and tradeoffs are produced by using the various method increments and method increment combinations, and what changes occur across methods as the domain complexity changes.

The data collected for some method increment combinations is more sparse than others (that is, only a subset of the tasks were tested for some combinations). The reason for this is that it was sometimes difficult and slow to do the test runs, and we wanted to obtain data points from a relatively large area of the experimental space.

Results are shown in bar graphs, which facilitate visual comparisons of relative differences between methods. In this chapter, summary graphs are presented, which show average measurements across the tasks in a domain. Then, in Appendix E, more

detailed graphs are listed, showing the individual task measurements. The differences in measurements across search methods were tested for significance using a pairwise Student T-test, with $\alpha = .05$. Unless otherwise indicated, all differences reported below are statistically significant.

The tasks which were solved using first-path search necessarily produced random results (with respect to existing search control biases), since at each choice point the problem-solver picked at random from its options. For the tasks solved using abstraction there was a smaller degree of randomness generated by situations in which abstract rule transfer suggested that more than one option was equally good. For such tasks, where several different runs of the same task and method were available, the results were averaged for that task. However, since the tasks themselves were randomly generated, it was not considered necessary to run multiple trials of each such task. Instead, the cumulative results of all such tasks are expected to be randomly distributed over the spectrum of possible results — some solutions will be better than average, and some will be worse.

6.3.4.1 Solution Quality and Problem-solving efficiency

In all experimental domains tested, the metric used by the domain evaluation functions was solution length — those task solutions which required the fewest number of operator applications were considered the best. Therefore, when evaluating the empirical results in each domain, solution quality was evaluated by comparing the solution lengths of the tasks. Since the problem-solver, using SPATULA, interleaves abstract planning and execution, the solution lengths discussed below include the execution of any repairs which may have been required if mistakes were made in the execution space.

Problem-solving efficiency was measured along two different dimensions: the number of Soar *decision cycles* (i.e., problem-solving steps) required to complete a task, and the computational resources necessary. (Recall from Chapter 2 that Soar's decision cycles include selection of problem-spaces and states as well as operators, and generation of goals.)

The number of problem-solving steps includes both planning and execution steps.

However, execution represents only a small proportion of the total number of steps; therefore, the total number of steps indicate how hard it was for the problem-solver to make decisions about what actions to take. For example, if contradictory abstract search control caused an operator *conflict*, the extra work to resolve the conflict would be reflected in the number of problem-solving steps required. If plans learned during earlier search do not apply once part of the plan has been executed, the necessary replanning would again be reflected in the number of problem-solving steps. Similarly, if the problem-solver makes a mistake and needs to patch, this is reflected also.

Soar's decision cycles are hypothesized to represent cognitive steps [Newell, 1990]. Thus, given the "right" distributed implementation of Soar, it is hypothesized that they will take constant time. If this turns out to be the case, then the number of problem-solving steps is the most telling measure of problem-solving difficulty. However, in the current implementation, the problem-solving steps can take variable amounts of time.

In Soar, average problem-solving time per step can increase for several reasons. It may be because there are many rules firing at once. This type of slow-down would be expected to be favorably impacted by a distributed implementation of Soar. Problem-solving slowdown can also occur because rule matching is more expensive. The expense of rule matching is often increased as the problem-solving subgoal depth is increased, since there are more goal contexts to match against the deeper the goal stack becomes. Thus, tasks with deep goal stacks will usually run slower per step than tasks with shallow goal stacks. In addition, the amount of time spent building new rules can also impact the overall problem-solving time.

Therefore, for our tasks, we looked at the following measures of computational resources in addition to the number of problem-solving steps. These measures are implementation-dependent and — in the case of time measures — machine-dependent, and thus are not meaningful as absolute measurements. However, with respect to the version of Soar used for these tests, a comparison across search methods gives an indication of the relative expense of the different methods:

- Total problem-solving time — measure of total computational resources required. Unfortunately, this measure was not always reliable[7]. Though it provides some indication of relative problem-solving effort, exact comparisons should be taken with a grain of salt.

- Total *token* changes, and average token changes per rule firing. The size of Soar's token memory at any point indicates the number of matches for each condition of each rule. The total number of token changes registers the changes in token memory over the course of a task, and thus measures the effort required to match and fire rules during the task (though not to build new rules). This expense is affected by the particular conditions (left-hand-sides) of the rules as well as the goal stacks generated while solving a task.

- Percent learning overhead — the percent of problem-solving time spent building new rules.

- Average time to build a new rule.

- Number of new rules learned.

The summary figures in this chapter and the next do not present all measurements for all tasks. More detailed data are in Appendix E.

In the presentation of the data, some tasks are marked as intractable with respect to a problem-solving method. Intractability could occur in three ways. In the EP and the Robot Domain tasks, a cutoff was set of a maximal number of problem-solving steps. If a task took more than this number of steps, it was considered intractable. In addition, a task could exceed its memory limits before reaching the cutoff number of steps if its token memory became too large; this was defined as intractability as well. Finally, a few of the abstract tasks were terminated because many abstract rules were

[7]This was the case for at least two reasons. First, in the implementation of Soar used for these tests, old internal data structures were retained during problem-solving, thus causing longer runs to gradually increase in time per step. This problem has since been corrected. Second, system processing time appeared to be included in the reported time. For example, the same task could vary in time by more than a factor of four depending upon whether or not other processes on the machine were requiring a lot of CPU time.

firing at some points during problem-solving, causing some problem steps to be very slow. (The reason for this effect, and the tasks with which it occurred, are described in Chapter 7.) All tasks were run with the same memory allocation and thus all had the same intractability limits. Clearly, the specific limit set on memory usage affects the results; some of the tasks would have finished if they had been allotted more memory. However, given a particular memory limitation, the results provide comparative information across methods.

Measurements of expense don't completely reflect the difficult end of the spectrum of tasks, since 1) for technical reasons, it was not always possible to gather final statistics on tasks which ran out of memory; and 2) such statistics would not be directly comparable to those of tasks which had finished. For example, disproportionate amounts of learning might have occurred. Thus, numerical data was in general gathered only for those tasks which were able to finish, and these measurements must be viewed in conjunction with the information on the number of tasks which were intractable.

6.4 Results: Basic Abstraction Method

As discussed in Section 5.1, SPATULA's basic abstraction method, without use of the method increments, can be useful in domains with very little search control. In these domains, abstraction permits the problem solver to more easily guess which operators are likely to achieve an aspect of the current goal and which are likely to be unrelated. In Section 4.3.2, it was shown that abstraction allows the problem solver to learn new MEA knowledge. However, as discussed in Section 5.1, the conditions under which the basic method produces discriminatory information may not arise very often when using goal-driven search control. Such conditions in fact rarely occurred in our test domains — side effects were usually hidden when all preconditions were abstracted, and the MEA knowledge (at this highest abstraction level) always caused operator sequences of equal length to be suggested. At the highest abstraction level in the TOH and Robot Domain, one operator would be proposed for each goal conjunct. In the EP domain, a goal conjunct could cause a sequence of operators to be proposed, but

these sequences, based on the "Manhattan Distance" to the goal state, were always of the same length as each other. Thus, for our tests, SPATULA's basic abstraction method alone was not useful.

In fact, the basic abstraction method alone gives results similar to those produced by *first-path* search (in which the problem solver chooses randomly at an operator tie), except that the system will have to repair or back out of mistakes at the execution level rather than within lookahead search. Experiments which used earlier versions of our test domains, though not presented here, confirm this similarity. Thus, the first-path results presented below provide information about the improvement in solution quality when abstraction method increments are added to the basic method.

6.5 Results: Solution Quality and Problem-Solving Efficiency with SPATULA's Method Increments

The *assumption-counting* method increment, described in Section 5.2, provides the problem solver with a heuristic for estimating the relative difficulty of expanding its abstract plans, and lets it incorporate this information into its decisions. *Iterative abstraction*, described in Section 5.3, provides a heuristic for finding a useful abstraction level, by iteratively increasing level of detail during search until the options of a control decision are discriminable. In this section, we analyze the impact of these two method increments on the experimental tasks, with respect to solution quality and problem-solving efficiency. Our experiments showed the method increments tended to work together synergistically, each improving on the results that would have been obtained by using one method increment alone. Together, they increased problem-solving efficiency while producing good (and at times optimal) solutions.

In the Robot Domain, results presented here are from runs which used the efficiency-driven method increments of Section 5.7 in addition to iterative abstraction and assumption counting, to increase tractability of abstract search for the harder tasks. With the goal achievement iteration method increment, the initial number of goal

conjuncts to achieve was set to three (one less than the highest number of goal conjuncts in the Robot Domain tasks tested). Section 7.4 presents a further set of tests done to gauge the impact of these methods on the experimental tasks in all domains.

In the TOH, results presented in this section are from tests which used the *extended-plan-use* method increment of Section 5.6 in addition to iterative abstraction and assumption counting. Section 7.3.3 will discuss the impact of the extended plan use method in both the TOH and Robot Domain. In addition, the TOH results presented here are from tests in which all iterative abstraction iterations were started at the second iteration level — that is, for these abstract searches, at least one level of preconditions was always required to be achieved. It was shown that in this domain, the problem solver never will have enough information to make a decision at the first iteration level (that is, it will always iterate). Thus, searching at the first iteration level in the TOH domain is a waste of effort. The issue of "tuning" the initial iteration level was discussed in Section 5.5, and is further addressed in Chapter 9.

In this section, we will present the numerical results obtained from testing iterative abstraction and assumption counting in our three domains, by analyzing solution quality and problem-solving efficiency. After presentation of the results for each domain, the results will be interpreted and trends discussed. Then, in Section 7.2 of the next chapter, a qualitative analysis will be made of the types of abstractions which were produced using these two method increments, and the situations in which the methods were the most and least helpful for the domains tested.

6.5.1 The Eight Puzzle

The EP domain, while simple to describe, can be difficult to search because of the potentially large amount of interaction between placement of the tiles. In this domain, SPATULA proved useful as a means of tractably estimating useful search paths, while avoiding the complexity of searches down bad paths.

Ten randomly generated tasks of solution length 10 were run in the EP domain. For these tasks, once the problem solver had started on a good path, it was easy for it to find a solution, given the domain's search control. However, a bad decision at an early choice point tended to be hard for the problem solver to recover from;

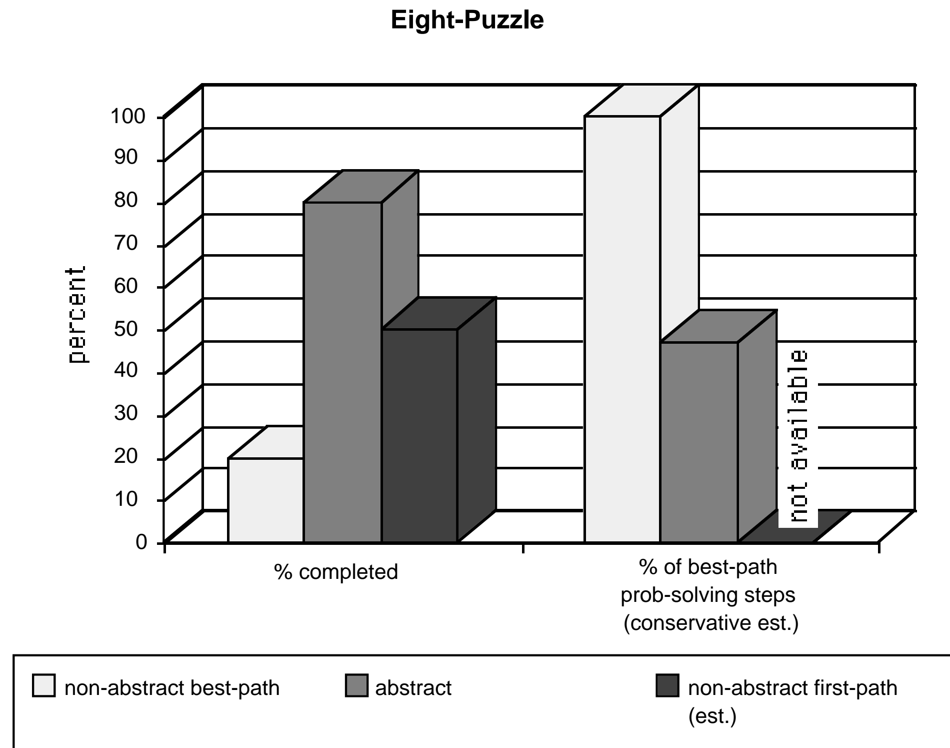


Figure 6.3: Eight-puzzle: comparison of search methods.

cascading interactions would move it further and further from a solution. Tasks which took more than 4000 steps were cut off and considered intractable. For this set of tasks, it was uninformative to compare the quality of the solutions produced. All solutions produced by runs in which the system managed to complete the task were optimal or close to optimal. Therefore, task completion indicated good decisions at choice points and thus high-quality solutions.

Figure 6.3 shows the percentage of tasks completed in this domain for each method of search. Only 2 of the tasks were easy enough to allow non-abstract best-path solutions. In contrast, 8 of 10 were solvable using abstraction. The figure for the percentage of non-abstract first-path tasks not completed is an estimate. The percentage of top-level choices which led to intractable paths was shown to be 50% over all tasks. This figure was obtained via a set of tests, one for each top-level operator-tie option of each task, in which the problem-solver initiated problem-solving by selecting

that option. Thus, it would be expected that from a pool of random first-path trials with these tasks, about half would not be able to complete[8].

The figure also shows the average percentage of problem-solving steps required to solve the tasks abstractly, with respect to the number of steps required for non-abstract best-path search. Because unfinished tasks were cut off at 4000 steps, the calculated difference between abstract and best-path search is conservative and would in reality be even greater. For those first-path searches which randomly chose good paths and thus managed to complete, it is expected that the number of problem-solving steps would have been relatively short compared to both other searches.

The results indicate that in the EP domain, decisions made by problem-solving with iterative abstraction and assumption counting are of better-than-random quality. Thus, the advantage gained from abstraction over both types of non-abstract search in this domain was the ability to explore all search paths at a high level of detail in order to pick a useful one, without becoming bogged down in those paths which were not likely to be promising. In this domain, abstract search tended to be good at finding the easy solutions; if an operator had a good abstract evaluation, it was usually on a good path.

Because the non-abstract problem-solving was largely intractable, most final statistics from non-abstract search, including problem-solving time, were not available. The available data are too sparse to be statistically significant, but suggest that in the EP domain, problem-solving time was reduced using abstraction for even the easier tasks. Had statistics been available for the harder tasks, the difference would be expected to be even more pronounced.

6.5.2 The Tower of Hanoi

In the TOH domain, an interesting result was obtained: all solutions produced using iterative abstraction and assumption counting were *optimal*. That is, using abstraction, the number of external moves made by the system was the minimum possible.

[8] Actual first-path searches were not performed in this domain, which was the earliest tested. However, there is no technical reason that this could not be done. In fact, earlier experiments with a slightly different EP version produced the result that less than half of a set of first-path tasks completed.

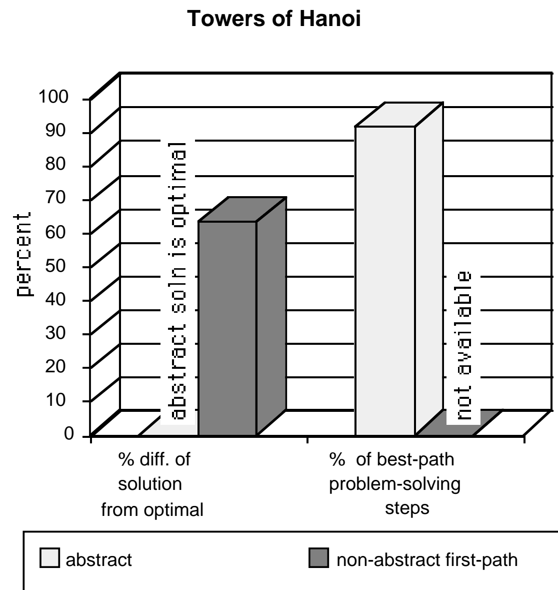


Figure 6.4: TOH: average solution quality and problem-solving steps.

The optimality of the solutions using abstraction was independent of the number of disks in a task, or initial and goal disk configurations, and stemmed from the interaction between iterative abstraction and assumption counting — neither one alone would have produced the optimal results. In addition, the abstraction method increments were created before this domain was tested, and thus the TOH's characteristics did not drive their development in any way.

The optimal strategy was a rather novel one, in which the system, at each choice point, picked the easiest of the subgoals along the optimal solution path to achieve next (rather than following the more common fully recursive strategy). This strategy is an instantiation of SPATULA's easiest-subgoals-first bias described in Section 5.8, and is discussed further in the following chapter.

For the tests in this domain, 4 tasks each were run for the 3-disk and 4-disk configurations, using both best-path and abstract search. For some runs, learned rules caused looping in goal conjunct selection during first-path search, and this caused difficulty (unrelated to abstraction issues) in collecting a large number of first-path runs. Six instances of first-path search were averaged to provide the first-path solution

quality data, and 4 of these runs were simulated by hand — using a random-number generator to select moves at decision points — to provide a better estimate of an average first-path solution. Figure 6.4 shows results for the 4-disk task. The figure shows that the optimal solutions produced by abstraction were significantly better than those generated by first-path search.

The figure also shows that with abstract search requiring about 90% of the number of steps of the non-abstract search, there is not a large efficiency difference between the two methods (though the difference is statistically significant). In this formulation of the TOH domain, the operator subgoals were relatively shallow, and thus did not provide SPATULA with an opportunity to avoid a large amount of problem-solving. As might be expected, the task with the deepest subgoals provided the largest savings from abstraction. The effect of shallow subgoals was particularly noticeable for the 3-disk tasks. Here, the difference between abstract and non-abstract search was minimal because very few abstractions were made during search, and thus the 3-disk results are not shown in the figure.

Because of operator subgoaling loop detection provided to the domain, it was relatively easy for the system to find a solution using first-path search, since — given the shallow operator subgoals — mistakes were easily recovered from, though such solutions were not very good. However, because of the above-mentioned difficulty with looping, only sparse data was available on the number of problem-solving steps for first-path search. Existing data suggests that the first-path searches take approximately 10% of the steps of best-path search.

The TOH five-disk tasks proved to be too hard for the system, both with non-abstract best-path and with abstract search. This was because the combinatorics of exploring all orderings of the five top-level goal conjuncts caused the problem-solver to run out of memory before it was finished. This illustrates the theoretical result discussed in Section 5.9. See Section 7.5 in the next chapter for further discussion of this issue[9].

[9] This five-disk limit was specific only to Soar — a lisp simulation was able to solve larger problems.

6.5.3 Robot Domain

In the Robot Domain, the solutions produced using abstraction were more variable than those in the two other domains; this reflected the increased variability of the domain, operators, and preconditions. In addition, the experiments in this domain provided an exploration of the problem-solving trends resulting from an increase of both task and domain complexity. In this section, we group the results by number of goal conjuncts in the task. Then in the following section, trends across sets of tasks are examined.

In the solution quality figures for this domain, "optimal" solutions are those that were produced by the non-abstract best-path problem-solving method — that is, they are optimal with respect to the other problem-solving biases of the domain. (For any non-abstract problems which were intractable, the optimal solutions were figured by hand.) In the results cited below, all average differences between methods are statistically significant unless otherwise noted.

6.5.3.1 2-Goal-Conjunct Tasks

Figure 6.5 shows the results from the set of 2-goal-conjunct Robot Domain tasks tested. The figure shows relative solution quality, number of problem-solving steps, and percentage of tasks completed. Data is shown for both the original and complex room layouts of Figures 6.1 and 6.2. For this set of tasks, non-abstract best-path runs which took more than 2600 steps were cut off. Thus, the average numbers for the non-abstract best-path method are lower than they would in actuality be.

Figure 6.5 shows that for both the original and complex room layouts, abstract search produced better solutions than first-path search. In addition, as domain complexity increased, the disparity in the abstract and first-path solution quality also increased. This is a notable result, since the more complex layout would seem to provide no additional advantage to abstract search; in the complex layout, the opportunities both to explore and choose longer paths are increased (though the optimal solution paths remain the same as in the original layout). Thus, the complex layout provided the iterative abstraction technique with more chances to miss an important

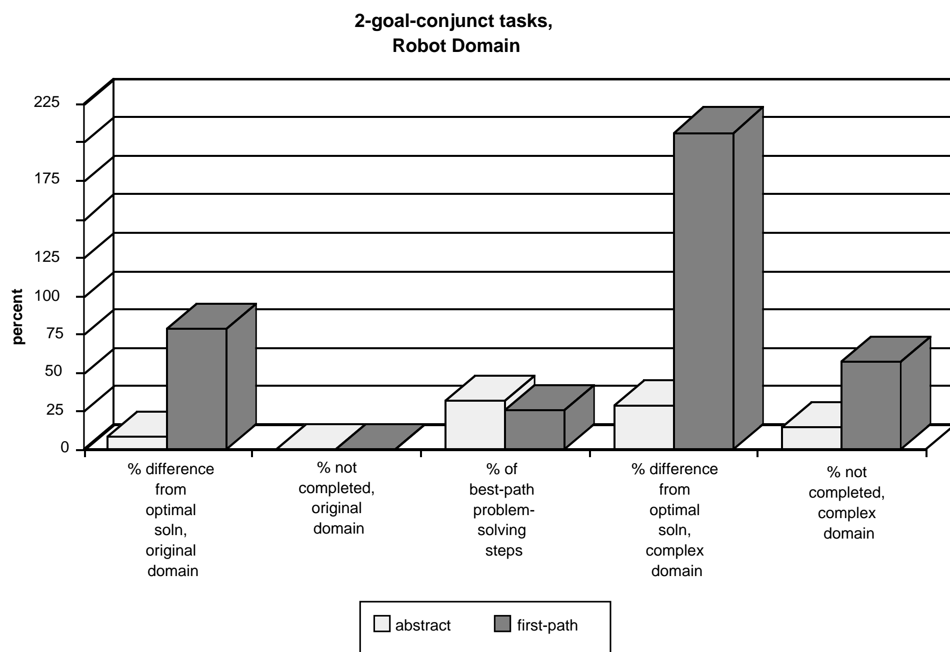


Figure 6.5: 2-goal-conjunct tasks in Robot Domain: average solution quality, percentage of tasks completed, and average problem-solving steps. 19 tasks were tested with the original layout. A subset of 14 of these tasks was tested with the complex layout.

interaction along a search path, by potentially halting iteration too soon at a spurious distinction, than did the original layout. Nevertheless, abstract search produced a markedly smaller solution degradation than did first-path search, with abstract solutions at 28% longer than optimal, and first-path solutions at 206% longer than optimal. This means that the use of abstraction continued to allow the problem-solver to make good decisions at choice points in the complex layout. In contrast, with first-path search, the increased complexity of the domain made the problem-solver's less informed choices more costly.

In terms of problem-solving steps, the difference between abstract and first-path search with the original layout was not significant; both took only a small percentage of non-abstract best-path steps. In the complex domain, non-abstract best-path tasks were not attempted (the non-abstract best-path searches would have had to explore the largely intractable first-path searches as just one of their possible search paths). However, the best-path searches would have been of greater difficulty than the best-path non-abstract searches in the original room layout, since the subgoal depths were greatly increased. In addition, they would have been of *relatively* greater difficulty than the abstract searches, since — as discussed in Section 5.9 — the relative savings achieved by abstraction increase as the subgoal difficulty increases.

The difference between number of abstract and first-path problem-solving steps remained small with the complex layout. However, with the complex layout, only a small percentage of the first-path searches were able to complete, as compared to all but two of the abstract searches. The intractability of the first-path searches in the complex domain was due to the long search paths and deep subgoals produced by the random search. Because the abstract searches tended to produce better decisions based on a shallower search, they did a better job of avoiding long search paths.

6.5.3.2 3-Goal-Conjunct Tasks

Figure 6.6 shows the results of tests run for 3-goal-conjunct tasks, again with both the original and complex room layouts. For these tasks, the non-abstract best-path method was (almost entirely) intractable; thus the optimal solutions are figured by hand, and thus there is no data shown on number of best-path problem-solving steps.

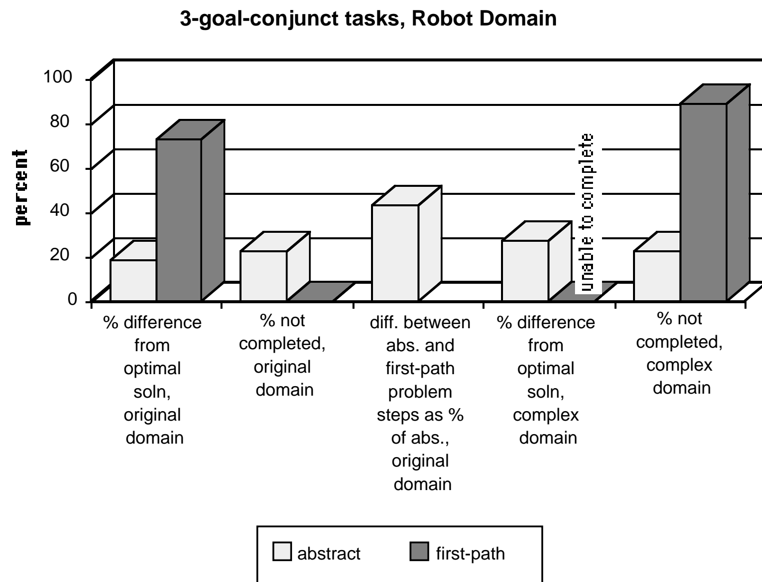


Figure 6.6: 3-goal-conjunct tasks in Robot Domain: average solution quality, percentage of tasks completed, and average problem-solving steps. 18 tasks were tested with the original layout. A subset of 9 of these tasks was tested with the complex layout.

However, had they finished, the difference in number of problem-solving steps between best-path and the other two search methods tested would have been greater than that observed for the 2-goal-conjunct tasks.

For the 3-goal-conjunct tasks solved with the original room layout, the abstract solutions were again much better than the first-path solutions. For these original-layout tasks, abstract problem-solving required a larger number of steps than did the first-path search. The difference stems from the abstract search to order the task goal conjuncts, a cost not incurred by the first-path search (which just selects goal conjuncts at random until the task is solved). Section 7.5, in the next chapter, discusses this issue further. To allow comparison between abstract and first-path search, we give the difference between abstract and first-path problem-solving steps as a percent of abstract steps required.

Of the 3-goal-conjunct tasks tested in the complex room layout, only 1 of the first-path searches finished without running out of memory. Again, the intractability

of the first-path searches in the complex domain was due to the long search paths and deep subgoals produced by the random search, and the 3-goal first-path searches showed increased intractability with respect to their analogs in the 2-goal-conjunct complex domain tests. The more goals in a task, the greater the opportunities to choose a search path which is globally intractable.

In contrast to the first-path searches, the percentage of uncompleted abstract searches did not increase with the complex layout. This means that abstract search continued to do well at avoiding hard search paths.

Because of the intractability of the first-path search in the complex layout, it was not possible to compare complex-layout solution quality of the two methods. However, observation during problem solving suggested that as with the 2-goal tasks, the solution quality of the complex-layout first-path searches was again much worse than with the original layout. In contrast, the complex-layout abstract solutions increased in length by only a small amount.

6.5.3.3 4-Goal-Conjunct Tasks

Figure 6.7 shows the results for the set of 4-goal-conjunct tasks solved with the original room layout[10]. Again, the use of abstraction produced better solutions than the use of first-path search. In addition, for this group, more of the abstract tasks are able to complete than the first-path tasks. The relative difference in solution quality produced by abstract and first-path search has decreased compared to that seen in the easier sets of tasks. However, examination of the 4-goal tasks suggested that the drop was spurious, and caused by the intractability of the randomly worse solutions. That is, only the easier 4-goal first-path tasks finished — those which randomly picked good solution paths. Inspection of those 4-goal first-path tasks which did not finish, and projection of the solutions they would have produced, suggests that the unfinished solutions would have produced a higher average solution length.

For the 4-goal tasks, there is a larger disparity than in the easier tasks between the number of problem-solving steps required for the abstract and first-path searches

[10]The complex room layout was not tested with the 4-goal-conjunct tasks; this remains for future tests.

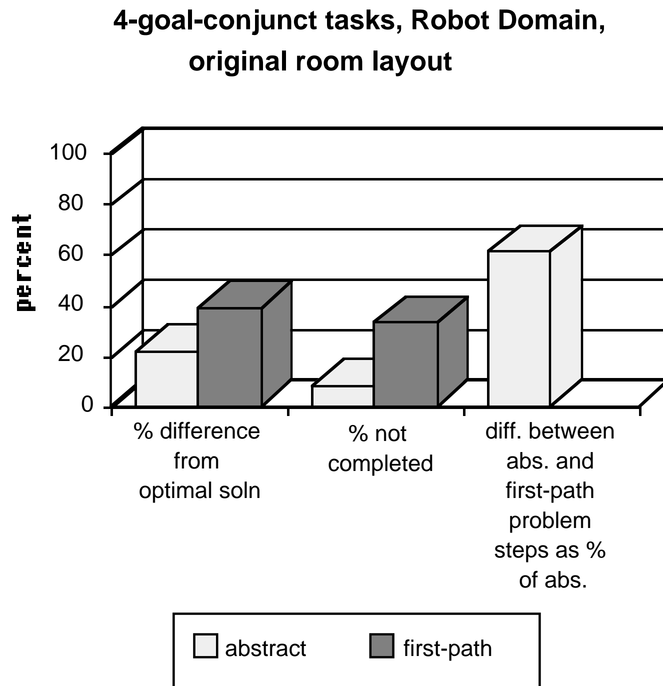


Figure 6.7: 4-goal-conjunct tasks in Robot Domain: average solution quality, percentage of tasks completed, and average problem-solving steps. 12 tasks were tested with the original layout. Evidence suggests that the relatively smaller difference in solution quality (as compared to the easier sets of tasks) is spurious, and caused by the increasing intractability of the solutions generated by the first-path searches (only the solutions from the tractable searches are recorded).

— again because of the increased goal ordering work done by the abstract searches. However, many more of the abstract searches, as compared to the first-path tasks, are able to complete at all. As before, this suggests that the larger number of goal conjuncts provide a larger opportunity for the first-path search to take difficult and unproductive paths, whereas the decisions made during abstract search — though they take longer to generate — lead to better paths.

6.5.4 Discussion

The TOH domain, because of its shallow subgoals, did not prove to be a good domain for producing large efficiency savings with SPATULA. However, it is an interesting

domain in that the solutions produced with SPATULA are optimal, though the development of the abstraction techniques had not been driven in any way by the TOH.

In the EP and the robot domain, the subgoals are deeper. Here, the use of SPATULA allows search that is significantly more efficient than best-path, while producing solutions that are significantly better than the solutions produced with first-path search. In the Robot Domain, the solutions produced using abstraction were more variable than those in the two other domains; this reflected the increased variability of the domain, operators, and preconditions. Several additional trends may be observed from these results.

6.5.4.1 Trends in Abstraction Across Increasing Numbers of Goal Conjuncts

Figure 6.8 summarizes the trends seen in the Robot Domain tasks across increasing number of goal conjuncts with the original room layout. As the number of task goal conjuncts increases, several effects are seen. The solutions produced by the abstract search degrade slightly as more goal conjuncts are added; the greater the task complexity, the greater the opportunity to make mistakes. (Exceptions are domains such as the TOH, where abstraction produces optimal orderings for any number of disks). The abstract solutions remain significantly better than the first-path solutions, though the 4-goal-conjunct tasks show a relative reduction in the difference. As discussed above, we believe this effect is due to the fact that a relatively smaller percentage of 4-goal first-path searches were able to complete – those in which shorter search paths (and thus better solutions) were generated.

The difference between the number of problem-solving steps required to solve a task abstractly and with first-path search increases as the number of goal conjuncts increases. This increase is due to the increased necessity of constructing an ordering for the goal conjuncts. The first-path searches do not carry out any goal ordering activity, and thus avoid this dimension of complexity.

First-path and abstract search both show increased intractability as the number of goal conjuncts increase, but for different reasons. With abstract search, as predicted in Section 5.9, as the number of goal conjuncts increases, a point is reached at which

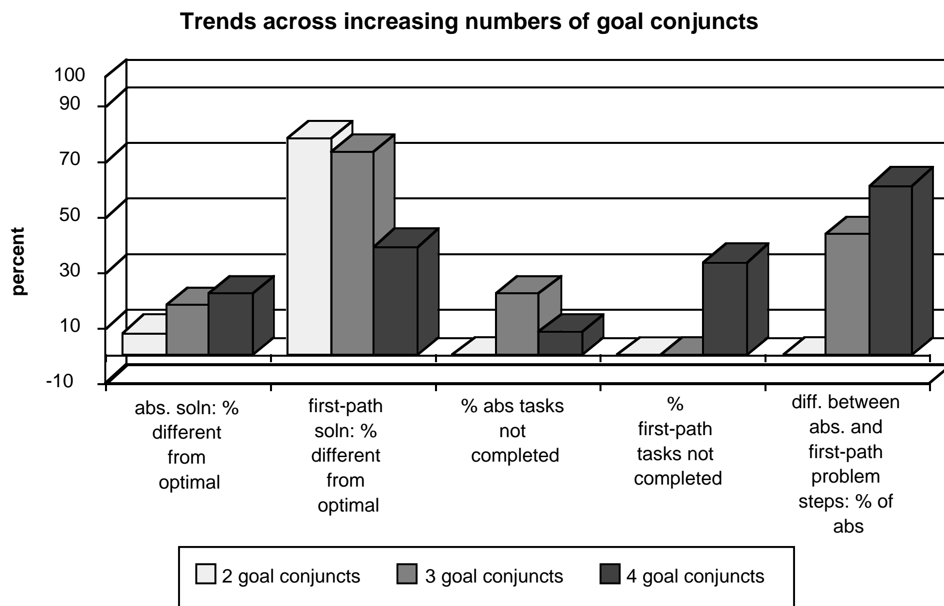


Figure 6.8: Summary of trends across increasing numbers of goal conjuncts in the Robot Domain, with the original room layout. Evidence suggests that the relatively smaller difference in solution quality for the 4-goal tasks is spurious, and caused by the increasing intractability of the solutions generated by the first-path searches (only the solutions from the tractable searches are recorded).

the increased effort required to order the conjuncts becomes prohibitive. The particular point at which this occurs is domain- and implementation-dependent. (Note, however, that non-abstract best-path search reaches this point much earlier than the abstract search; only the 2-goal-conjunct Robot Domain tasks were tractable with non-abstract best-path search). The results show that while SPATULA can greatly increase problem-solving tractability while producing good solutions, it can not do so indefinitely, as the number of goal conjuncts becomes larger, without the use of additional problem-solving knowledge about how to decompose the conjunct ordering process. See Section 7.5 in the following chapter for further discussion of these issues.

The first-path search also shows increased intractability as the number of goal conjuncts in a task increases, but for a different reason. With first-path search, the more complex the task, the greater the chance for the system to make mistakes and explore paths that are hard to recover from. Thus, with the 4-goal-conjunct tasks, the abstract searches require more problem-solving steps than the first-path searches which finished, but the superior solutions produced with abstraction allow more abstract searches to finish.

6.5.4.2 Trends in Abstraction Across Increasing Domain Complexity

The experimental results demonstrated that the more complex the operator-subgoaling search, the more efficient abstraction is with respect to non-abstract best-path search. This was observed by comparing the TOH with the Robot Domain and EP. The Robot Domain and EP, with harder subgoals, demonstrated a much better comparative savings in number of steps than the TOH, in which the operator subgoals were shallow and did not provide SPATULA with the opportunity to abstract away large portions of the search space.

The two Robot Domain layouts also provide a comparison of domain complexity. Figure 6.9 shows task results across the two layouts. In the more complex domain, both abstract and first-path solution quality is worse than in the original domain; this is not surprising since the complex layout affords a greater opportunity for costly mistakes. However, the solution degradation is much more marked for the first-path searches than for the abstract searches, even though the complex layout provides

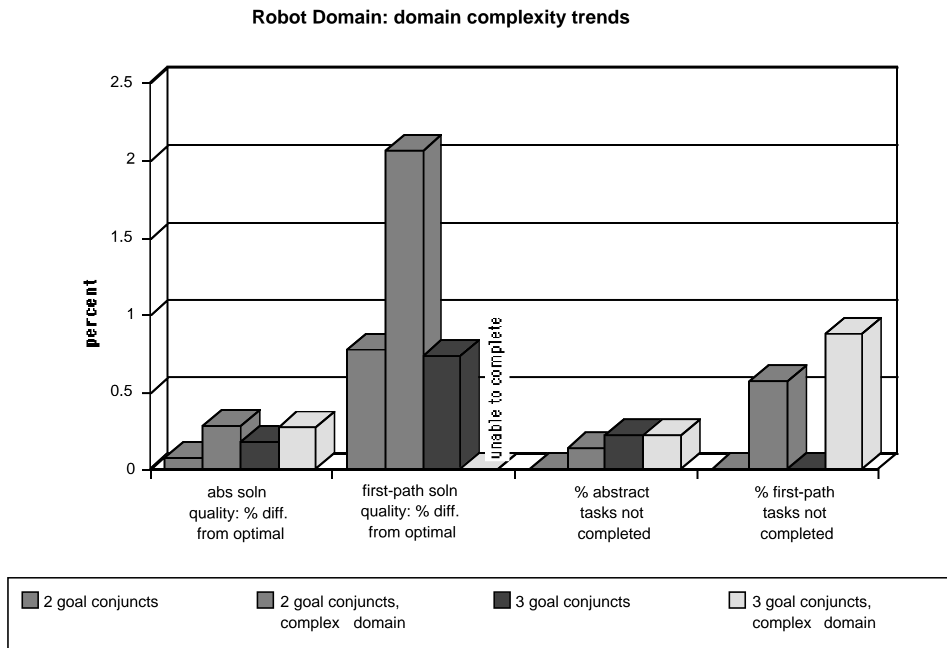


Figure 6.9: Summary of complexity trends in Robot Domain.

relatively greater opportunities for SPATULA to make mistakes, as compared to the original layout. This suggests that the use of abstraction continued to produce good decisions in the complex domain. With first-path search, more mistakes were made in both domains, and the cost of these mistakes — with respect to solution quality — increased with the complexity of the domain.

The figure also shows that the more complex the operator subgoaling in a domain, and the greater the interactions between subgoals, the more tractably abstraction performs when compared to first-path search. If search is complex, then mistakes have also greater consequences with respect to problem-solving efficiency than they would in a less difficult domain – they lead to a greater chance of the problem solver becoming bogged down in complex subgoals that cause unwanted interactions or are hard to achieve. Thus, as the cost of repairing random mistakes increases, the relative cost of a first-path solution can be expected to increase as well. This was borne out in the first-path results for both the EP as well as the complex Robot Domain, in which many of the potential search paths led to intractability. In contrast, the abstract

search in these domains remained significantly more tractable.

Thus, the experiments suggest an important result: the harder the domain with respect to the penalties paid (both in repair time and solution quality) for mistakes, the larger the payoff— in solution quality as well as efficiency — to be obtained by tractably finding a good plan. The use of SPATULA was shown to be one way to allow a good solution to be more tractably produced.

6.6 Abstract Plan Utility

Learning plays an important part in the abstraction process. The previous section discussed solution quality and problem-solving time when using SPATULA (with learning), but did not examine the type of abstract rules learned, or when they were used. Here, we examine the interaction of learning and abstraction. There are several aspects of problem-solving to consider in the evaluation of learned abstract plans:

- The cost of building the abstract plans.
- The generalizations produced by abstraction.
- The amount of transfer of the plans, both within the same task and to other tasks in the domain, and their subsequent impact upon solution quality and problem-solving efficiency.

All of the considerations above must be combined to give a global picture of plan utility, since for example, it is possible for an initial expense in learning plans to be amortized over subsequent tasks.

6.6.1 Expense of Building Abstract Rules

The expense of building new rules can be gauged both by the percent of problem-solving time spent building rules and the number of rules built; and by the cost of building a single rule.

It is not necessarily informative to compare the abstract and first-path searches' rule-building expense, since the coverage provided by the two sets of learned rules

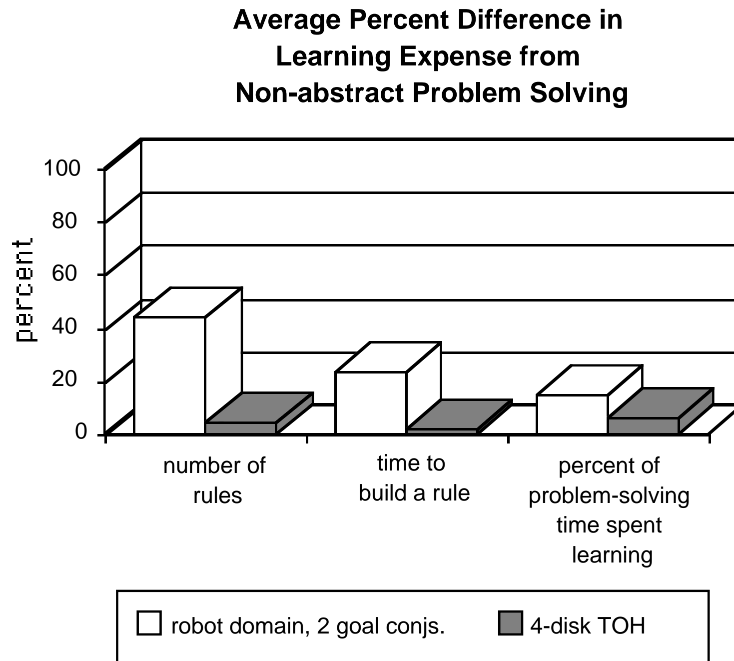


Figure 6.10: Summary of learning expense for abstract and non-abstract rules. The graph shows average difference as a percentage of non-abstract expense (i.e., $\frac{\text{non-abstract}-\text{abstract}}{\text{non-abstract}}$).

is different. Abstraction provides a wide but shallow coverage, and generates rules which compare many options at a high level of abstraction. First-path search provides a deep and narrow coverage, and generates rules which learned in detail about one particular path, but which do not provide information about other parts of the search space. However, the average learning overhead and cost per rule was in fact less for abstraction than for first-path search throughout the Robot Domain tasks.

Non-abstract best-path search produces the same type of coverage of learned rules as does the abstract search, in that both sets of learned rules cover *comparison* of options at a control decision. Therefore, the percent of problem-solving time spent learning rules, as well as the average time to build a rule, are comparable across these two methods. These data were available for the TOH domain, and the 2-goal-conjunct tasks in the Robot Domain (as discussed above, the non-abstract searches were intractable for the harder Robot Domain tasks).

Figure 6.10 summarizes the results for both domains, by showing the average difference between non-abstract and abstract learning, as the percentage of non-abstract expense. In both cases, the average percentage of problem-solving time spent learning was less for the abstract than non-abstract search. Both differences were statistically significant, but the difference was less in the TOH than in the Robot Domain. The average difference in time to build a rule, as well as the number of rules, was significantly less in the Robot Domain, but not in the TOH.

Thus, as with other experimental results discussed above, the difference between rule-building expense for non-abstract best-path search vs. abstract search increases as the complexity of subgoal search in the domain increases. We expect that had non-abstract best-path data been available for the 3- and 4-goal-conjunct tasks in the Robot Domain, the difference would have continued to increase for these tasks.

6.6.2 Plan Transfer

Because abstract rules are more general, they have the potential to apply in a wider range of situations than their non-abstract counterparts. An example comparing the generality of an abstract and non-abstract rule was given in Section 4.3.1. The abstract rules' greater generality over the non-abstract rules may not necessarily be beneficial to problem-solving. Thus, an analysis of the utility of abstract rules includes a consideration of the extent to which rules learned from solving one task or situation in a domain transfer to other tasks or situations in the domain, and the impact that the transfer rules have on solution quality and problem-solving efficiency. A comparison of these measurements across methods indicates the relative utility of abstract and non-abstract plans.

In this section, *plan transfer* experiments — done to assess the relative impact of abstract rules both within the same task and across tasks — are presented and discussed. The results suggest that in domains with variability in the situations and operators (i.e., in domains in which the tasks are not very similar and there are more than just a few instantiated operators), abstract rules transfer with greater utility than the non-abstract.

6.6.2.1 Within-Task Plan Transfer

As discussed previously, it was taken as given that Soar should learn while it problem-solved; therefore, there was no extensive comparison done between tasks run with and without learning. However, to observe the impact of within-task transfer of abstract rules on solution quality, we ran tests of SPATULA for a set of 6 3-goal-conjunct tasks in the Robot Domain with learning turned off. For these tasks, there was a only a small difference in the solution of one of the tasks, producing an average percent solution difference of .7% (not statistically significant) as compared to the solutions produced with learning. However, the tasks using SPATULA without learning took many more steps to complete, with an average 40% increase in problem-solving steps over the tasks using SPATULA with learning.

These experiments suggest that in the Robot Domain, there was very little overgeneral within-task transfer of abstract plans. That is, the abstract rules did not cause significant overgeneral transfer within the same task to situations for which they were not learned, since there was only minimal degradation of the solutions which were produced by using abstraction without learning. In the TOH, of course, such tests would provide no information about overgeneral transfer, since all solutions are already optimal.

6.6.2.2 Across-Task Plan Transfer

Experiments were also performed to test abstract plan transfer from one task to another. Across-task plan transfer was tested in the 2-goal-conjunct set of tasks from the Robot Domain, and in the TOH. (In the TOH, as with previous tests, the extended-plan-use method increment was utilized; its use is further discussed in Section 7.3.3). The purpose of these tests was both to compare the amount of transfer occurring in abstract and non-abstract search, and to observe the impact of the transfer upon solution quality and problem-solving efficiency. The transfer tests were made with rules produced from both the abstract and non-abstract best-path problem-solving methods. For purposes of comparison, we were restricted to testing tasks for which the non-abstract method was able to finish. In these tests, transfer of abstract plans was

tested on tasks run with abstraction, and transfer of non-abstract plans was tested on tasks run without abstraction[11]. Although the absolute change in problem-solving steps, etc., is not directly comparable across methods, a comparison was done of the effects of transfer on the tasks relative to their performance without the additional plans.

Two opposing factors can contribute to the transfer results. The non-abstract best-path plans, since they are more detailed, can provide information about a greater number of previously-encountered situations. This means that in new, relevantly similar situations, less search may be necessary when using the non-abstract plan as compared to the abstract. However, the abstract plans are more general than their non-abstract counterparts. Therefore, they may transfer to new situations where the non-abstract plans would not. Thus, the relative utility of abstract and non-abstract plan transfer depends upon the similarity of the new tasks to previous ones, as well as the level of generality of the abstract plans.

In the tests described here, we used a "batch" approach to testing transfer, in which, for each test of transfer, rules from the other tasks in the set of transfer experiments were provided together to the problem-solver when the test task was run. For example, in the TOH, Task 1 was run with rules from tasks 2,3, and 4 loaded in, Task 2 was run with rules from tasks 1,3, and 4 loaded in, etc. In the Robot Domain, a randomly selected set of 8 2-goal-conjunct tasks was tested; thus, for each of the tasks, plans from the other 7 tasks were used. As a topic for future work, it would also be informative to examine the interactions which occur when rules are accumulated incrementally.

Tower of Hanoi. For the TOH, the task solutions remained optimal when using both abstract and non-abstract rules from other tasks. This is an interesting result with respect to the learned abstract rules, since it means that the optimal abstract

[11] Abstract and first-path rule transfer were not compared, since the type of rule coverage is not the same. In addition, it did not make sense to test abstract rules with tasks run non-abstractly, or vice versa. If non-abstract rules are used during abstract problem-solving, they may not transfer to abstract states (since the abstract states may contain inconsistent or missing information). If abstract rules are used with non-abstract problem-solving, then the abstract rules will only transfer in the top-level execution space (since lookahead search will not be abstract).

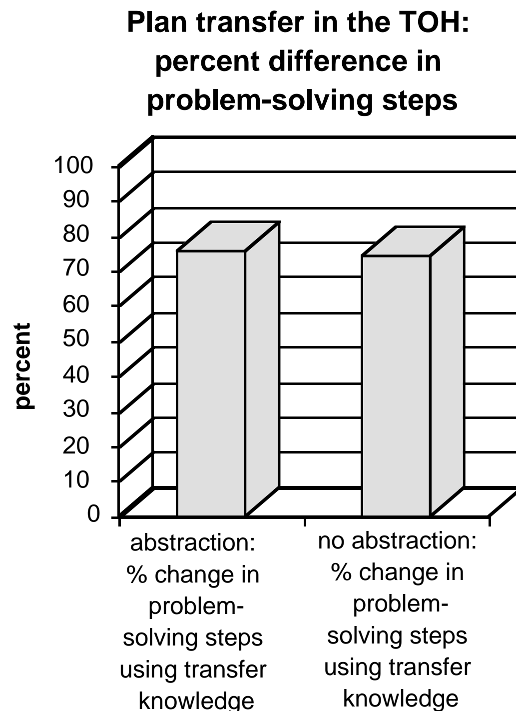


Figure 6.11: Summary of plan transfer results in the Tower of Hanoi.

strategy learned for one task is equally accurate for tasks using the same number of disks.

However, in the TOH domain, there was not much difference between transfer effects for abstract and non-abstract plans. Figure 6.11 shows the average difference in number of problem-solving steps between tasks run with and without the rules from other tasks. Here, both abstract and non-abstract transfer rules provided a significant savings. The difference between abstract and non-abstract problem steps using the additional rules (shown as a percentage of non-abstract steps) was not statistically significant (though abstract search took slightly fewer steps). The reason for this is that in the TOH, the tasks are all very similar to each other; though they have different initial states, they all have the same goal, and tend to share the same subgoals. With these characteristics, non-abstract subplans learned during one task were often applicable to another. Thus, the more comprehensive plan coverage provided by the non-abstract plans — containing a larger total number of rules —

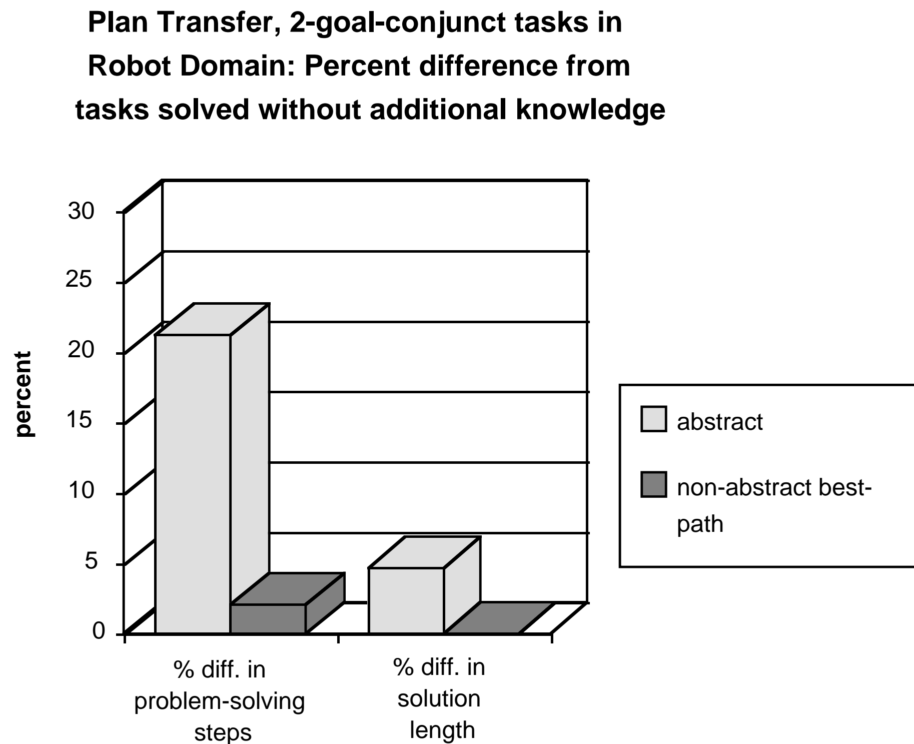


Figure 6.12: Summary of plan transfer results in the Robot Domain, shown as average percent difference from the results obtained without use of the rules from other tasks (i.e., $\frac{\text{non-transfer}-\text{transfer}}{\text{non-transfer}}$).

paid off on average.

Robot Domain. In contrast to the TOH, in the Robot Domain the abstract plans transferred to a much greater extent than did the non-abstract plans. In the Robot Domain, there was more task variability than in the TOH — the task goals were different, and it was not as likely that one task had a great deal in common with another. Here, the non-abstract plans proved too task-specific to provide more than minimal transfer.

Figure 6.12 summarizes the result of the transfer tests on solution quality and problem-solving steps in the Robot Domain. The comparisons are shown as percentage difference from the results obtained without using the additional rules. Of the 8 non-abstract tasks tested, transfer of other non-abstract plans impacted only one.

Overall, the number of problem-solving steps decreased by an average of only 2%, with no change in solution length. For the other non-abstract tasks, there was *no* impact on the number of problem-solving steps or on solutions. The non-abstract rule from the example in Section 4.3 illustrates the reasons for the minimal transfer — the rules were too specific to apply in situations in which, e.g., room connectivity was slightly different.

In contrast, the abstract plans were general enough to provide significant rule transfer from one task to another, while using fewer total rules, for an average decrease of 21% in problem-solving steps. It is interesting to note that though the abstract plans produced a significant average decrease in problem-solving steps, two tasks required more steps using the additional rules than did the original tasks. In one case, the increase was caused by the occurrence of an operator conflict (from conflicting search control), which then required additional problem-solving to resolve. In the other case, the additional plans led the problem-solver down a different solution path than in the original task, one which required more problem solving.

In the abstract transfer tests, the solution quality degraded in only one of the tasks, as compared to the solution quality without the use of the additional plans. The difference was not statistically significant. The solution degradation occurred in one of the same tasks which took longer to solve using the additional plans. For this task, transfer was overgeneral — the different path suggested by the additional plans turned out to be worse.

6.6.2.3 Discussion

The transfer test results suggest that unless a domain's tasks are very similar, much more effective transfer will occur with abstract than non-abstract plans. In addition, they suggest that the solutions generated using abstract plan transfer will be of good quality and require fewer problem-solving steps than the tasks without the additional plans.

6.7 Summary

Iterative abstraction and assumption counting proved useful in all domains tested. In all domains, SPATULA produced good solutions, significantly better than those produced by first-path search — in the TOH, the solutions were in fact always *optimal*. The use of SPATULA significantly reduced problem-solving time over that required to find an optimal solution. The relative efficiency advantages of abstraction over best-path non-abstract search increased as the complexity of the operator subgoaling required in the domain increased. The experiments also suggested that the weaker the previously existing domain search control, the greater the relative payoff in search reduction obtained from abstraction.

When compared with first-path search, the results showed that in domains where little search was required to find *some* solution, first-path search produced solutions more efficiently (though the solutions were not as good). However, in domains for which there was a significant amount of search, and in which it was more difficult to recover from a bad search path, abstraction performed better both in terms of efficiency and solution quality. It remained tractable when first-path search did not; and the abstract solutions deteriorated much less sharply than did those of the first-path tasks which did manage to complete. These results suggest that as the complexity of a domain or task increases, the payoff provided by searching for a good solution (rather than any solution) increases as well. Abstraction provides one method to increase the tractability of this search for a good solution. This general effect has been observed elsewhere as well [Knoblock, 1991].

The use of SPATULA produced rules which were easier to learn than non-abstract rules, and transferred to a wider range of situations than their non-abstract counterparts while continuing to produce good solutions. (Although the large amount of recursive similarity in the TOH produced good plan transfer both with and without abstraction, this was not the case in the Robot Domain, with greater variability in both tasks and subgoals.)

In the next chapter, we continue discussion of the experimental results.

Chapter 7

Further Experimental Results

7.1 Introduction

Chapter 6 described the results relevant to the goals stated in the development of the abstraction method. In this chapter, we describe the unexpected results of the experiments, both beneficial and problematic. A section is devoted to each of the following topics:

- A qualitative analysis of iterative abstraction and assumption counting, including the synergistic interaction between the two method increments. The novel problem-solving strategy produced in the TOH domain, as a result of this interaction, is discussed.
- Frequency of abstract rule use. Because the abstract rules are more general, more can fire per step than non-abstract, and this can cause a problem-solving slowdown.
- Detection of rule overgenerality. In our experiments, the wider the range of previously-learned abstract plans available, the greater the detection of plan overgenerality via the generation and resolution of plan conflicts.
- The extended-plan-use method increment. Results for the method increment are discussed, including results for two variations, and issues with its current

implementation.

- The efficiency-driven method increments. Results are discussed; the utility of these method increments proved to be domain-dependent.
- The relationship between goal ordering issues and the method increments.

We discuss each topic in turn, then summarize.

7.2 Qualitative Analysis: Iterative Abstraction and Assumption Counting

The numerical analyses in the previous chapter described only part of the impact of abstraction in the test domains. It is also informative to examine the observed qualitative effects of iterative abstraction and assumption counting in each domain.

In all domains, assumption counting improved the efficiency of iterative abstraction, by providing additional information which enabled the problem solver to make a decision in fewer iterations than would otherwise have been possible. In addition, it at times improved the quality of the decisions that would have been made using iterative abstraction alone.

Conversely, iterative abstraction provided the problem solver with additional information about a decision when assumption counting alone was not sufficient to discriminate among the options. As discussed in Section 5.3, the iteration process also let the problem solver discover, for each iteration level, any preconditions which were *critical* (in the ABStrips sense) at the previous level — that is, any preconditions for which no method of achievement could be found. Thus, the two method increments proved to interact helpfully with each other, improving the performance which would have been obtained with either alone.

7.2.1 The Eight Puzzle

In the EP domain, search can be difficult due to the interaction of the tiles. However, the domain representation itself is simple, with one operator. This means that for

abstraction methods such as [Knoblock, 1991] (see Chapter 8), which create abstract spaces based on the interactions between operators, this domain does not produce any abstractions. In contrast, abstractions produced by SPATULA's method increments proved useful in this domain.

Assumption counting proved to be so effective that, of the 10 tasks tested, using both assumption counting and iterative abstraction, the problem solver only iterated on abstraction level for one task. For all other tasks, the use of assumption counting was sufficient to allow the system to distinguish between its options at the highest level of abstraction (that is, with all preconditions abstracted away).

The reason for this effectiveness stemmed from characteristics of the Eight-Puzzle tasks and domain. Recall that the problem representation was such that an operator was generated for each movement of a tile to an adjacent square, with the precondition that the adjacent square be clear. Given this representation, it was often possible to find some solution to the tasks for which very few such preconditions were unmet (i.e., for which very little operator subgoaling was necessary). That is, if the moves were ordered properly, then tiles could be moved incrementally along the x and y directions towards their goal locations without much need to move other tiles out of the way first[1]. Therefore, there was usually one abstract plan with just one or two assumptions, and this would be preferred over the others. In all but one of the tasks tested, choosing such a path resulted in a good solution.

It is worth noting that the search control used by the EP, in which operators with no unmet preconditions are preferred (or even a variant in which the operator with the fewest unmet preconditions is preferred), would not have been useful in the other two domains tested. In general, more useful information is provided by finding the search path with the fewest *total* assumptions, since this provides information about interactions between goal conjuncts. Thus, such search knowledge as used by the EP is only useful if there tend to be paths with very few total assumptions. Otherwise, a "horizon effect" will tend to occur.

The use of iterative abstraction alone (without assumption counting) was tested

[1] This domain characteristic may have been due partly to the relative shortness of the tasks, which had optimal paths of ten moves long.

in the EP domain as well[2]. Iterative abstraction alone also proved effective in the EP domain. The reason for its effectiveness was again the fact that it tended to be possible to construct some solution path which used very little operator subgoaling. At the first iteration level of abstract search in the EP, the number of steps required to reach the goal was always the same for each path considered. With further iterations (as further effects of clearing destination squares were calculated), some paths would require more steps than others to reach the goal. When the problem solver found an iteration level where there was a difference in the number of steps, there tended to be an actual (ground-level) difference between the choices as well. This is because after some small amount of subgoaling, the solutions for the good choices tended to fall into place with no more subgoaling necessary, whereas the bad paths tended to cause more and more displacement at each iteration, and thus longer paths. The example of Figure 5.3.2.1 illustrated this effect.

7.2.2 Tower of Hanoi and Robot Domain

In the TOH and the Robot Domain, neither assumption counting nor iterative abstraction by itself was sufficient to make abstraction as useful as if the two method increments were combined. This is because these domains were represented such that ground-level solutions could not be reached without some amount of operator subgoaling. Thus, in these domains, assumption counting by itself was rarely sufficient to provide much discrimination among control decision options– all search paths at the most abstract level tended to make about the same number of assumptions. However, used in conjunction with iterative abstraction, the problem solver was able to iterate until its searches were at a level of detail such that assumption counting was useful.

Conversely, the use of assumption counting with iterative abstraction allowed the problem solver to make decisions at an earlier abstraction iteration than the problem solver would otherwise have required to distinguish between its options, thus increasing the efficiency of iterative abstraction. In addition, assumption counting at times allowed the problem solver to *improve* on the decisions that would have been

[2] These tests used an earlier version of Soar as well as a slightly different version of the domain and search control, and thus are not directly comparable with the other EP results discussed here.

made using iterative abstraction alone, since a path with a shorter number of abstract steps (i.e., with a better domain evaluation) is not necessarily the easiest path. Thus, the two method increments worked synergistically together. Below, we discuss their effects in each domain in turn.

7.2.2.1 Tower of Hanoi

The TOH domain was represented in our tests using one "move-disk" operator, which was instantiated with a particular disk and peg as appropriate. Using this representation, SPATULA produces abstractions which generate optimal solutions — that is, the use of abstraction in the TOH domain allowed the problem solver to reach the goal state without any false moves.

The synergistic effect of iterative abstraction and assumption counting on solution quality was most marked in the TOH domain. In this domain, the use of assumption counting by itself did not produce any discrimination of options, and thus choices would be as random as if assumption counting had not been used at all. Iterative abstraction alone produced discrimination of options (at the second abstraction iteration), but caused the problem solver to make the *wrong* choices (since shorter abstraction solutions were not in actuality easier to expand) and to produce a non-optimal solution.

Using both assumption counting and iterative abstraction, the problem solver was able to find optimal solutions. The use of both method increments, and the lexical ordering of the evaluation function — which placed assumption counting over domain evaluations — proved crucial to this result. In addition, tests were run in the TOH domain *after* the method increments were developed; the TOH did not drive their development in any way).

The optimal strategy produced for the TOH is rather novel, and stems from one of SPATULA's problem-solving biases described in Section 5.8 — easier subgoals appear preferable to work on first, all else being equal. With this bias, the problem solver, rather than adopting a more common completely recursive approach to setting subgoals, selects the *easiest* of the subgoals along its optimal solution path.

This strategy is illustrated in Figure 7.1 for the canonical 4-disk task (the task with

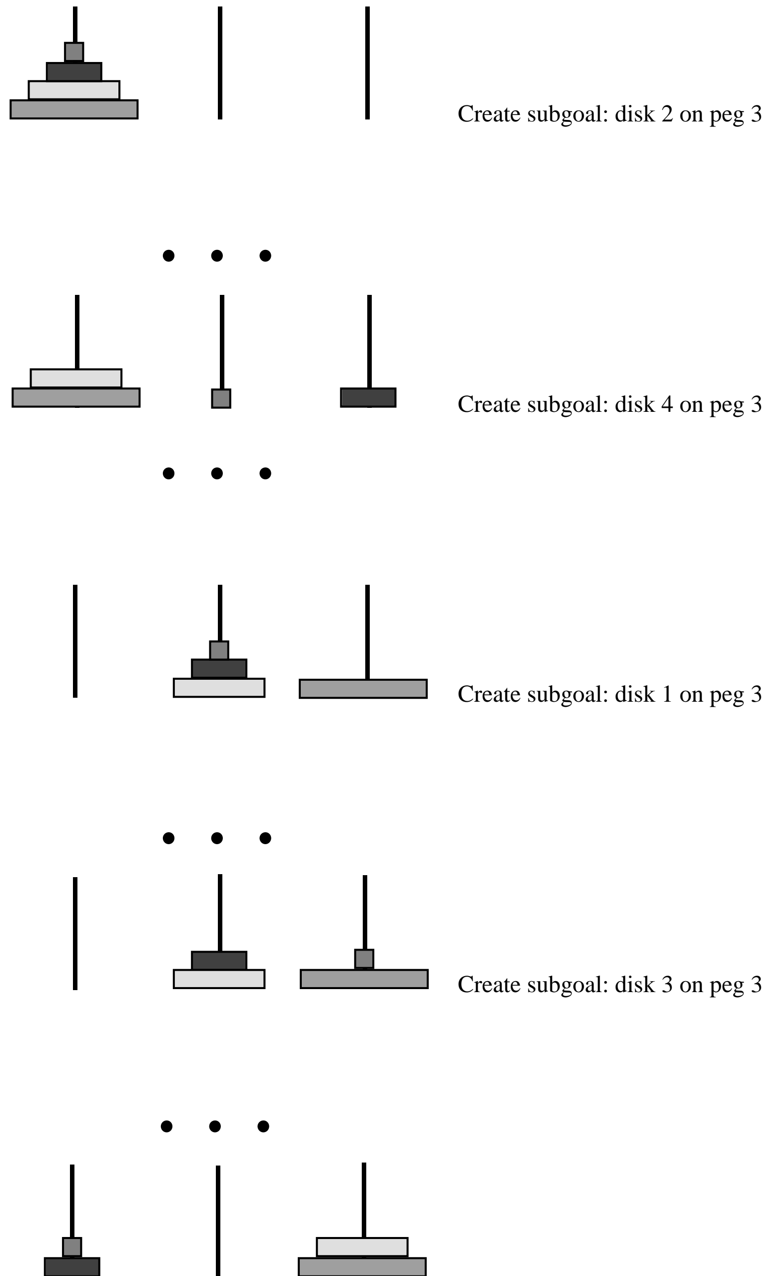


Figure 7.1: Easiest-subgoal-first strategy employed with SPATULA in the TOH.

the initial state of all disks on the first peg, as shown at the top of the figure and a goal to place all disks on peg 3). The figure shows the first part of the series of task subgoals generated by the problem-solver when using SPATULA. (In the figure, intermediate lower-level subgoals are omitted). The system first chooses a task subgoal of moving disk 2 to the goal peg. (Its abstract evaluations have lead it to prefer disk 2 over disk 1). After that subgoal is accomplished, the system chooses the goal of moving disk 4. With disk 4 in place, it next chooses to move disk 1 to peg 3, then disk 3, and so on. This strategy is employed by the system because of SPATULA's easiest-subgoals-first bias.

In contrast, a fully recursive approach would chose an initial task subgoal of moving disk 4 to the goal peg, and when that was completed, a task subgoal to move disk 3 to peg 3. In the process of moving disk 4 from the initial state, an optimal set of moves requires an intermediate state in which disk 2 is placed on the goal peg as shown in the figure. (Note that an initial goal to move disk 1 or 3 to the goal peg would not be on the optimal path). Similarly, once disk 4 is on the goal peg, an optimal set of moves for the goal to move disk 3 to peg 3 requires an intermediate state in which disk 1 is placed on the goal peg. However, the fully recursive strategy differs from SPATULA's in that with the fully recursive strategy, these intermediate moves are not explicitly cast as top-level task goals. Using SPATULA, the system constructs a strategy in which the top-level task subgoals are on average simpler than those generated using the fully recursive approach, yet the solution path is still optimal.

In the TOH, the task solutions also remained optimal when using abstract rules from other tasks. This is an equally important result with respect to abstract plan use, since it means that the abstract strategy learned for one task is equally accurate for other tasks using the same number of disks. Thus, the abstract plans produced using SPATULA for the TOH domain had no overgenerality.

Although the combination of the two method increments produced solutions of optimal quality in the TOH, the method was not relatively efficient — that is, it did not require fewer steps than solving the problem non-abstractly — unless the iterative abstraction began its abstract searches at the *second* iteration level by always solving for one level of preconditions rather than initially ignoring all unmet preconditions.

This occurred because the TOH domain and search control knowledge is such that the search depth for operator subgoaling is not very deep, and thus the abstract searches were not that much shallower than the non-abstract ones. In this domain, because the highest level of abstraction provided no discrimination among control decision options, the overhead of the iteration process made the two abstract searches (at successive abstraction levels) more expensive together than one non-abstract search[3].

7.2.2.2 Robot Domain

In the Robot Domain, abstraction used with iterative abstraction and assumption counting generated good (sometimes optimal) solutions, while providing a more significant efficiency gain with respect to the non-abstract search than seen in the TOH domain– the search to achieve a subgoal could potentially get much deeper in the Robot Domain, and thus the relative savings were increased as well. Using assumption counting and iterative abstraction, the problem solver was usually able to make decisions at a relatively high level of abstraction (that is, after just a few iterations).

In this domain (as with the TOH domain), the use of either assumption counting or iterative abstraction alone would not have produced results as useful as those produced by combining the methods. Assumption counting by itself rarely enabled sufficient discrimination to make useful choices among the options. Iterative abstraction provided the necessary detail, and allowed discovery of critical preconditions. Conversely, experiments with a subset of the tasks indicated that the use of iterative abstraction alone required more problem-solving steps to generate a solution (since without assumption counting, many more iterations were required to make a decision and/or there were more options per iteration), and were not always as good. The example of Section 5.4 illustrated the way in which iterative abstraction and assumption counting worked synergistically in this domain; the task in the example was one of the 4-goal-conjunct tasks used in the empirical tests.

Although the method increments worked well in the Robot Domain, more mistakes

[3]The analysis of Section 5.9 shows that in general, abstract searches should be cheaper than non-abstract regardless of the number of iterations required. However, analogously to iterative deepening, this not true for a very small number of assumptions.

— with respect to non-optimal solution paths — were made with abstraction in this domain than in the other two. The experiments in the different domains suggest that the greater the variability in the domain operators, the greater the improvement in solution quality produced by being more conservative in the interpretation of the information provided by the method increments. For example, in the Robot Domain, the meta-evaluation function (which combined the assumption count with the domain evaluation) and the iteration halting conditions were probably too simplistic. A large subset of the problem-solver's mistakes could have been avoided if it had not chosen one option over another on the basis of just *one* assumption, but rather had taken a more conservative approach, and required a difference of either more than one assumption, or a difference in domain evaluation as well as number of assumptions. The impact that such changes would have had on problem-solving time is not obvious; although they would cause more work during abstract planning, this might have been compensated by increased ease of solution refinement, particularly with the complex room layout. The problem-solving impact of tuning the method increment parameters is an important area for future work, and may drive the development of effective but less sensitive domain-independent method increment parameters. Section 5.5 discussed this issue, and other possibilities will be further explored in Chapter 9.

7.3 Learning and Plan Utility

In Chapter 6, we showed that in our experimental domains, abstract plans were learned more easily, and — except in domains of great regularity — produced better transfer to new situations while continuing to generate useful solutions. In this section, we discuss some of the other issues that arose, both beneficial and problematic, in the use of abstract plans.

7.3.1 The Generality of Abstract Plans and Problem-Solving Time

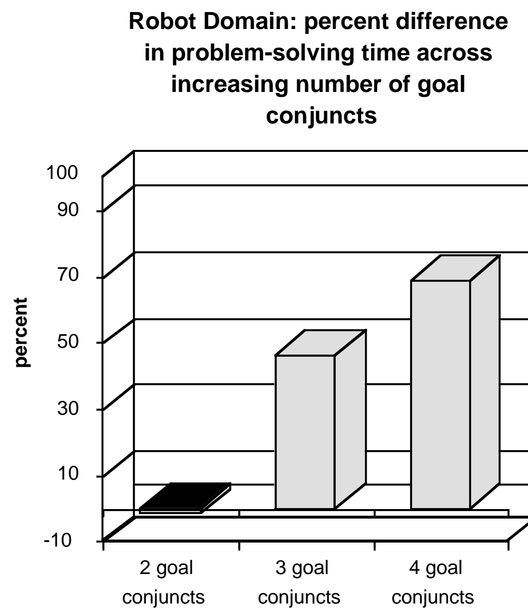


Figure 7.2: Relative problem-solving times of abstract and first-path search in the Robot Domain (i.e., $\frac{\text{abstract} - \text{first-path}}{\text{abstract}}$).

Because learned abstract plans are more general than non-abstract plans, they tend to match in more situations and fire more often. In our experiments, this was particularly the case when compared to rules learned during non-abstract first-path search, since the first-path rules are learned for just one search path rather than many, and thus are less likely to be applicable in new situations. As a result of the larger number of average rule-firings per step in abstract search, the abstract search could require more time per step than the first-path. As the number of goal conjuncts increased, the average relative difference in time per step increased as well. This is because an increased number of goal conjuncts provide an increased opportunity for the application of comparative search control rules. Figure 7.2 shows average relative difference in time to solve abstract and first-path tasks. Compare this figure with the last set of columns in Figure 6.8, which shows relative differences between task problem-solving *steps*. For the 3- and 4-goal-conjunct tasks, the percent difference in task time between the two methods is slightly larger than the difference in task steps (thus indicating that time per abstract step is slightly greater as well); and as shown by the two figures, the disparity increases with number of goal conjuncts.

However, run statistics on tokens per rule, given in Appendix E, showed that the abstract rules were *not* more expensive to match on a per-rule basis. Thus, the slow-down may be primarily an implementation-specific plan utility issue. With a distributed implementation of Soar, in which each applicable rule was fired in parallel, we would expect this problem to be alleviated, as long as the number of rules fired at each *elaboration* was bounded. This is an area for further investigation. Note that the issue is not specific to the rules produced by SPATULA, but pertains to any rules of increased generality, regardless of origin.

7.3.2 Plan Transfer: Detection and Correction of Overgeneral Plans

As described in Section 6.6.2, the plan transfer experiments showed that abstract plans from previously-encountered tasks allowed significantly more transfer than the non-abstract plans (unless the tasks were very similar to one another), with only small

degradation in solution quality over that which would have been produced without the additional plans.

Examination of the transfer runs determined that the good solution quality was due in part to an interesting interaction between the different abstract plans, which curtailed their potential overgenerality. Operator *conflicts* were generated several times when the additional plans contradicted each other, and the system had to deliberately resolve the conflict by re-evaluating the operators in question. Thus, the conflicting preferences were a flag that over-general rules were being applied and that further evaluation was required. Only once did this additional problem-solving cause the transfer task to take longer than the original; in the other cases, the overall transfer of information more than made up for the extra effort at the conflict.

The experiments suggest that the greater the range of experience applied to a new task, the greater the chance to detect that a rule is too general. This result has implications not only for abstract planning, but for the use of any type of potentially over-general rule, whether learned or pre-provided. It suggests that the wider the range of experience available to a problem-solver, the greater the opportunities for the system to detect — via inconsistencies — that its knowledge is not sufficiently accurate in a given situation. Additional work is required to explore and confirm this hypothesis.

7.3.3 Extended-Plan-Use Method Increment

The investigation of abstract plan utility included an exploration of the different ways of using learned abstract plans, by employing different variations of the extended-plan-use method increment. In this section, we describe the results of tests performed with this method increment.

As described in Section 5.6, the extended-plan-use method increment provides the system with a means to reason explicitly about the extent to which it will use abstract plans learned during lookahead *sub*-searches. It also provides a heuristic, based on the motivation behind iterative abstraction, for guessing which rules from these sub-searches will be most useful, and will use only these more useful plan fragments as the plan is refined (the conservative version of extended plan use). It is not a priori

evident that the problem-solver will always benefit from the use of either version of the extended-plan-use method increment. It may sometimes be more useful to replan in the abstract space using new information rather than using all of a previously-generated plan.

The extended-plan-use method increment was tested in the TOH and Robot Domains. The bulk of the tests used only the least conservative version of the method. That is, all of the plans from the most detailed sub-searches were utilized. In the Robot Domain, a small set of additional tests was also performed using the conservative version of the method increment, in which the only rules which transferred from the sub-searches to the execution space were those for which a discrimination between options was made. Further exploration of this spectrum of plan usage is an important area for future work.

7.3.3.1 Tower of Hanoi

Because SPATULA always produced correct plans at the second iteration level in the TOH, the full use of the plans produced from sub-search at that level had no detrimental effect on solution quality, and their use always improved problem-solving efficiency. In fact, when the extended-plan-use method increment was *not* utilized, the abstract searches took longer than the non-abstract. This was because the relatively short operator subgoals did not afford much savings from abstraction, and the regularity of the domain allowed a large amount of non-abstract plan transfer from one part of the task to another.

7.3.3.2 Robot Domain

In contrast to the TOH, in the Robot Domain the subgoals could become much deeper and thus abstraction provided much more of relative savings in problem-solving effort regardless of whether or not full use was made of the abstract plans. The Robot Domain experiments described in the previous chapter did not use the extended-plan-use method increment, and thus took the most conservative approach in which the system did no reasoning about the iteration levels of its plans; only those plans learned for the top-level control decision could transfer to situations independent of

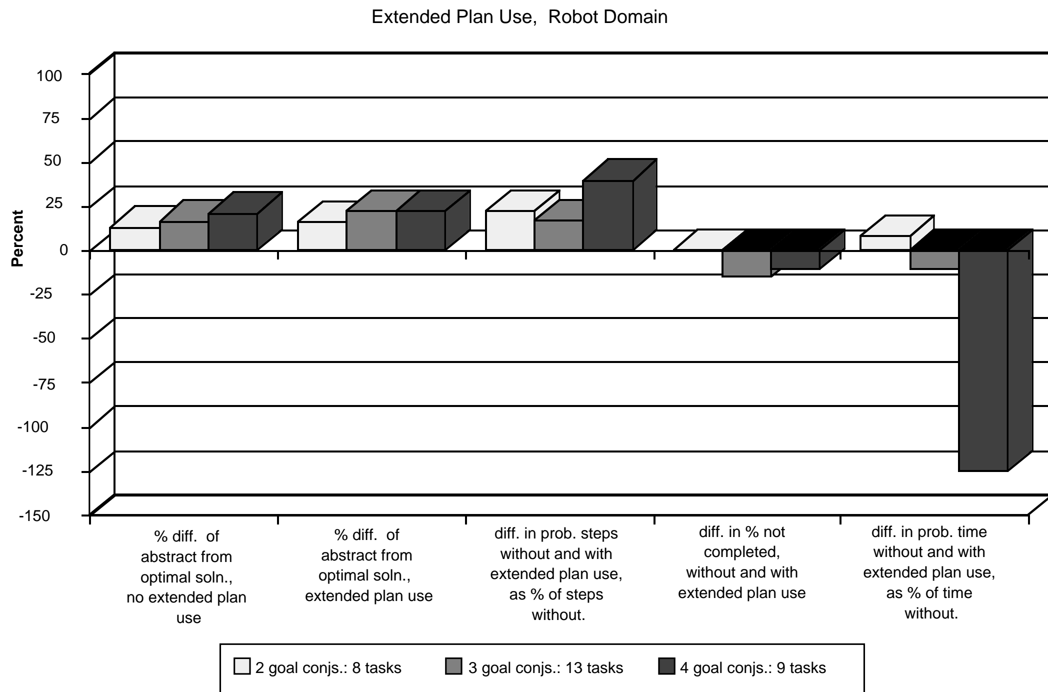


Figure 7.3: Summary of extended plan use results in Robot Domain.

their iteration level. In addition, the abstract search control in the Robot Domain was not always correct, so it would be expected that more extended use of the abstract plans — instead of re-planning — could produce some degradation in solution quality.

Thus, it was less evident how beneficial the extended plan use method would prove to be in the Robot Domain. Experiments were performed to observe both the resultant problem-solving savings and the change in solution quality when the method increment was used. These extended-plan-use tests were made on a randomly selected subset of the Robot Domain tasks from each goal conjunct group, and used the least conservative version of the method increment.

Figure 7.3 summarizes these results. Only little solution degradation occurred with these tasks, as may be seen by comparing the average percent difference from the optimal solution for abstraction with and without extended plan use. The difference in solutions with and without the method increment was not statistically

significant. Most of the new mistakes made using the extended-plan-use method increment occurred because of the *indifferent* preferences in the extended plans, which proved too abstract in even the situations for which they were learned. This means that the conservative version of the extended plan use method would have improved the solution quality results. For two of the tasks, the solutions generated with extended plan use produced solutions that were better than those without. This result, which may appear non-intuitive, points out that the abstract plans are not completely correct. Thus, on occasion, a less informed choice actually causes the better of the options to be selected.

With the extended-plan-use method increment, the use of the additional plans means that the problem solver has more knowledge about how to select its task operators. In general, one would expect that problem-solving using the additional plans will take fewer steps. As shown by the difference in problem steps, this proved to be the case. However, for a few of the tasks, use of the extended plans actually added to the number of steps required to solve the task. This occurred in our tests for two reasons. First, the additional plans, since they are more abstract, could transfer overgenerally and cause an operator conflict which did not occur with the more conservative plan usage. Further problem-solving would then be required to resolve the conflict. Second, the solution path suggested by the additional plans could take longer to refine (e.g., if a sub-operator-tie was particularly hard to resolve). The 2-goal and 4-goal differences in problem-solving steps were statistically significant; the 3-goal average difference was reduced and not statistically significant due to a large increase in problem-solving steps observed for one task.

Some of the tasks solved with the extended-plan-use method increment (EPU) took much longer to run than without. This is shown by the negative time percentages in the figure, which indicate average $\frac{\text{(time without EPU)}-\text{(time with)}}{\text{(time without)}}$. The cause of this slow-down was that more rules were firing per step, although the total rule matching costs were not more expensive, nor were individual rules more expensive. The reason for this was not directly related to increased generalization due to abstraction. Rather, it was because the system was required to fire applicable rules from all iteration levels in a new situation, and then retract the ones which

were not of the highest applicable level. The effect increased with increased task difficulty. Some of the extended-plan-use tasks were terminated if they were taking an excessive amount of time per step, and judged intractable even if they weren't memory-intensive. This is reflected by the intractability percentages in Figure 7.3, which indicate average $\frac{\text{(completed without EPU)–(completed with)}}{\text{(completed without)}}$.

As discussed in Section 7.3.1, a slow-down due to rule firing may be primarily an implementation issue, which would be avoided with a distributed implementation of Soar. However, since there may be an arbitrary number of abstraction iteration levels, it is not clear that the number of rules fired and then retracted with the extended-plan-use method increment can be expected to remain bounded. It may be the case that the idea behind the method increment — that of selecting and using the most detailed of the available abstract sub-search plans — will need to be re-implemented using a different approach if extended plan use is to have practical utility.

7.3.3.3 Extended Plan Use, Conservative Version

As discussed above, for some of the tasks tested with the more liberal version of the extended-plan-use method increment, the full use of abstract sub-search plans proved too abstract to produce good solutions. Of these tasks, 3 were used to test the use of the conservative version of the method increment. Use of this version of the method increment only allows execution-level transfer of those abstract plans formed when control decision options are *discriminable* (or when no assumptions are made).

For the tasks tested with the conservative version of extended plan use, quality degraded by only 3% as compared to the solutions generated without any use of sub-search abstract plans, and improved by 23% over those generated using the more liberal version. At the same time, the average problem-solving steps were only 77% of those required without the extended-plan-use method increment.

7.3.3.4 The Spectrum of Abstract Plan Usage

The results of the previous section are still preliminary, but indicate that the heuristic provided by the conservative version of the extended-plan-use method increment is a useful one. Although the less conservative use of all sub-search plans produced

good solutions in both domains tested, the experiments suggest that the full use of abstract sub-search plans may in general prove to be too abstract and expensive. At the other end of the spectrum, tasks solved without any version of the extended-plan-use method increment (thus only using the problem-solver's top-level abstract plans and allowing the system to reason about any new information after each operator execution), generate the most accurate decisions. However, the empirical results suggest that this approach is too conservative, and that judicious use of the lower-level plans can improve efficiency with very little solution degradation.

Therefore, on the spectrum of abstract plan usage, use of the more conservative version of the extended-plan-use method increment may provide the best balance between solution quality and planning effort. This issue remains an important area for future research, since the relative utility of each point on the plan use spectrum may be impacted by the characteristics of a given domain.

7.4 Impact of Efficiency-Driven Method Increments

Two additional method increments were tested in the experimental domains as well; *goal achievement iteration*, and the *abstraction-gradient* method increment. As discussed in Chapter 6, these methods were used with the set of Robot Domain tests presented in that chapter. In this section, we evaluate the impact of these method increments on our experimental domains.

As described in Section 5.7, the two method increments both use greedy, or localized, approaches to making an evaluation — they examine most closely the more immediate effects caused by the operator being evaluated. They share in common the idea that it is possible to estimate the effects of applying a particular operator (and thus to make a control decision) without determining the effects of that operator on the entire task, and that this can be done to keep the cost of successive abstraction iterations from escalating as sharply. Unlike assumption counting and iterative abstraction, these method increments don't have the potential to provide any new

information to the abstract search. Instead, they provide heuristics about how to reduce the information used during abstract search, to increase its efficiency while still allowing relevant decisions.

Goal achievement iteration decreases the number of goal conjuncts solved for at each abstraction iteration; that is, it decreases the number of conjuncts as the level of detail increases. The second method increment, abstraction-gradient, decreases the level of detail as the problem-solving progresses, thus focusing the most detailed analysis on the beginning of the network of effects caused by applying an operator.

One would expect the utility of both of these method increments to be impacted by both the degree and depth of the interdependencies between the goal conjuncts. The more tightly interacting the goals, the more desirable one would expect it to be to examine much of the problem in lookahead before making a decision about what to do next. The deeper the interactions, the smaller the likelihood that they will be discovered at a high level of abstraction, or when the detail of lookahead search tapers off. Experiments were performed in the Robot Domain and Tower of Hanoi domains to test the two efficiency-driven method increments in conjunction with iterative abstraction and assumption counting; these hypotheses were borne out in our experiments. As will be discussed below, the method increments proved useful in the Robot Domain, but notably non-useful in the TOH domain. Thus, their utility is domain-dependent.

7.4.1 Tower of Hanoi

With the Tower of Hanoi, use of either of the two efficiency-driven method increments (alone or together) proved detrimental to the optimality of the solutions produced using abstraction. This is because the interactions between the subgoals in the TOH not only are very tight, they can not in general be detected at the first level of abstraction.

If the problem solver utilizes the abstraction-gradient method increment, then it is not able to examine the effects of achieving the task subgoals in sufficient detail to make good decisions. Thus, although total problem-solving time is decreased, the problem solver does not choose optimal moves. If the problem solver uses goal

achievement iteration in the TOH domain, then when it iteratively decreases the level of abstraction (thus allowing detection of interactions between subgoals) it is no longer solving for all goal conjuncts. Since the goal conjuncts are tightly interactive, again this does not give the problem solver enough information to make a good decision, and again it chooses moves off the solution path (e.g., it will plot solutions which do not take the largest disk into account). This proved to be the case regardless of whether or not goal achievement iteration was used with abstraction. In addition, the search control rules learned as a result of employing goal achievement iteration proved to produce an unusually high number of search control conflicts, thus requiring extra problem-solving time to resolve the conflicts, and causing the search to take more time than if all goal conjuncts had been used to begin with.

7.4.2 Robot Domain

In contrast to the Tower of Hanoi, the efficiency-driven method increments provided a much more acceptable tradeoff between efficiency and accuracy in the Robot Domain. This was because the subgoals of the tasks tested were not as tightly interacting across all subgoals. That is, it was rare that all subgoals had large interactions with *all* other subgoals (even though it was often the case that all subgoals interacted with *some* other subgoals.) Therefore, it was possible for the problem-solver to find good (sometimes optimal) goal conjunct sequences while using these method increments, thus increasing search efficiency. Below, we discuss the results obtained with and without each method.

7.4.2.1 The Abstraction-Gradient Method Increment

In the Robot Domain, unlike the TOH, solution quality was not greatly degraded by increasing abstraction as a search progressed. While the goal conjuncts typically interacted with each other, they were not as tightly interacting as were the goals in the TOH. Recall the example of Figure 5.24, in which the problem-solver is able to reach a good decision by examining the search path for each of its options in the most detail at the beginning of the searches.

As discussed above, the majority of the tests in the Robot Domain were performed using the abstraction-gradient method increment. Thus, to assess the impact of this method increment on these results, we tested 15 tasks from the 3- and 4-goal-conjunct groups with and without the use of the method, and compared the results. Iterative abstraction and assumption counting were used in all cases, as was the goal achievement iteration method increment.

For one task, attempts both with and without the abstraction-gradient method increment were intractable. Of the 14 remaining tasks, three of the 4-goal-conjunct tasks were intractable without the use of the method increment. For those tasks which finished both with and without the method increment, abstraction-gradient decreased the number of problem-solving steps for all but two of the tasks, with a total average decrease of 13%. (For the two tasks in which problem-solving steps were actually increased with the method, one increase occurred because a choice on the solution path required many iterations to resolve. The other increase occurred because the more expensive solution produced by the method required enough additional execution-space steps to make up for the efficiency benefit). Solution quality was slightly but not significantly degraded with use of the abstraction-gradient method increment; its use produced an average 7% increase in solution length, as a percentage of the solution without abstraction-gradient. Thus, the method increment appeared to be a useful technique for the Robot Domain.

7.4.2.2 Goal Achievement Iteration

As discussed in Section 6.5, for the experiments run using goal achievement iteration in the Robot Domain, the initial number of goal conjuncts to solve for was set to three (one less than the highest number of goal conjuncts in the Robot Domain tasks tested), and decreased by one at each iteration. Thus, goal achievement iteration had very little impact on the 2-goal-conjunct tasks, and the greatest impact on the 4-goal-conjunct tasks.

As shown by the results in Section 6.5, for many of the tasks in the Robot Domain the problem solver did not require an initial total ordering of the goal conjuncts to produce a good solution. An incrementally constructed partial goal ordering provided

sufficient information much of the time. Recall the example of Section 5.4, in which it was not necessary to consider all total orderings of the goal conjuncts to determine which operators should be applied first; once a partial ordering has been constructed, any additional top-level operator ties generate a new search which reaches further to the goal and refines the ordering. In fact, an examination of the problem-solving traces in the Robot Domain showed that most mistakes were made because of undetected interactions between pairs of goal conjuncts (rather than a failure to consider a large enough number of goal conjuncts).

Thus, the goal achievement iteration method would be expected to impact Robot Domain tasks primarily in terms of problem-solving time. To assess this impact, we compared a small subset of the 3-goal and 4-goal-conjunct tasks solved with and without the method. The 2-goal tasks were not included in the comparative tests since the method increment rarely had an impact on them. There was also no need to perform comparative tests for *non-abstract* goal conjunct iteration (by terminating the non-abstract searches after 3 goal conjuncts had been achieved). Such truncated searches would only have affected the 4-goal-conjunct tasks. These 4-goal-conjunct tasks would have been more difficult than the non-abstract three-goal-conjunct tasks, and thus would have been intractable.

Four 3-goal-conjunct tasks were tested without the goal-achievement-iteration method increment. As hypothesized, consideration of all task goals during lookahead did not affect resolution of the control decision impasses; the same operators were still picked in all tasks — that is, the tests showed that solutions had not been degraded by use of the method increment. However, an interesting result was observed when the tasks were run using the extended plan use method increment. The sub-search plans learned without goal achievement iteration were less general, and caused fewer operator conflicts than did the plans learned using extended plan use, thus sometimes reducing total problem steps. This provided another indication that the full use of abstract sub-search rules (e.g., the less conservative version of extended plan use) produces plans that are too general. Three of the 4-goal-conjunct tasks were tested without goal achievement iteration. Two of these tasks proved to be too hard without the method increment (the system ran out of memory), thus suggesting that

use of the method increment improved tractability in the harder Robot Domain tasks. For the task which finished without use of the method increment, solution length was the same (though a different solution was selected).

Thus, these tests suggest that goal achievement iteration improved the overall tractability of the Robot Domain tasks with little to no solution degradation. However, the tests also suggest that there is an interesting interaction between efficiency of the method increment — in terms of problem-solving steps — and the extended plan use method increment, which requires further study. The intractability of the 4-goal-conjunct tasks without goal achievement iteration again illustrates that even with the large savings provided by SPATULA, the efficiency of the problem solving is still limited by the combinatorics, required by the problem-solving methods, to order the task's goal conjuncts. (The exact point at which this "wall" is reached is implementation- and task-dependent; 4 goals were not too difficult in the TOH). With the goal achievement iteration method, the goal ordering effort is reduced for each iteration as the detail increases. These tests therefore support the theoretical results of Section 5.9.

7.4.3 Discussion

Experiments indicate that the utility of the goal achievement iteration and abstraction-gradient method increments is dependent upon the degree and depth of goal conjunct interaction in a domain. In the Robot Domain, the methods in general increased problem-solving tractability with only little degradation of solution quality. However, they were not as successful with the recursive and tightly interacting TOH. Therefore, although they are general methods, they can not be unilaterally recommended without first determining their impact on a domain via preliminary testing or analysis. It may be the case that they will prove appropriate for many "every-day" tasks, in which only a small number of goal conjuncts interact at one time.

7.5 The Method Increments and Goal Conjunct Ordering

In our experiments, we contrasted abstract search (in which search options were compared), with non-abstract first-path search (in which an action was selected arbitrarily from the set proposed by the existing domain search control). When the number of task goal conjuncts became sufficiently large, the abstract search was constrained by the work required to order the goal conjuncts (or more exactly, the work required to order the operators that achieved the task goal conjuncts), and became intractable. With the non-abstract first-path search, no goal ordering effort was required, and thus this effect was not observed. (However, first-path intractability did increase with the harder tasks, particularly in the more complex domains, for another reason: the larger the task, the greater the chance for first-path search to select bad paths from which it was hard to recover).

The experiments did not test *abstract* first-path search, though this might seem a natural comparison with non-abstract first-path search. In fact, with SPATULA, an abstract first-path search will provide no more information than a search without any use of iterative abstraction and assumption counting. Consider that both iterative abstraction and assumption counting method increments get their leverage from *comparison* of the choices at a control decision, via lookahead search. With a strictly first-path search, such comparative information is not utilized to make decisions (the only information used to select an operator is that of the pre-existing search control).

So, with first-path search, iterative abstraction and assumption counting provide no new information about which operator to select. Thus, a first-path abstract search with the method increments will be only as useful as SPATULA's basic abstraction method with first-path search. Such a first-path search will only be useful if failures or goal clobberings are detectable at the problem solver's initial level of abstract search. The more abstract the initial search, the smaller the probability that such detection will be possible. (A parameter of the basic method can be the number of levels of preconditions for which the problem solver initially solves). In the experimental domains described here and with the initial level of abstraction set to abstract all

unmet preconditions, an abstract first-path search would rarely have provided useful information. However, this does not mean that domains do not exist in which abstract first-path search would have more utility.

With other abstraction methods in which the abstractions are produced prior to problem-solving, the abstractions may be better (in the sense of capturing more of the important aspects of the domain) than SPATULA's basic abstraction method. Recall that the purpose of SPATULA's method increments is in fact to dynamically search for and produce a useful level of abstraction, in those cases for which the system can not construct abstractions prior to problem-solving. This means that first-path searches using pre-determined abstractions may tend to provide more information about goal clobberings, etc., than will SPATULA's basic abstraction method.

This analysis does not imply that a full combinatorial exploration of all different operator orderings must be performed in order to use SPATULA's method increments, but that some comparative search must occur for the method increments to have an impact. The experimental results confirmed the theoretical results of Section 5.9, and suggest that for tasks which require the ordering of a large number of goal conjuncts, SPATULA — while it increases tractability — still needs to be used in conjunction with some other method of controlling search through the space of goal conjunct orderings, whether it be the two efficiency-driven method increments described here, or some other method for partitioning and/or search limitation.

7.6 Summary

In this chapter, we discussed the unexpected results, both beneficial and problematic, that were obtained from testing SPATULA.

In all of our test domains, iterative abstraction and assumption counting worked synergistically together and improved the results which were obtained when either was used alone. The simultaneous use of these method increments produced an interesting bias, in which, all else appearing equal, the problem solver chose to attempt the easiest subproblems first. This bias manifested itself most notably in the TOH domain, where a novel problem-solving strategy was generated, in which at each decision point, the

easiest subgoal along the optimal solution path was selected.

The abstract plans learned using SPATULA produced good solutions, were easier to learn, and transferred in a wider range of situations than corresponding non-abstract rules. However, because more abstract rules could fire per step, an average increase in time per step was observed. This effect increased with the complexity of the task, since the larger the number of comparisons explored, the greater the number of abstract rules learned.

The plan transfer experiments produced an additional unexpected and interesting effect: the wider the use of plans learned in previously encountered situations, the greater the ability of the system to detect and correct over-general portions of the plans, by re-evaluating the situations in which the different plans provide conflicting information. We expect that this effect will be observed with the use of any type of generalized knowledge, and will thus have ramifications beyond the use of SPATULA.

The extended-plan-use method increment produced good solutions in domains for which it was tested; use of the problem solver's abstract sub-search rules caused only a small increase in solution degradation, with a significant decrease in problem-solving steps required. However, the results indicated that the lower-level rules which did not discriminate among options were of only limited utility. Thus, the experiments (though still preliminary) suggest that the conservative version of the extended-plan-use method increment — prohibiting transfer of such rules to top-level problem-solving — provides the most useful cost/accuracy tradeoff, and that use of the sub-search plans should be restricted only to those rules which preferred one option over another.

An important issue with extended plan use was that for many tasks, it exacerbated the slow-down per step observed with abstraction. This is because the implementation of the method increment entails firing all applicable sub-search rules and then retracting the less detailed ones. Further work is required to determine the extent to which this slow-down is a Soar implementation issue, and to investigate alternative implementations of the technique.

The two efficiency-driven method increments (goal achievement iteration and the abstraction-gradient method increment) proved effective in the Robot Domain.

There, they allowed many important interactions to be detected, while increasing the efficiency of the search. However, these method increments will not produce good solutions in domains such as the TOH, in which many subgoals are highly interactive, or in which interactions occur at a deep level of operator subgoaling. Thus their use can not be unilaterally recommended.

Chapter 8

Related Work

Within the general framework of using abstraction to guide and constrain search, there are a wide variety of systems which utilize abstraction in some way. There are several dimensions along which such a system can be considered, including:

- the type of abstraction used (its properties, etc.)
- the way in which the abstraction is created (e.g., manually or automatically), when it is created (constructed as part of the problem-solving for the task or via prior analysis, etc.) and with what information.
- how the abstraction is used to reduce search.
- whether or not the system is able to learn about its abstractions, and if so, how the information is used in new situations.

We will first give an overview of the basic types of abstractions found in problem-solving systems and the ways in which they are used to reduce search; then, discuss how systems generate abstractions; and the way in which abstract knowledge can be learned. Last, we will briefly survey other search-reduction research related to abstraction.

8.1 Classes of abstractions

Abstraction research has produced problem solvers which use both TD- and TI-Abstractions [Giunchiglia and Walsh, 1990a]. As discussed in Chapter 1, TD-Abstractions are those for which all theorems in the abstract space are abstractions of ground-space theorems, but not all ground-space theorems necessarily have corresponding theorems in the abstract space. Conversely, TI-Abstractions are those for which the abstraction of any ground theorem is a theorem in the abstract space, but there may exist theorems in the abstract space which do not correspond to any ground-space theorems.

Research which makes use of TD-Abstractions includes Tenenberg's [1988] *downward-solution-property* formalization. [Subramanian and Genesereth, 1987] describes a technique for generating abstractions of a theory which are both TD- and TI-Abstractions with respect to a particular set of task goals. [Levy *et al.*, 1992] also provides an example of a system which uses TD-Abstractions. Given the task of constructing a proof, the system determines heuristically which theorems are likely to provide fruitful information (to be "relevant"), and as a result of this analysis removes "irrelevant" domain theorems. The remaining abstraction of the theory is not guaranteed to find an answer if one exists, but if an answer is found, it will be correct (when a monotonic representation language is used).

In qualitative modeling research, describing a function as monotonically increasing (or decreasing) with respect to some input is a TD-abstraction of the function [Kuipers, 1986]; any behavior which may be generated using the monotonic constraint will hold for a more detailed function as well. Similarly, Weld [1992] describes a class of model approximations called *fitting approximations*, which can generate useful model abstractions. One model is a fitting approximation of another when the difference between the behaviors they predict approaches zero, as a "fitting parameter" of the more detailed model approaches an endpoint of its range. For example, one fitting approximation would be frictionless motion, where friction is the fitting parameter. Given a fitting approximation on a given parameter, model sensitivity analysis can be performed to determine the effect of the simplification, by computing the sign of

partial derivatives in the more detailed model. The analysis allows one to determine whether the approximation will overestimate or underestimate model output parameters. (E.g., a frictionless model will overestimate velocity). Once the sensitivity analysis has been performed, the fitting approximations can be interpreted under certain conditions as TD-abstractions. For example, if it is known that an approximate model always overestimates, then certain questions about the more detailed model may always be answered accurately with the approximate one. The sensitivity analysis allows approximate models to be generated on a per-query basis, with respect to their use as TD-abstractions.

However, most abstraction problem-solvers in the existing body of abstraction research use some type of TI-Abstraction, most usually Proof-Increasing (PI) Abstractions [Giunchiglia and Walsh, 1990a]. As discussed in Chapter 1, with PI-Abstractions, if there exists a ground-level solution to a task, then there always exists *some* abstract proof for which a monotonic refinement to the more detailed proof is possible [Knoblock, 1991]. With PI-Abstractions, the structure of the abstract proof, rather than just its final output or evaluation, is likely to be useful when used to guide the more detailed search; it is therefore reasonable to try to "fill in" such an abstract proof to get a ground-level solution. Because SPATULA uses this type of abstraction, such systems will be the primary focus of the survey below.

8.2 Abstract Search Representations

In this section, we briefly discuss the paradigms in which abstract search may take place. Planners are usually classified as performing either "state-space" search or "plan-space" search. Below, we describe each type of search, and provide a few examples of systems which fall into each category[1].

[1]The sets of examples are not intended to be exhaustive, but are presented for the purpose of illustration. Additional systems from each category will be described in the following sections.

8.2.1 State-space planners

SPATULA is an example of a state-space planner. With state-space problem-solving, lookahead search, or projection, takes place in an abstract problem space or spaces. Information about the abstract search is mapped to a more detailed space to provide search constraints. The search constraints may be represented in at least two different ways: 1) as actual subgoals generated from abstract states and mapped back to the more detailed representation; or 2) suggestions about operator sequences in the more detailed space (where the applicability conditions of the operators then implicitly define a set of subgoals which the system must achieve). The latter is the approach taken by SPATULA.

Examples of planners which use state-space search include the following systems:

Planning GPS [Newell and Simon, 1972] was one of the earliest examples of a problem-solver which used abstraction. It used a reduced-model problem space in the domain of propositional logic, constructed by replacing all logical connectives by a single abstract symbol. The problem was first solved in this abstract space using GPS, by selecting operators which reduced a (largest) difference between the current and goal state. The abstract solution was then mapped back into the original problem space and refined.

ABStrips [Sacerdoti, 1974] used multiple levels of abstraction created by precondition elimination. Levels of abstraction corresponded to levels of precondition *criticality*, where at each abstraction level, only those preconditions with the corresponding criticality value or higher were considered. The criticality levels were defined for a domain via a semi-manual process. The system was provided with a partial criticality ordering of domain literals, and attempted to find a short plan to achieve each literal. Those literals for which no such plan could be found were assigned a criticality greater than the highest rank in the partial order. Otherwise, the literal was assigned its rank in the partial order. The same set of abstraction levels was used for each task in the domain. This analysis did not consider possible interactions between subgoals, and could cause the system to perform inefficiently for tasks in which there was subgoal interaction [Knoblock, 1992].

Hansson and Mayer [1989] describe the construction of subgoals through abstract

problem-solving using precondition relaxation. Their experimental domain is the Eight-Puzzle. They only map back those abstract states which are "legal" in the ground space; e.g., states with stacked tiles are eliminated from consideration as subgoals. They discuss the use of both *static* and *dynamic* subgoaling. With static subgoaling, the ground-level search is guided by a static series of subgoals. With dynamic subgoaling, the selection of — and abstract search for — subgoals are interleaved; thus allowing one to inform the other. The system described in [Hansson and Mayer, 1989] implements only static subgoals, but the authors discuss future work involving dynamic subgoaling as well.

8.2.2 Plan-space planners

Plan-space planners construct a solution by searching through a space of partial plans. Typically, they are least-commitment planners[2]. Abstract planning is performed by constructing a partial order of abstract operators and then expanding or refining the operators — e.g., by expanding them into a set of lower-level actions, or by attending to their preconditions. The partial orderings on the abstract operators serve to constrain the search.

Plan-space planners sometimes encounter what has been termed the "hierarchical inaccuracy" problem [Yang, 1990], in which, if different parts of the plan are not refined in temporal order, inconsistencies may arise — a temporally later part of the plan may be based on conditions which will not in fact turn out to exist. In plan-space search, careful attention must be paid to ensure that changes are propagated properly, so that these situations are detected. State-space searches, since they re-project the search at each abstraction level, find it easier to avoid this problem.

Plan-space planners which use abstraction to hierarchically decompose the plan space include NOAH [Sacerdoti, 1977], SIPE [Wilkins, 1984; Wilkins, 1988], MOLGEN [Stefik, 1981], NONLIN [Tate, 1977], and O-Plan [Currie and Tate, 1988]. For all these systems, the abstraction hierarchies must be pre-defined.

[2] Least-commitment planning is sometimes confusingly characterized as "non-linear", since the plan-space representation provides a means for reasoning about and avoiding the *linearity assumption*. However, state-space planners may be designed to avoid the linearity assumption as well [Rosenbloom *et al.*, 1992; Veloso, 1989]. See [Hendler *et al.*, 1990] for a discussion of this issue.

With ABTWEAK, [Yang and Tenenberg, 1990] presented a formalization of domain-independent, least-commitment planners with respect to precondition abstraction, and showed that the monotonicity property [Knoblock, 1991] holds in such planners.

In the future, it may prove fruitful to use SPATULA's techniques in conjunction with a least-commitment planner. A least-commitment planner, though it allows a partial operator ordering, must still perform search at points in the plan for which there is more than one way to accomplish a subgoal. Use of iterative abstraction and assumption counting would provide a means for a plan-space planner to heuristically prune this search and commit to expanding only a subset of the possible search paths.

8.3 Use of Abstract Search Information

Abstractions are used to guide and reduce search for a ground-level problem. Their use can be categorized according to how the results of abstract search are interpreted by the system. The results of abstract search may be used in two general ways:

- As (admissible) search heuristics, which may then be used with ground-level search technique such as A* or IDA* [Nilsson, 1980; Korf, 1985a].
- To suggest search or execution subgoals in a more detailed space; or to suggest actions in a more detailed space. (Suggested actions may then provide the problem solver with new subgoal information).

Each use will be discussed in more detail below, and examples given of systems which use abstraction in such a way.

8.3.1 Using Abstractions to Produce Admissible Search Evaluations

For many abstractions, including PI-Abstractions, the length of an optimal abstract solution to a task is always shorter than the optimal corresponding non-abstract solution. Therefore, abstract solutions are a source of admissible heuristics for the

non-abstract problem-solving. Much work has been done on deriving such admissible heuristics [Gaschnig, 1979; Kibler, 1985; Pearl, 1983; Valtorta, 1984].

Typically, the admissible evaluations are coupled with search algorithms which ensure optimality, such as A* search. In such a paradigm, abstraction is not used to directly eliminate part of the non-abstract search tree. Rather, all (non-abstract) leaf nodes are still considered at each step in the search, using the abstract evaluations, to determine which is best to expand. Therefore, this approach does not require that abstract problem-solving produce mappable solutions, just an evaluation as output.

Valtorta [Valtorta, 1984] has shown that if optimal-output (e.g., breadth-first) dropped-precondition abstract searches are used with A* search in this manner, the total amount of search required in the two spaces (ground and abstract) will always be greater than that required to simply perform breadth-first search in the non-abstract space. Therefore, if this type of abstraction is to be useful in generating admissible heuristics, search in the abstract space must be constrained to be less than a complete exponential search.

For example, Mostow and Prieditis' ABSOLVER [Mostow and Prieditis, 1989] generates admissible heuristics by dropping information about a problem definition via various problem transformations. Then, the abstractions are analyzed and optimized (e.g., by factoring the abstract problem into independent subproblems). The optimizations produce less expensive abstract searches. Then, the system determines the conditions under which the abstractions may be used to produce evaluations which will speed up the base-level search.

8.3.2 Using Abstraction to Suggest Actions or Search Subgoals

The problem-solver can use information from abstract searches in a less conservative manner than that described above. That is, it can use the information from the abstract search to *constrain* the set of ground-level search paths that it considers. The constraints may provide information about which problem subgoal(s) to pursue in a given situation during search, or may constrain the set of actions which will be

applied in a situation. In contrast to the approach described above, here some search paths in a more detailed space are eliminated from initial consideration based on the abstract information (though some may be reconsidered if backtracking is necessary). As Korf [1987] and others have shown, the use of abstract solutions to constrain more detailed searches can provide an exponential reduction in total search complexity. As discussed previously, the characteristics of PI-Abstractions make them natural candidates for use with this type of approach.

Within this framework of using the abstract plan to constrain search, there can be various degrees of commitment to the abstract plan and the extent to which the abstract problem-solving is used to guide the more detailed problem. Systems may be classified along several dimensions with respect to the use of constraining information provided by the abstract plan, as follows.

8.3.2.1 Extent to which the Abstract Plan is Used

The problem solver may use *all* of an abstract plan to guide more detailed search, and commit to using the entire plan until part of it proves inapplicable. Alternatively, the problem solver may use only an initial segment of an abstract plan to guide refinement, and then re-evaluate the problem, thus interleaving search for abstract plans and expansion of those plans. Such an approach allows the abstract re-evaluation to utilize any new information provided by the previous plan refinement.

Historically, most abstraction planners do in fact attempt to use the full abstract plan, backtracking only if difficulties are encountered. In contrast, SPATULA allows the second, interleaved approach to be used by the problem solver if desired. In the work by Hansson and Mayer [1989] described above, their proposal for *dynamic* subgoaling is another example of interleaved abstract search and refinement, though it had not yet been implemented.

The RTA* algorithm [Korf, 1990] takes an interleaved plan/execute approach similar to both that suggested in Hansson and Mayer and that used by SPATULA, though it does not explicitly use abstraction. RTA* utilizes a planning algorithm (approaches such as time-limited A* or threshold-limited iterative deepening are suggested, but

other methods may be used) to produce an evaluation function. The evaluation function is used to estimate the merits of every node relative to the current position of the problem solver (rather than the initial state), and executes a move based on that evaluation. During search, RTA* keeps a list of the states previously visited, and uses the heuristic that it should return to a previously visited state when the estimate of solving the problem from that state plus the cost of returning to that state is less than the estimated cost of going forward from the current state. It is shown that for problem spaces in which a goal state is reachable from every state (and in which there are positive edge costs and finite heuristic estimations), RTA* will find a solution.

Plan-space planners which refine portions of their plan "vertically" without completely expanding the plan at each higher level can also be viewed as working in an interleaved plan/refine mode. (As discussed above, problems can arise if the planner is not careful in propagating the results of such refinement throughout the rest of the plan structure).

8.3.2.2 Interpretation of the Abstract Plan

The abstract proof provides ordering information about some of the components in the ground-level solution, but does not in itself specify how the problem solver will use the information. For example, if the states of the abstract proof were to be mapped to a series of ground-space subgoals, this information might be interpreted by the problem solver as specifying an ordering on subgoal *achievement*, but still allowing interleaved work on more than one subgoal at once. Alternatively, the series of subgoals might be interpreted by the problem solver as a series of independent subproblems, each achieved before the next is begun. This is the approach taken by many systems, e.g. [Knoblock, 1991; Sacerdoti, 1974]. The assumption of subgoal independence allows a best-case exponential-to-linear reduction in search complexity from the abstract to the ground space, using multiple levels of abstraction [Korf, 1987; Knoblock, 1991]. However, with PI-Abstractions it is not guaranteed, for an arbitrary domain and abstraction, that a ground solution is reachable by assuming subgoal independence; thus this is not always a reasonable assumption.

The interpretation of an abstract solution in the less abstract space (e.g., whether

or not the subgoals generated by the abstraction are treated as independent) is often viewed as being part of the abstraction method. However, this need not be the case. With SPATULA, the use of abstraction is implemented such that the refinement of an abstract proof is influenced by the various problem-solving methods that the problem solver uses for a task. Different refinement approaches may be implemented as appropriate in different domains.

8.3.2.3 Context of Abstract Plan Use

As described in Section 4.4.2, the system may use its abstract information in different contexts. It may use the information to decide what action to execute next (i.e., what to do in the "real" world), or it may be more conservative and use the information to suggest subgoals in a more detailed planning space. The advantages of each approach were discussed in Section 4.4.2.

Systems which use abstraction to guide further planning include those which abstract during both "plan space" and "state space" search. All of the systems discussed above in Section 8.2 used the results of abstraction to guide search in a more detailed planning space. Additional examples of such planners will be given Section 8.4.

Using Abstract Plans in the Execution Space. Alternatively, the system may map information from the abstract plan directly into the "execution" space. Again, the abstract plans used by such systems can be constructed using state-space or plan-space search; and as above, the abstract plans may be used with varying degrees of commitment. Often, these systems utilize "impasse-driven" or "failure-driven" refinement of their abstract plans, in which feedback from execution of the abstract plan in the environment suggests when and where refinement is necessary.

SPATULA is an example of such a system, since it maps the results of abstract state-space search directly back into the execution space, and interleaves plan refinement with execution. Other systems which make use of this general paradigm — that of using abstract information to select executable actions — include those whose abstractions are "necessary" rather than deliberate: during the planning phase, a system may not have all of the information required to construct a complete plan,

and thus may be required to construct incomplete, or partial plans. Such systems are not always viewed as using abstraction, but the distinction is in the semantics of what is abstracted; the impact on the solution proof tree can be the same as that which occurs from deliberate abstractions.

Among the abstraction systems which map abstract plans into the execution space are those which are categorized as performing assumption-based or default reasoning, e.g. [Morgenstern, 1990], abduction, e.g.[Fawcett, 1989; Elkan, 1990a; O'Rorke, 1990], theory refinement, e.g. [Rajamoney and DeJong, 1987], and deferred plan construction, e.g. [Gervasio and DeJong, 1989].

For example, theory revision/refinement, such as that discussed in [Rajamoney and DeJong, 1987], often involves construction of (what turned out to be) an abstract plan for a task, and then correcting and refining the theory (filling in the holes in the proof) based on feedback about mismatches between the abstract theory's prediction and what actually happened.

Abductive reasoning involves the construction of explanations which incorporate hypotheses for which there is no deductive proof, or for which proof is very expensive to obtain. While constructing the explanations, the system must reason about which hypotheses, or assumptions, it will use — that is, which part of its implicit non-abstract proof tree it is best to prune away[3].

Similarly, planners may have incomplete information about future configurations of the environment; here too they are forced to make assumptions. Therefore, in addition to abstracting for tractability purposes, planners may postpone expansion of part of their plan out of necessity. In [Gervasio and DeJong, 1989], a system is presented which plans using "placeholders" for parts of the plan which require reactivity. To reason about the reactive components of the plan, the planner uses *contingent explanations*, which conjecture the existence of values which will satisfy certain conditions.

Further examples of systems that use execution-space abstract plan expansion are discussed in more detail in the following sections.

---

[3]Whether a proof is considered an "explanation" or a "plan" depends upon the way it is used by the system. For the purposes of this chapter, the terms are used interchangeably; both are the products of reasoning about a domain theory.

8.4 Automatic Creation of Abstractions

Until recently, most research with abstraction problem-solving involved pre-defined abstractions and abstraction hierarchies. However, recent work has focused on development of ways in which the problem solver can automatically, or semi-automatically, generate the abstractions itself. SPATULA is an example of such a system.

Those automatic abstraction systems which produce PI-Abstractions — as does SPATULA — are particularly relevant. The different ways of automatically producing PI-Abstractions can be thought of as different methods of automatically deciding how to prune a proof tree to produce an abstract proof tree which subsumes it. Most automatic abstraction methods described below fall into this category.

Automatically generated abstractions can be created prior to planning by problem-space analysis, or be driven by the planning process (as is the case with SPATULA). In addition, abstract heuristics can be constructed post-planning by empirical analysis of the problem traces. Hybrid approaches are possible as well. In this section, we discuss systems which utilize the first two approaches; the latter approach, which does not tend to actually utilize abstract problem-solving, is discussed below in Section 8.5.

8.4.1 Determination of abstractions prior to search

Systems which create abstractions prior to problem-solving can take both analytical and empirical approaches. The analytical approaches typically utilize declarative information about operator preconditions and effects, and perform a static problem-space analysis. Examples of such approaches include the following systems:

ABStrips, discussed above, produced levels of precondition abstraction based on a semi-automatic process. Historically, ABStrips has had a great impact on automatic abstraction research.

PABLO[Christensen, 1991] performs reduced-model abstraction. It uses the general philosophy behind ABStrips, in that it ranks predicates based on how many steps are required to achieve them — predicates which require longer plans are worked on first. PABLO allows a finer-grained abstraction hierarchy than ABStrips; it relaxes predicates by regressing them over operators to produce the weakest relation that

ensures the subsequent truth of the predicate after executing the operator. Any predicates which are true for some regression sequence of n ops are ignored at the nth abstraction level. The system's analysis of plan length depends upon knowing information about the initial state, as well as the operator descriptions. E.g., a travel domain example is given in which whether or not "having a bus token" is a detail depends upon whether or not money is available with which to buy the fare, etc. Thus, the analysis must be done with respect to the initial state for a particular task. A potential difficulty in using the method is that in complex domains, it may be very expensive to perform a complete analysis (particularly if recursion can take place). In addition, as with ABStrips, the system does not take into account interactions between achievement of different literals, and thus can be inefficient in domains for which there are such interactions.

PABLO can store the compiled plans learned from its abstraction derivations to use in a reactive manner if it is interrupted during planning. This use of its knowledge bears similarity to approaches described in [Etzioni, 1991] or [Schoppers, 1990], though it does not analyze interactions between its compiled plans.

OPIE[4] [Anderson and Farley, 1988; Anderson and Fickas, 1990] takes a hybrid approach with respect to generating its abstractions prior to or after search. Prior to search, it produces hierarchies of operator generalizations, either from generalizing over objects which are used in the same way (e.g, objects which may be "grasped" in the same way), or from generalizing by combining those operators which share some, but not all, literals. (In the worst case, the hierarchy-building process may take an exponential amount of time). Then, ground-level plan traces are generalized using this information to produce abstract macros (i.e., a plan containing abstract operators). Thus, problem-solving influences the actual abstract plans which are produced — the abstractions preserve the user/consumer structure which was present in the non-abstract plan. The generalization is conservative; it is taken as far as possible up the hierarchy without removing plan structure. Use of the abstract plans then allows deferred commitment to a particular operator as well as deferred reasoning about preconditions and secondary effects; this reduces the branching factor. A potential

[4] Apparently previously named PLANEREUS.

problem with OPIE's approach is that those preconditions which did not need to be achieved in the ground-level plan are then abstracted away, although they may turn out to be important in future situations. (SPATULA avoids this particular problem by incorporating knowledge about those preconditions which *were* met into its comparative search control rules.) Alternatively, the preconditions which were incorporated into the abstract plan may in fact — since generalization is conservative — be details which cause unnecessary abstract computation.

ALPINE[Knoblock, 1991] applies reduced-model abstraction. Abstractions come from determining interactions between operators in domain (in addition, information about particular task goals is utilized to avoid unnecessarily restrictive abstractions for that task). The analysis produces partial orders of problem-space literals, which are used to create abstraction hierarchies with the *ordered monotonicity* property. If a hierarchy has this property, then literals established at one level of abstraction can't be unestablished by anything done at a lower abstraction level as the plan is expanded. This property then greatly increases the chance that an abstract plan will "work", since expansion at a lower level is guaranteed not to spoil the plan (the property still doesn't guarantee that a particular abstract plan is refineable, however). The abstraction hierarchy is then used to guide search in successively more detailed problem spaces, using state-space search. The approach can be overly constraining. SPATULA can produce useful abstractions when ALPINE can not. For example, this is the case for the EP formulation used in Chapter 6.

Other abstraction methods utilize empirical testing to decide which abstractions to use. These include the following systems:

POLLYANNA[Ellman, 1988; Ellman, 1990] has a space of abstraction transformations which use generic simplifying assumptions and transformations, including Bernoulli's Principle of Insufficient Reason. The transformations define a space of simplified theories, in which abstract problem solving is performed. The abstract problem traces are then used to generate abstract heuristics using EBL. The space of heuristics can then be evaluated empirically, using a test set of tasks, to determine the best region of the space with respect to (domain-dependent) cost/accuracy tradeoffs. POLLYANNA's method of learning abstract rules is similar to that used by

SPATULA, but the method of building the abstractions is more domain-dependent and less context-sensitive, and POLLYANNA does not have the ability to learn and reason about plans at multiple levels of abstraction.

Martin and Allen[Martin and Allen, 1990] use two semi-independent systems, one reactive and one which does planning. Based on statistical information about similar problems and knowledge of the reactor's capabilities, the planning component decides whether to decompose a subproblem during planning, or to leave the plan segment abstract and let the reactive system refine it.

ABSOLVER[Mostow and Prieditis, 1989], described above, takes a similar generate-and-test approach in determining what abstractions to use (though its use of these abstractions is different).

8.4.2 Determination of abstractions during search

Another group of abstraction problem-solvers abstracts in a more situated manner: the abstractions are driven either partially or entirely by the current problem-solving context. SPATULA belongs to this group.

GRASPER[Bennett, 1990a; Bennett, 1990b] uses approximations while planning. The system is hybrid with respect to the origination of the abstractions. Though some of the abstractions are generated prior to problem-solving, others are generated during planning. Some of these abstractions are generated because of sensor limitations (externally generated abstractions). In addition, other abstractions deliberately reduce the complexity of sensed objects, e.g. by transforming a sensed polygon to one with a smaller number of sides (internally generated abstractions). These abstractions are dynamically generated, since the problem context determines the particular approximations that are created. GRASPER is similar to both POLLYANNA and SPATULA in that EBL is performed over the abstract problem-solving to learn approximate plans. Only if execution failures occur does the system reason about the approximate nature of its data. It then tunes its rules, focusing on those aspects of the plans in which expectations failed. GRASPER differs from SPATULA in that its dynamically generated abstractions are data abstractions, rather than operator abstractions. (GRASPER does employ rule approximations as well, but these are pre-defined).

Chien [Chien, 1989] increases the tractability of planning by using limited inference. He does this by procedurally relaxing the truth criterion used by his plan-space planner; established facts in a developing plan are not protected against conditional effects, unless the conditional effects are used to achieve a goal. Thus, the abstractions are dynamically determined by the problem-solving context. (This type of abstraction affects different parts of the proof tree than does precondition abstraction, and will not tend to impose the same hierarchical structure on the search.) The system uses failure-driven plan refinement— when failures occur, simplifications are incrementally retracted by examining the failure and adding a previously-ignored negative effect-protection interaction. The approach assumes that a strong diagnostic capability exists to construct explanations for failures, the cost of failure is low, and that the system is allowed multiple chances to solve a problem.

The non-monotonic reasoning system described in [Elkan, 1990b; Elkan, 1990a] uses the heuristic that default assumptions are reasonable points at which to abstract proofs. The default assumptions are represented as negated conditions. Normally, the system, called PERFLOG, uses negation-as-failure to show that the negations do not hold; that is, it attempts to prove the positive version of the negated condition. This can take an arbitrarily large amount of computation. With abstraction, the system searches iteratively deeper to prove the default assumptions don't hold; if it can not disprove them for a particular iteration depth, it assumes that they do hold at that depth. Thus at each iteration the proof becomes increasingly less abstract. The more resources given the planner, the better its plans will be. The "depth" measure for the iterative process is defined using *conspiracy numbers* [Mcllester, 1988]. The conspiracy set then implicitly defines the abstractions constructed by the system. Thus, the abstractions are dynamically generated, and determined by the conspiracy sets for a particular task. The SEPIA system [Segre and Turney,] has incorporated Elkan's technique to do route-planning.

The system bears a similarity to both SPATULA and iterative deepening [Korf, 1985a] (described in Section 5.3) in its iterative approach to constructing successively more detailed plans by expanding deeper proof subtrees. The differences are that PERFLOG is already provided with information about allowable abstractions (the

default assumptions), and is iterating to find the most accurate plan in the time available. SPATULA does not require such semantic knowledge (though it could make use of this information if it had it) and instead uses its method increments to try to discover a useful set of abstractions with respect to the cost/accuracy tradeoff.

The problem of constructing abductive explanations is very similar to a default reasoning task. The issue of deciding which hypotheses to use is close to that of deciding which default assumptions to make — the impact on the proof tree is the same — although there can be a semantic distinction in that with abduction, a system may not be able to prove its hypotheses, rather than choosing not to for computational reasons.

O'Rorke [1990] describes a system called AbE which performs best-first search through the space of abductive explanations, using a scoring function which includes preference of those explanations which make the fewest assumptions. Fawcett [1989] describes an abductive explanation scoring function which includes not only number of assumptions, but length of explanation and use of more specific rules in the explanation construction.

The abductive comparison functions, particularly Fawcett's, are similar to those currently used by SPATULA; both make use of the idea that although such scoring functions are meaningless in an absolute sense, they provide relative information about options; and both prefer shorter explanations/solutions with fewer assumptions. These systems differ from SPATULA in that the abduction systems are biased towards more detailed solutions (e.g., Fawcett's system does not allow the use of assumptions for an antecedent once the antecedent has been shown to have a complete explanation) whereas SPATULA uses iterative abstraction to look for the most abstract solution which is still useful in guiding search.

8.5 Learning Abstract Heuristics

The more general, or more abstract a search heuristic is, the easier it should be to match and the more widely applicable it can be. A problem-solver with learning capabilities can use abstraction to produce inductively generalized heuristics, which

can then guide problem solving both in later parts of the same task, and in new tasks.

There are at least two general methods by which a problem-solver can learn abstract, or more general, heuristics and plans (where "learning" is loosely used to mean that the system is able to remember the plan or heuristic and has the potential to use it in a new situation, distinct from the one in which the heuristic was originally learned). It can 1) learn from performing abstract problem solving; or 2) generalize from non-abstract proof trees/concepts to produce abstract heuristics.

8.5.1 Learning from abstract problem-solving

As discussed in Chapter 4, learning from problem-solving with an abstract domain theory can produce rules which are deductively correct with respect to the abstract theory, but *inductively generalized* with respect to the corresponding ground-level theory. The abstract theory provides an inductive *bias* to the learning mechanism. The abstract rules can be used to guide further abstract problem-solving, or can be used as abstract heuristics for more detailed problem-solving, or both. The advantage of this approach to generalization is that it can make the learning process more tractable — there is a smaller proof tree to generate and explain.

Soar with SPATULA learns abstract plans in this manner, as do other systems including [Ellman, 1990; Bennett, 1990b; Chien, 1989], discussed above. Recent work has also been done to combine Alpine (discussed above) and Prodigy [Knoblock, 1991; Minton *et al.*, 1989; Knoblock *et al.*, 1991]; Prodigy's EBL capabilities have been used to allow learning at each abstraction level that Alpine generates. Thus far, the abstract rules have not been used in different abstract levels than the ones for which they were generated, but future work is planned in this direction.

SPATULA is distinct from the other systems which use this technique in that Soar's capabilities allow it to take a uniquely integrated approach to both learning and using abstract plans. It was in fact the first system to learn inductively via EBL from abstract searches. It uses its abstract plans for new situations in both the abstract and non-abstract spaces — which we believe no other system does — and interleaves learning with execution to produce a multi-level refinement of the abstract plan. With SPATULA, abstract learning only occurs when necessary, is context-sensitive,

and allows learning of rules from different levels of abstraction during the same search.

8.5.2 Abstracting from more detailed problem-solving

An alternative approach to learning abstractions is to perform "ground-level" problem-solving, and then abstract from the problem trace(s) or concepts for use in future situations. This approach does not provide any computational advantages during the initial non-abstract problem-solving and learning, but may more easily allow induction over multiple problem traces, or comparison of various small approximations to the same non-abstract rule.

For example, Keller [Keller, 1990] suggests an approach in which a concept (learned using EBL) may be made approximate by replacing subtrees in its explanation by either *true* or *false*. Then, given information about acceptable cost/accuracy tradeoffs in the domain, experiments can be performed to determine the most useful abstractions; these can be used as future heuristics. This bears many similarities to the work described in [Ellman, 1988]; the difference is that here the heuristics are abstracted after problem-solving rather than during.

A related approach is taken in Chase et al. [Chase *et al.*, 1989], which describes a method of approximating rules by analyzing which rule conditions tended to be statistically predicted by other conditions (the rules are analyzed in top-to-bottom order) and removing them from the rules.

OPIE [Anderson and Fickas, 1990], described above, creates abstract macro operators from non-abstract problem traces: it is a hybrid system in that its abstractions are partly created using problem-space analysis and partly driven by generalization from the non-abstract problem solving.

Danyluk [Danyluk, 1989] describes an abduction system in which incomplete abductive explanations are examined by a similarity-based learning component. Using contextual information from previous explanations, the SBL component is better able to induce useful abductive hypotheses.

In the SteppingStones system [Ruby and Kibler, 1991], a non-abstract problem is examined to produce generalized state descriptions which can be used heuristically as subgoals in new situations. The system does not always use the subgoals; first it tries

to achieve a task's goal conjuncts using goal protection — that is, the achievement of each new conjunct is not allowed to undo previously achieved conjuncts. If this tactic fails, then the system searches its memory for applicable, previously learned, subgoal sequences. If this fails as well, then it does an exhaustive search, solves the (sub)problem, and generalizes new subgoals from the solution trace. The subgoals appear to be abstract in that they focus only on predefined aspects of the state which relate directly to maintaining previously achieved goal conjuncts while achieving the new one.

8.6 Analyses of Abstraction Properties

Most analyses of abstraction properties have focused on a best-case refinement scenario, in which no backtracking to higher abstraction levels is necessary. Recently, probability-based analyses have considered scenarios in which an abstraction is not always refineable.

Bacchus and Yang present a probabilistic analysis of the utility of using multiple levels of abstraction [1992], which extends earlier work by including as a parameter the probability that a TI-abstraction may be successfully monotonically refined. The results show that abstraction is cost-effective both when there is a high probability of refinement, and a low probability. (If there is a low probability of refinement, then even though the number of bad subtrees which must be searched is large, it does not require much effort to search them since most paths will die out at a high abstraction level). In the middle region, where the probability of refinement is not small nor very high, search complexity depends upon both the number of abstraction levels and the branching factor, and may become almost as expensive as non-abstract search. The analysis assumes that the individual "gap subproblems" created during refinement from one abstraction level to another may be solved without significant interaction, and that the abstraction hierarchy is regular, in the sense that the number of operators needed to solve gap subproblems is approximately constant across abstraction levels. The authors use this analysis in an extension of ALPINE called HIGHPOINT, in which the partial ordering of literals produced by ALPINE is further refined via an

analysis of the refinement probabilities.

In contrast to the approach in [Bacchus and Yang, 1992], the use of SPATULA does not necessarily imply that the system will backtrack at a violation of monotonic refinement; instead, the plan may be patched if possible. The approach used for a domain depends upon the search knowledge used in conjunction with SPATULA for that domain. However, with patching instead of backtracking, there still remains an analogous difficult middle region of search, in which it may take much goal replanning to find a refinement or discover that one does not exist. SPATULA's method increments provide heuristics for generating situation-dependent abstractions that avoid this region.

In [Williams, 1992], Williams builds a probabilistic model of abstraction, which suggests that the probability of successful refinement should not be modeled as independent of the abstraction "strength"(the ratio of the length of a proof in the ground space to that in the abstract space) — a more reasonable model is produced if the refinement reliability decays as the abstraction strength is increased. The model produced here is dissimilar to that of [Bacchus and Yang, 1992], because here it is not assumed that the abstraction is Theorem-Increasing. Because it is possible that no solution can be found at an abstract level, the predicted cost must include the probability that search at the ground level must be re-initiated. Due to this restriction, the model does not show a decrease in search cost as the probability of refinement failure becomes very high.

Ginsberg [Ginsberg, 1991] discusses the heuristic of abstracting default assumptions in the context of a non-monotonic reasoning system, and gives an analysis of the situations under which the basic use of this heuristic will be expected to be cost-effective, depending on the ratio of default assumptions to other subgoals, and the expected probability that a default will fail. His approach is similar to that of [Williams, 1992], in that he assumes that re-planning occurs at the ground level if an abstraction refinement fails.

8.6.1 Aggregation Abstractions

This chapter has focused primarily on systems which use approximation abstractions, since this is the technique employed by SPATULA. Some abstraction planners employ aggregation abstractions instead. Aggregation abstractions are those in which more than one object at a lower level of abstraction are grouped into a single object at a higher abstraction level. For example, [Benjamin, 1989] describes an algebraic method for problem decomposition as the formation of a hierarchical machine. In the three-disk Tower of Hanoi domain, the method maps the top two disks to a single disk in the abstract space. Campbell's CHUNKER [Campbell, 1988], in the chess domain, generates aggregate abstractions based on groupings of chess pieces, and then searches the abstract space based on interactions between the groupings.

8.7 Related search reduction approaches

Abstraction is one way to reduce problem-solving search. There exist many other related approaches to search reduction.

Section 5.8 described how SPATULA provides the problem-solver with several search biases. Such a bias determines which parts of the space of possible plans are searched, and thus influences which solutions are generated [Rosenbloom *et al.*, 1992]. [Lee and Rosenbloom, 1992] describe a framework for explicitly representing several other planning biases — *linearity, goal protection*, and *directness* – and combining them to produce a planner with multiple planning methods, in which the most restricted approaches are tried first.

Problem reduction techniques are those which decompose a problem into subproblems. This definition encompasses a broad range of approaches. For example, a set of transformation rules may be applied to a task to break it into subtasks, where some of the subproblems created via reduction transformation may potentially be in a different problem space than the parent problem [Bresina, 1988; Amarel, 1967; Riddle, 1990]. Abstraction can be viewed as one source of heuristics about how to break up a task, since an abstract solution may be used to define subproblems in a more detailed space.

Similarly, Joslin and Roach [1989] propose a graph formalism for analyzing goal conjunct interactions in domains with finite search spaces, to determine when a task requires a solution in which the goal conjuncts interact.

Lansky compares abstraction with *localization* [Lansky, 1992], a search reduction technique implemented in the GEMPLAN system, which decreases the search space by working on (possibly overlapping) subproblems (regions) independently, and then modifying and combining the sub-plans so that they are globally consistent. She claims that abstraction is subsumed by localization, and makes this mapping by utilizing her planner's capability to incrementally add regions, constraints on the regions, etc.

Macro-operators package the effects of a series of actions into one aggregate operator [Fikes and Nilsson, 1971; Iba, 1989]. In contrast to abstract operators, the macro-operators are added to a domain's original search space. Because they encode more than one step into a single operator, the use of macro-operators reduces the number of operators required to construct a solution during search, and thus can exponentially reduce the size of a search space [Korf, 1987]. However, depending upon the implementation of a system, the addition of compiled knowledge such as macro-operators to a domain sometimes reduces efficiency [Minton, 1990], because it increases the breadth of the search. Soar's chunking mechanism allows the production of macro-operators, as described in [Laird *et al.*, 1986a].

In [Korf, 1985b], Korf describes a method for solving a problem by developing a set of macros and compiling a macro table. For each macro, the table specifies the context under which it may be applied to achieve a subgoal, such that it leaves all previously achieved subgoals achieved on its completion (though the previously achieved goals might be temporarily violated during macro application). Tasks which admit such an approach are termed *serially decomposable*. The approach was developed to address the solution of tasks which are serially decomposable in this manner, but can not be solved by making the linearity assumption. A related approach is taken by Guvenir and Ernst [1990]). They extend Korf's work by developing a technique for finding a series of serially decomposable subproblems. Then, macro-operators can be generated to assist search within such a subproblem.

Chapter 9

Conclusion

This chapter will review the approach to abstraction taken by SPATULA, the thesis claims, and results. Then, we will describe some of the factors that influence SPATULA's abstractions, and the capabilities a problem-solver needs to use SPATULA. Next, we will discuss some of the limitations of SPATULA's weak-method approach, and conclude with future work.

9.1 Summary of Approach

This thesis has described the implementation and testing of SPATULA, a weak method for abstraction.

The foundation of SPATULA is a basic abstraction technique, which provides knowledge to a problem solver about how to automatically create abstract problem spaces from non-abstract domain spaces dynamically during problem-solving. The dynamic reformulation is achieved by abstracting unmet preconditions and propagating the abstractions via partial application. In addition to describing this technique, we have specified a set of problem-space design guidelines which, if followed, ensure that the process of dynamic abstract-problem-space creation will produce abstract searches which are both complete and efficient.

SPATULA's dynamic abstraction method is used by the problem solver in the context of search to make a control decision; abstraction allows the decisions to be

made more tractably, and abstract plans to be produced.

SPATULA's *method increments* then build on the basic abstraction techniques. We have described two method increments which use the problem context to provide the system with heuristics for constructing more useful abstract plans than would be produced solely using the basic abstraction method. The *assumption-counting* method increment helps the system estimate the relative difficulty of refining a plan, and detect subgoal interactions. The *iterative-abstraction* method increment, via relative comparison of options, estimates the most useful level of abstraction for a particular context.

In addition, three other method increments were described. The *extended-plan-use* method increment allows the system to deliberate about the extent to which it will use its abstract plans. The *goal-achievement-iteration* and *abstraction-gradient* method increments reduce the abstract search complexity while employing heuristics about how to focus on the most relevant aspects of the search.

9.2 Review of Contributions and Results

SPATULA is designed to provide a problem solver with an automatic and general weak method for abstraction. The method's abstraction techniques, reviewed above, provide these weak method capabilities. The abstraction techniques are problem-driven and may be applied in any domain without requiring knowledge of what the abstractions should be, or declarative descriptions of the problem spaces. As implemented in Soar, the techniques are provided by new default rules; the architecture remains unchanged.

The use of SPATULA provides a problem solver with an integrated framework for learning, using, refining, and repairing abstract plans. With SPATULA, the problem solver only abstracts when necessary; it does not generate abstractions which it will not use, or perform abstract search when it already possesses sufficient search control to make progress in a task.

The integrated framework enables an emergent multi-level abstraction behavior, in which the multiple levels result from further abstraction during refinement. The

abstract plans produced during abstract search are inductively generalized with respect to the ground-level domain theory and — using the method increments — are learned at context-sensitive levels of abstraction.

The design goals of SPATULA as a weak method were that it — on average — produce useful solutions, increase problem-solving efficiency, and make it easier to learn and use abstract plans. Empirical tests of SPATULA were performed in three distinct domains to determine the extent to which it met these goals. For the empirical tests, none of the problem spaces, including representation of operator actions or preconditions, were driven by the abstraction method (the given problem-space representations, once created, were of course factored as described in Chapter 3). In the Robot Domain, operators and tasks were obtained from an outside source.

In all experimental domains, the abstraction method on average produced good (significantly better than random) solutions. In the TOH domain, it produced *optimal* solutions. It was observed that the two primary method increments worked together synergistically in producing the useful solutions. The optimal TOH solutions were most notably enabled by this synergistic interaction — either method increment alone would not have produced optimal results. The method increments were developed prior to TOH testing, so that the results confirmed rather than drove their development.

SPATULA produced useful abstractions in situations for which other automatic abstraction mechanisms, which order literals according to operator interactions, would not be able to produce an abstract space. For example, the method described in [Knoblock, 1991] would not be able to produce abstractions for our EP domain, in which the operator had just one precondition.

The number of problem solving steps required to find a solution using abstraction was significantly less than the number of steps using the corresponding non-abstract search, and the use of SPATULA rendered previously intractable tasks tractable. For the TOH domain, in which the non-abstract operator subgoals were relatively shallow, abstract search still took less time but afforded the smallest efficiency gains.

Results using SPATULA were also compared with results using first-path search, in which the problem solver selects randomly from its set of possibilities (constrained

by existing search control), at each new operator tie during lookahead search. The first-path searches produced significantly inferior solutions, but in the less search-intensive experimental domains in which the system could not stray very far from a solution path — the smaller Robot Domain room layout and the TOH — were usually more efficient. In contrast, in the more search-intensive experimental domains — the more complex Robot Domain room layout and the EP — a greater amount of search was required to construct even a random solution. With these domains the effort invested to find a good abstract solution — which could then be used to constrain more detailed search — paid off; in such domains, SPATULA allowed tasks to remain tractable which became intractable with random search. This general effect has been observed elsewhere as well [Knoblock, 1991].

The use of SPATULA resulted in abstract plans which were both easier to learn, and applied in a wider range of new situations, than the corresponding non-abstract plans. It was observed that as the regularity of the domain (e.g., with respect to initial states and goals) decreased, there was very little transfer of non-abstract plans learned previously to new tasks — the non-abstract plans were often too detailed to match. In contrast, the more general abstract plans provided a significant amount of transfer to new situations, with only little accompanying solution degradation. However, the abstract rules would sometimes slow down problem-solving time when many of them fired at once, even though they were not individually more expensive. This issue is discussed further below.

9.3 Factors Which Influence SPATULA's Abstractions

SPATULA's abstractions, since they are generated dynamically, can be harder to characterize than those which are statically generated for a task prior to problem-solving.

During lookahead search, the operator subgoaling process can be thought of as dynamically producing an implicit hierarchy, or tree, of preconditions, where each precondition has below it the preconditions of those operators which were applied to

achieve it. The tree is temporally ordered, since the way one precondition is achieved can affect the work needed to achieve a later precondition. At the leaves of the tree are those preconditions which are met or assumed met.

Using SPATULA, two different types of abstraction hierarchies may be built during work on a task. First, there is the abstraction hierarchy generated during the several iterations of a single lookahead search. Such a hierarchy may be described by a series of precondition trees. Roughly, a level of unabstracted preconditions is added to a precondition tree at each new search iteration to make a new tree. However, this description provides only an approximate view of the process. Even though there exists such a monotonic refinement to a full tree, the abstract precondition trees that are actually produced may not necessarily be monotonically refineable in this way. That is, the precondition tree may change nonmonotonically at each iteration. In addition, a new level of preconditions may not be added uniformly across the entire tree, depending upon the particular method increments employed.

A second abstraction hierarchy is defined by the abstractions chosen as a result of the lookahead searches, and used by the problem-solver to guide more detailed searches. This abstraction hierarchy is refined each time new abstract plan fragments are learned; it is the multi-level abstraction described in Chapter 4. Each level in this hierarchy may correspond to more than one level in a precondition tree (and again, the refinement process may not necessarily be monotonic, nor leaves in the tree added uniformly across the whole tree). The shape of the first abstraction hierarchy influences the second.

A useful way to view the factors which determine what is abstracted is to break them down into three parts: those factors which influence the shape of the precondition trees independent of the abstraction method; those factors which influence the way in which a particular search is abstracted at each iteration; and those factors which influence the way in which a particular precondition tree is selected as the result of lookahead search.

- Factors which influence the shape of the precondition trees independent of the abstraction method:

1. The operator representations: what operator knowledge is represented in terms of preconditions (rather than operator implementations, for example).

2. The current problem-solving context, including the order in which the previous preconditions at each level of operator subgoaling have been achieved.

3. Search control knowledge, including what search control knowledge is used with respect to precondition achievement (that is, which operators are proposed to achieve which unmet preconditions). This knowledge need not be traditional MEA search control knowledge, with which for each unmet precondition, operators are proposed whose primary effects achieve that precondition. Rather, this knowledge may be of arbitrary form. (For example, the precondition tree would be flatter than with traditional MEA knowledge if produced by a predominately forward-chaining search strategy). This search control knowledge may change dynamically as new chunks are acquired.

4. What compiled knowledge about precondition achievement exists. This dimension of the abstraction technique was not explored in our experiments. However, if there exists compiled knowledge about how to achieve an unmet precondition, which can be accessed and applied during a decision cycle, then an unmet precondition impasse will not be generated and the precondition won't be abstracted.

- Factors which influence the way in which a particular search is abstracted at each iteration. That is, those factors which determine where the border is on the precondition tree below which all unmet preconditions are ignored:

  The goal-achievement-iteration and abstraction-gradient method increments fall in this category. For example, the abstraction-gradient method increment increases the level of abstraction the further the lookahead search progresses. So with this method increment, the left-hand side of the precondition tree will be deeper than the right (when the search is temporally ordered from left to right).

- Factors which influence the way in which a particular precondition tree is selected – as the result of lookahead search – to be used and refined:

  Iterative abstraction and assumption counting fall into this category. The factors are not independent; iteration level affects which option looks better, and the way in which assumption counting is used in the evaluation function affects which level of iteration is necessary.

All of these factors must be included in a consideration of the abstractions that SPATULA will generate.

9.4 Capabilities required to implement SPATULA

SPATULA has been implemented by adding rules to Soar's memory. However, the ideas behind the basic approach and method increments should be applicable to other systems with the following basic capabilities. (The following assumes that state changes are represented in terms of operators and their effects — whether in plan-space or state-space search).

1. The system must be able to use a problem-solving strategy similar to operator subgoaling. That is, it must be able to work towards applying operators whose preconditions are not met, and not be restricted during search to selecting only those operators for which all preconditions are met. If it does not have this ability, then the abstraction method will have no effect (since there will be no unmet preconditions to abstract).

2. No particular way(s) of achieving an operator precondition should be hard-wired in to the system, such that the additional knowledge
   "if abstracting and current operator's precondition-X not met
   $\Rightarrow$ precondition-X is met"
   can not be added.

3. The problem solver must then able to select this additional way of achieving a precondition, when abstracting.

4. The system must be able to take note of when it is searching to make a choice, and to allow abstraction only within such contexts. With least-commitment planners, this includes contexts in which the system is working with a partial plan for which there is more than one potential ordering of operators.

5. At choice points, the system must have the ability to compare and evaluate all its options, rather than selecting one option arbitrarily and backtracking only if a solution can not be reached.

6. The system must have the ability to *factor* its operations. This may be done representationally, as described in Section 3.3, or by analyzing the structure of the operations, and applying them partially as appropriate. Factorization must be possible for operators as well as any other problem-solving operations which might reference abstracted information.

7. For assumption counting, the problem solver must be able to make note of when and how many times it has used the abstraction rule of Item 2.

8. For iterative abstraction, the system must be able to keep track of the number of levels of operator subgoals generated during search, and to use that information in deciding when to abstract within the search context (when using iterative abstraction).

9. If the system is to implement the learning aspects of the abstraction method, it must have the ability to remember the operator choices made during abstract search, and use them as a plan to guide and constrain more detailed search[1]. The system should have the ability to transfer the abstract plans to new relevantly similar situations as well as using them to guide refinement of the situation for which they were originally learned.

10. For iterative abstraction and the extended-plan-use method increment, the system must have the ability to determine the iteration level at which a plan or

[1]Efficacy of the transfer may not be independent of representation, e.g., the plan transfer may be less effective if the plans are represented monolithically.

plan fragment was learned, and to use that information in deciding when a plan should be used.

9.5 Limitations of SPATULA

SPATULA's weak method approach is not without its limitations. The utility of the abstractions produced is influenced by problem representations, additional search control, domain characteristics, and given tasks. SPATULA performs less usefully when the method increment parameters are inappropriately tuned for a particular domain; and in domains with deep manifestation of plan refinement difficulties or interactions, shallow operator subgoals, or which require many goal conjuncts to be ordered. Below, we discuss each in more detail.

9.5.1 Domain Representations and Method Increment Parameters

As discussed in Section 5.5 and above, SPATULA's abstractions are driven by the way the domain operators are represented and the particular search control used with that representation. If the method increments are not responsive to these factors, the abstractions may not always be useful. For the experiments described in this thesis, the method increment parameters were not adjusted according to the domain characteristics. However, Section 5.5 discussed some possible probabilistic approaches for adjusting the parameters used in the method increments to respond to a given representation and search control. Additionally, if the problem-solver were to be provided with explicit representations of its operators, then further problem-space analysis could be performed (using, e.g., some of the methods described in Chapter 8) to obtain information about the conditions under which subgoals tend to be hard to achieve and/or tend to interact with other subgoals. Such information could be used in adjusting the method increment parameters, as well as selectively abstracting only some preconditions. Such techniques, however, would require more assumptions

about the explicit domain knowledge available than is the case with SPATULA's weak-method approach.

9.5.2 Deep Manifestation of Plan Refinement Difficulties and Interactions

In some domains, difficulties in refining an abstract plan may not manifest themselves until search is performed at a very detailed level. An example might be a domain in which there is only a limited amount of time to perform a task, and in which many task search paths must be almost completely refined to determine whether or not they may be applied within the time limit. Even if SPATULA's method increment parameters were adjusted such that good decisions were made in such a domain, abstraction would not provide big efficiency savings.

9.5.3 Shallow Operator Subgoals

In domains with shallow subgoals, SPATULA has relatively little impact on task efficiency. This was seen in the TOH experiments of Chapter 6. Another class of tasks in this category are those for which there is an ordering of operators, given the search control used in the domain, such that all (or most) operator preconditions are met.

An example of such a domain is the "job shop" described in [Minton, 1990]. In this domain, most preconditions for applying machining operations (such as lathing, polishing, etc.) are requirements such as "object must be cold", or "object must be unpolished". The tasks are such that for most of the operators which achieve a task goal conjunct, there exists a global operator ordering such that the preconditions of the operators are met. Using SPATULA in this domain, the problem solver will successfully choose such an ordering, since it requires the fewest assumptions. However, abstraction will provide little efficiency savings, since little work will be abstracted to find the solution.

9.5.4 Goal conjunct ordering

Because SPATULA creates abstractions by dynamic precondition relaxation (rather than creating a reduced model via deliberate removal of literals from the domain language), the goals of a task are not abstracted. Although propagation of the initial abstractions through partial application can consequently create a reduced problem space, these reductions can not be relied upon to reduce the number of goals which must be considered. Hence, as described in Section 5.9, the efficiency of problem solving using SPATULA's techniques is limited by any work necessary to order a task's goal conjuncts.

As discussed in Section 7.5, this result is of relevance because SPATULA's method increments obtain their leverage from the comparison — via lookahead search — of operators at a choice point. With a non-comparative search method, such as first-path search, such information is not utilized in the decision-making process; the only information used to select an operator is that of the pre-existing search control. So, with non-comparative methods, iterative abstraction and assumption counting provide no new information. Thus, a non-comparative search with SPATULA's method increments will be only as useful as the basic abstraction method with the same technique. E.g., an abstract first-path search will only be useful if failures or goal "clobberings" are detectable at this initial level of abstraction. The more abstract the initial search, the less likely this will be. (A parameter of the basic method can be the number of levels of preconditions the problem solver initially solves for).

This analysis does not imply that a full combinatorial exploration of all different operator orderings must be performed in order to use the method increments, but that some comparative search must occur for the method increments to have an impact. For tasks/methods which require many goal conjuncts to be ordered, it is likely that SPATULA will be most effective when used in conjunction with other techniques for reducing the combinatorics of goal ordering, while still allowing useful comparisons of possible solution paths. The secondary method increments described in Chapter 5 were in fact such techniques (though they did not prove to be efficacious in domains with a high degree of goal conjunct interaction).

9.6 Future Work

There remain many interesting areas for future development of SPATULA. First, we hope to test SPATULA in additional domains. We also hope to explore SPATULA's potential — using iterative abstraction — to aid the problem solver in resource-limited environments, as was discussed in Section 5.3.3.

In addition, we would like to explore the use of SPATULA in further learning experiments, with respect to plan utility, knowledge acquisition in knowledge-poor domains, and detection of plan overgenerality. We would also like to investigate the integration of SPATULA with a least-commitment approach to planning, as well as extension of the method increments in several ways.

9.6.1 Learning Experiments

Soar's ability to learn about abstractions using SPATULA provides several interesting areas for future work.

9.6.1.1 Plan Utility

In Soar, when more rules fire per step, problem-solving time per step increases. This effect can be expected for any situation in which the generality of the learned rules is increased, and is not specific to use of SPATULA. It is hypothesized that if 1) the individual rules do not increase in expense; and 2) the number of rules firing per step can be bounded; then the slow-down will not occur with a distributed implementation of the architecture. In the future, we would like to explore this hypothesis further, in particular with respect to the abstractions produced by SPATULA. We would like to determine if there is any inherent slow-down attendant with the use of SPATULA (and if so, what changes to the method address this effect), or if the slow-down may be considered an implementation issue rather than a conceptual one.

9.6.1.2 Knowledge Acquisition in Knowledge-Poor Domains

All of our empirical tests were made in domains with fairly strong search control knowledge; although the system still had to search to determine subgoal orderings and to choose among different ways of accomplishing a subgoal, it had knowledge about which operators were likely to be useful with respect to achieving a subgoal. In domains for which the system has not yet acquired knowledge correlating operator actions and subgoal achievement, search is likely to be much more intractable.

With SPATULA, abstract plans are easier to acquire than non-abstract and are applicable in a wider range of new situations. Thus, as suggested by the experiments described in Section 4.3.2, it may be possible to use SPATULA to "bootstrap" the system in new domains for which it has very little search control. Using abstraction, the system could more easily obtain a rough idea of which operations accomplish which desired results, and thus more quickly begin to take relevant actions in the new domain. Once the system has acquired its rough action model, the refinement of the model would be expected to be more tractable.

9.6.1.3 Detection of Over-General Search Control

In Soar, a control decision *conflict* will be generated whenever one search-control rule states that one operator is more useful than another, and a second rule states the reverse[2]. Using abstraction, such situations are most likely to have occurred because the rules were learned in situations fairly different from each other, but were generalized such that they are both applicable in the new situation. The fact that the rules are in conflict suggests that at least one is too general and is not useful. The conflict triggers a new subgoal in which the system re-evaluates the operators in conflict and chooses the one it wants to use. With the re-evaluation, the system has a better chance of making a decision which is useful with respect to the current situation.

It is our hypothesis — supported by our empirical tests — that the greater the range of abstract plans learned in other situations which are available to the system,

[2]More generally, a conflict will be generated for any such cycle of two or more operators.

the greater the system's chance to detect and correct overgeneral abstract search control (rather than to use it unsuspectingly). Additional work is required to explore and confirm this hypothesis.

9.6.2 Least-Commitment Planning

It may prove fruitful to use SPATULA in conjunction with a least-commitment planner. The least-commitment approach can help alleviate the combinatoric expense of goal ordering (since the planner avoids committing to an ordering for as long as possible), while SPATULA can provide heuristic guidance at points for which there is more than one possible way to accomplish a subgoal, and for which the system must therefore still perform search.

9.6.3 Extensions to Method Increments

There are several interesting directions in which the method increments can be extended.

Additional information may be used to adjust and tune the method increments. Probabilistic information may be useful in this adjustment, as discussed in Section 5.5. In addition, the maintenance of such information could be used to predict expected task costs with respect to refinement probability, similarly to the work described in Section 8.6. Such research could involve both the use of a language to talk about cost/accuracy tradeoffs, and the construction by the problem solver of declarative data structures, based on problem-solving experience, to allow it to reason about its possibilities.

The suite of abstraction methods could also be strengthened by providing the problem solver with declarative domain information, and the ability to reason about this information, as discussed in Section 9.5. For example, if the system was provided with explicit representations of its operators, then it could utilize some of the approaches described in Chapter 8 for constructing abstractions via static problem-space analysis, and use this information to guide the dynamic problem-driven abstractions. In turn, dynamic and experiential heuristics about how to construct abstractions may

be useful when the domain does not afford a rigorous analytic construction of abstraction levels. Such a hybrid method, combining the best of both approaches to abstraction, could be very effective.

Further investigation of the extended-plan-use method increment is also warranted. For each point on the plan-use spectrum, it would be useful to characterize the classes of domains for which that point provides a good cost/accuracy tradeoff. Such a characterization may be linked to the research in method-increment parameter settings suggested above. E.g., although use of the abstract sub-search plans produced good results in our experimental domains, it seems likely that the less accurate the abstractions, the more the system may obtain good results by re-evaluating later parts of its plans given updated plan execution information, rather than attempting to use an entire previously generated abstract plan without question. In addition, more extensive research is required to determine the impact of the conservative version of extended plan use, in which the system does not use those sub-search plan fragments learned when options were not distinguishable. Experiments suggest that this mode of plan use has the greatest utility. Further work is also required to address the plan utility issue that arises when using the extended plan use method increment; it is expensive to apply and then retract rules from potentially applicable sub-searches. A more efficient implementation of this method increment may be possible.

Additional method increments may be developed as well. An interesting area of research would again involve providing the problem solver with the means to construct "bookkeeping" information about the abstractions it makes as the abstract search proceeds. Such information could include declarative structures representing the preconditions which were abstracted and the effect of the abstract operator applications on the abstract space. The structures would then provide the system with the ability to perform (possibly problem-driven) analyses of the impact of its abstractions and to learn from these analyses.

Appendix A

Default Rules: Implementing the Abstraction Methods

This appendix lists the Soar default rules (Soar version 5.0.2) used to implement SPATULA. The rules are roughly organized according to function. The rules assume that an operator-subgoaling capability has been provided to the problem-solver, as well as the ability to count and keep track of the task goals achieved, and to calculate and represent domain evaluations. These capabilities are not represented here, since the way in which they are implemented is independent of the abstraction method. However, in some places, the abstraction rules below need hooks into the information provided by these capabilities; these places are indicated in the rule comments. For such rules, certain variable names, etc., are assumed. For example, the rules assume a certain way of representing operator subgoaling information.

Some of the rules contain square brackets around names. This is not Soar syntax, but indicates a "template" rule, which is to be filled in, e.g., with the name of an operator.

A.1 Basic Abstraction Method

A.1.1 Detection of Search Subgoals

```
;;; explicitly notice that evaluation of objects (i.e., lookahead
;;; search) is occurring, by tagging the goal with this information.
(sp find-in-eval-obj-space
    (Goal <g> ^object <sg>)
    (Goal <sg> ^operator <sq>)
    (Operator <sq> ^name evaluate-object  ^role Operator)
    -->
    (Goal <g> ^in-eval-obj-subgoal t)
    )
;;; Copy down from a goal to its subgoal the information that these
;;; goals are within a lookahead search.  In addition, tag the subgoal
;;; with the information that it's not the first (highest-level)
;;; subgoal within the lookahead search (this knowledge is needed for the
;;; iterative abstraction method).
(sp copy-down-eval-obj-tag
    (Goal <g> ^object <sg>)
    (Goal <sg> ^in-eval-obj-subgoal)
    -->
    (Goal <g> ^in-eval-obj-subgoal t)
    (Goal <g> ^not-first-eval-goal true)
    )

;;; Copy down to a new subgoal the tag which says that preconditions may
;;; not be abstracted in the subgoal.  This will always be the case
;;; before the problem-solver generates an evaluation subgoal (before it
;;; begins lookahead). The "not-in-abstr-subgoal" tag is added to the
;;; initial goal.  So, as long as the problem solver only generates
;;; operator subgoals from the initial goal, the tag is copied down.
;;; (Currently, this only works for domains in which there are no operator
;;; implementation subgoals, but it could easily be modified.)
(sp copy-down-non-selection-tag
    (goal <g> ^name operator-subgoal
   ^object <supergoal>)
    (goal <supergoal> ^not-in-abstr-subgoal)
    -->
    (goal <g> ^not-in-abstr-subgoal t))

;;; Detect when, within a lookahead search, operator preconditions should be
;;; abstracted.  This production compares the number of operator-subgoals
;;; currently generated with the "abstract-after-level" variable (the number
;;; of precondition levels to be solved for before preconditions are
;;; abstracted) and determines whether it is time to begin abstracting.  When
;;; using just the 'basic' abstraction method, the "abstract-after-level"
;;; will have been set to 1.  With iterative abstraction, it is incremented
;;; at each iteration.
(sp [ps-domain-name]*propose*evaluate-state*detect-time-to-abstract
    (goal <g> ^problem-space <p> ^state <s>
  )
    (goal <g> - ^not-in-abstr-subgoal)
    (goal <g> ^abstract-after-level <ss-count> ^subgoal-count <count>)
    (count <ss-count> ^num <num>)
    (count <count> ^num >= <num>)
    -->
    (goal <g> ^in-abstr-subgoal t)
    )
;; The rule above must be considered an operator application.
(operator-applications '( robot*propose*evaluate-state*detect-time-to-abstract))
```

A.1.2 Abstracting Preconditions

```
;;; This rule makes an assumption about (i.e., ignores) an unmet operator
;;; precondition, if an impasse is caused when the problem-solver attempts
```

```
;;; to apply the operator.  At the same time it notes that an assumption has
;;; been made.
;;; As currently implemented, a separate such rule is required for each
;;; operator precondition.   The square brackets indicate where
;;; precondition names are substituted.
;;; However, it would be possible to attach to the problem space the
;;; information about each operator's preconditions, and then use one
;;; template rule to grab this information off the problem space.  Note
;;; that the precondition names are not required to be 'meaningful'-- the
;;; system just needs to distinguish between them to count the number of
;;; assumptions made.
;;;  When the rule below fires, a chunk is built which looks like this.
;;;  Next time the operator is selected, this chunk will fire if the
;;;  precondition is not met (the operator's preconditions will have been
;;;  tested before it is selected.)
;;;(sp p121 elaborate
;;;   (goal <g1> ^in-abstr-subgoal t ^problem-space <p1> ^state <d2>
;;;              ^operator <d3> -^[precondition-name]-precond <d3> -^applied <d3>)
;;;   (problem-space <p1> ^name [ps-domain-name])
;;; -{(goal <g1> ^desired-op <d1> ^operator <d3>)
;;;   (operator <d3> ^duplicate-of <d1>)}
;;;   (operator <d3> ^name [operator-name])
;;; -->
;;;   (goal <g1> ^[precondition-name]-precond-abs <d3> &, <d3> +
;;;              ^[precondition-name]-precond <d3> &, <d3> +)
;;;        )
;;;
(Sp [ps-domain-name]*apply*[operator-name]*abs-prec-chk*[precondition-name]
    (Goal <subgoal>  ^impasse no-change ^attribute Operator ^object <g>)
    (Goal <g> ^problem-space <p> ^state <s>
  ;; if abstracting
  ^in-abstr-subgoal
  ^operator <q>)
    ;; if the current operator has not yet applied
    (Goal <g> - ^applied <q>)
    ;; and if the current operator is not the one (or rather a duplicate of
    ;; one) which was subgoaled-on, since don't want to ignore
    ;; preconditions
    - { (Goal <g> ^desired-op <des> ^operator <q>)
    (Operator <q>  ^duplicate-of <des>) }
    (Problem-Space <p> ^name [ps-domain-name])
    (Operator <q> ^name [operator-name])
    ;; and if a precondition of the current operator is not met
    -(Goal <g> ^[precondition-name]-precond <q> )
    -->
    ;; then add the information that the precondition IS met (this lets the
    ;; operator application continue).  In addition, add the information
    ;; that the precondition was abstracted.
    (Goal <g> ^[precondition-name]-precond <q> <q> &)
    (goal <g> ^[precondition-name]-precond-abs <q> + &)
    )

;;; This declaration is necessary for each such rule above.
(operator-applications
 '(
   [ps-domain-name]*apply*[operator-name]*abs-prec-chk*[precondition-name]
   )
 )

;;; For completeness, the following rule sketches out the format of a rule
;;; that would test a precondition of an operator once it has been proposed.
;;; This is done when the operator is proposed. Then, if some preconditions
;;; aren't met, they can be deliberately ignored when the operator is
;;; selected (if abstracting).
(Sp [ps-domain-name]*propose*[operator-name]*check-precond*[precondition-name]
    (Goal <g> ^problem-space <p> ^state <s>
   ^op-set <opss> - ^applied <q>)
    (Problem-Space <p> ^name [ps-domain-name])
    ;; if the operator has been proposed
    (op-set <opss> ^operator <q>)
    (Operator <q> ^name [operator-name] )
    ;; then check one of its preconditions.
```

```
    [various other tests involving the operator and state information]
    -->
    (Goal <g> ^[precondition-name]-precond <q> <q> & )
    )
;;; Check that all preconditions of an operator are met.  There needs to be
;;; one such rule for each operator (again, a more general format could be
;;; used if information was stored on the problem-space about the
;;; precondition names for each operator.)  This rule will apply regardless
;;; of whether or not the preconditions were REALLY met or were ignored
;;; through abstraction.  Then, the domain operator application rules will
;;; check that there is an 'operator-may-apply' tag for that operator
;;; before application can occur.
(Sp [ps-domain-name]*propose*[operator-name]*check-precond*all-preconds
    (Goal <g> ^problem-space <p> ^state <s>
   ^op-set <opss>)
    (op-set <opss> ^operator <q>)
    (Problem-Space <p> ^name [ps-domain-name])
    (Operator <q> ^name [operator-name] )
    (Goal <g> ^[precondition-name-1]-precond <q> )
    (Goal <g> ^[precondition-name-2]-precond <q> )
    -->
    (Goal <g> ^operator-may-apply <q> <q> & )
    )
```

A.2 Method Increment Parameters: Initialization and Management

A.2.1 Task Initialization

```
;;; For the extended plan use method increment to work, the state in the
;;; execution space needs to contain some information for the chunks
;;; learned in the iterative searches.  We would like the chunks to apply
;;; regardless of what abstraction level they were learned at (then the
;;; ones not at the highest level will be retracted).  Ditto for chunks
;;; learned with various values of the 'goal-achievem-req' parameter, etc.
;;; Therefore, info is added at the top level to let the iterative plans
;;; apply.  If the "extended plan use" method increment is not used, then
;;; it is not necessary to add this extra information.
(sp [domain-name]*create-initial-state&desired-state
    (goal <g> ^problem-space <p>
  ^name [top-level-ps-domain-name] ^desired <d>)
    (desired <d> ^name desired-state)
    (problem-space <p> ^name toh-domain)
    -->
    (goal <g> ^state <s> ^abstract-after-level <all> ^subgoal-count <s-all>
  ^goal-achievem-req <gar>)
    (state <s>
   ^g-ach-count <gac>
   [other initial state initialization]
   )
    ;; set the 'abstract-after-level' and 'subgoal-count' numbers to range
    ;; from 1 to the expected highest number of iterations the system will
    ;; reach on this task.  Although this number is arbitrary (unless the
    ;; domain has been carefully analyzed), it is easy
    ;; to set the high end of the range to a number large enough so that
    ;; it's virtually certain that there will not be that many iterations.
    (count <all> ^num 1 + &, 2 + &, 3 + &, 4 + &,
    5 + &, 6 + &, 7 + &, 8 + &, 9 + &, 10 + &)
    (count <s-all> ^num 1 + &, 2 + &, 3 + &, 4 + &,
    5 + &, 6 + &, 7 + &, 8 + &, 9 + &, 10 + &)
    ;; set the 'goal-achievem-req' numbers to range from 1 to the total
    ;; number of goal conjuncts for the task.
    (count <gar> ^num 5 + &, 4 + &, 3 + &, 2 + &, 1 + &)
```

```
    ;; set the 'g-ach-count' numbers to range from 0 to the total number of goal
    ;; conjuncts for the task.
    (count <gac> ^num 5 + &, 4 + &, 3 + &, 2 + &, 1 + &, 0 + &)
    ;; Describe desired state.  If the 'goal achievement iteration' method
    ;; will be used, need to represent the goals separately since need to
    ;; reason about them separately.  The specific way in which this is
    ;; done does not matter.  E.g.:
    (goal <g> ^desired-state <d>)
    (desired <d> ^goal-conjuncts <gc> ^better lower
     )
    ;; list the goal conjuncts
    (goal-conjuncts <gc> ^[goal-type-1] <goal-id-1>...
    )
    ;; describe each goal conjunct
    ([goal-type-1] <goal-id-1> ^<attr1> <value1> ...
   ^goal-name <goal1>)
    ;; note that they are initially false
    (state <s> ^goal-false <goal1> + &, <goal2> + &, ...)
    )

(Sp [ps-domain-name]*create-problem-space
    (Goal <g> ^object nil) ;no supergoal
    -->
    (Goal <g> ^name [ps-domain-name]
   ^desired <d> <d> &  ^problem-space <p>
   )
    (Goal <g> ^desired-op <d-op> <d-op> &
  ;; mark the top-level goal with the information that there can be
  ;; no abstraction in this goal (can never abstract except within
  ;; lookahead search)
  ^not-in-abstr-subgoal t)
    [additional goal and problem-space information]
    (Problem-Space <p> ^name [ps-domain-name]
   ;; add the information that this problem space is a
   ;; 'domain', rather than 'meta', problem space. (A
   ;; 'meta' problem space is the selection space.)
   ^type domain-problem-space)
    ;; augment the problem-space with all possible abstraction levels --
    ;; that is, with numbers up to the highest abstraction iteration the
    ;; system is expected to encounter.  These numbers will be used later
    ;; as 'canonical' abstraction level numbers, by chunks learned during
    ;; iterative abstraction.
    (problem-space <p> ^abstraction-level <co1> + &, <co2> + &, <co3> + &,
   <co4> + &, <co5> + &, <co6> + &, <co7> + &, <co8> + &,
   <co9> + &, <co10> + &)
    (count <co1> ^num 1)
    (count <co2> ^num 2)
    (count <co3> ^num 3)
    (count <co4> ^num 4)
    (count <co5> ^num 5)
    (count <co6> ^num 6)
    (count <co7> ^num 7)
    (count <co8> ^num 8)
    (count <co9> ^num 9)
    (count <co10> ^num 10)
    )
```

A.2.2 Initializing at new goals and copying to subgoals

```
;;; While in the "execution" space, just copy information about subgoal
;;; count from goal to subgoal.  Subgoals are only REALLY counted during
;;; lookahead search-- the information copied here is 'dummy' information,
;;; which exists to allow the chunks built during iterative planning to
;;; fire in the 'execution space', when using the "extended plan use"
;;; method increment.
(sp copy-down-subgoal-count*top-level
    (goal <g>
  ^problem-space <p> ^state <s>
  ^object <supergoal> ^not-in-abstr-subgoal t
  )
    (goal <supergoal> ^subgoal-count <count>)
```

```
    -->
    (goal <g> ^subgoal-count <count> + &)
    )
;;; Given a new subgoal, add to the "subgoal count" if in lookahead search.
;;; 'Subgoal count' keeps track of how many levels of precondition
;;; achievement the problem solver has operator-subgoaled to.  This count
;;; will then be used to tell the system when to start abstracting.
(Sp abstraction*iterate-in-subgoal
    (Goal <g> ^problem-space <p>
   ^object <supergoal> ^state <s>)
    (problem-space <p> - ^name selection)
    ;; only add to the count if not in the "execution space".  In addition,
    ;; don't add to the count for an evaluate-object subgoal, since are
    ;; counting levels of operator-subgoaling before abstracting.
    (goal <g> - ^name implement-evaluate-object
  - ^not-in-abstr-subgoal)
    (Goal <supergoal> ^subgoal-count <count>)
    (COUNT <count> ^num <num>)
    -->
    (Goal <g> ^subgoal-count <count> - )
    (goal <g> ^subgoal-count <ncount> + &)
    (COUNT <ncount> ^num (Compute 1 + <num> ))
    )
;;; At the highest-level operator tie (that is, a tie generated in the
;;; "execution space", indicated by a subgoal-count of 0), want to begin
;;; counting the subgoal levels.
(Sp abstraction*keep-count-in-subgoal*eval-obj*first-eval
    (Goal <g> ^problem-space <p>
   ^object <supergoal>)
    (goal <g> ^name implement-evaluate-object)
    (Goal <supergoal> ^subgoal-count <count>)
    (COUNT <count> ^num 0)
    -->
    (goal <g> ^subgoal-count <count> -)
    (Goal <g> ^subgoal-count <ncount>  )
    (count <ncount> ^num 1)
    )

;;; for any EVALUATE-OBJECT subgoal other than the first (top-level) one,
;;; just copy down the "subgoal-count". (don't increment subgoal-count for
;;; evaluate-object subgoals).
(Sp abstraction*keep-count-in-subgoal*eval-obj
    (Goal <g> ^problem-space <p>
   ^object <sg>)
    (goal <g> ^name implement-evaluate-object)
    (Goal <sg> ^subgoal-count  <count> )
    (COUNT <count> ^num { > 0 <num>})
    -->
    (Goal <g> ^subgoal-count <count>  )
    )

;;; For "selection" subgoals, just copy down the "subgoal-count".
(Sp abstraction*keep-count-in-subgoal*selection-sp
    (Goal <g> ^problem-space <p>
   ^object <sg>)
    (problem-space <p>  ^name selection)
    (Goal <sg> ^subgoal-count <count>
  )
    (COUNT <count> ^num <num>)
    -->
    (Goal <g> ^subgoal-count <count>  + &)
    )

;;; However, don't want to copy the "subgoal-count" if the selection
;;; subgoal is for a top-level operator tie generated in the execution
;;; space. Therefore, reject this information.
;;; Other productions will initialize the subgoal count for the lookahead
;;; search.
(sp abstraction*reject-top-level-subg-count
    (goal <g> ^problem-space <p> ^object <sg> ^subgoal-count <count>)
    (problem-space <p> ^name selection)
    (goal <sg> ^subgoal-count <count> ^not-in-abstr-subgoal)
    -->
```

```
    (goal <g> ^subgoal-count <count> - )
    )

;;; This rule is used with the goal achievement iteration method increment.
;;; If the selection space goal has no information about the number of goals
;;; to achieve, initialize this information.
(sp select*create-state*add-g-ach-req*no-g-ach
   (Goal <g> ^problem-space <p> ^state <s> )
   (Problem-Space <p> ^name selection)
   (Goal <g> - ^goal-achievem-req)
   -->
   (State <s> ^goal-achievem-req <newcount> + &)
   (COUNT <newcount> ^num [initial number of goals to solve for])
   ;; add the information that this goal is for the top selection space in
   ;; the lookahead search.
   (Goal <g> ^top-lkahead-sel-spce t
 )
   )

;;; Initialize the variable used with the iterative abstraction method
;;; increment, which hold the information about how many precondition levels
;;; to solve for in the initial iteration).  Initialize the subgoal count
;;; (which keeps track of how many operator subgoaling levels the search will
;;; generate) as well.
(sp select*create-state*add-abs-after-level*no-abs-after
   (Goal <g> ^problem-space <p> ^state <s> )
   (Problem-Space <p> ^name selection)
   (Goal <g> - ^abstract-after-level)
   -->
   (State <s> ^abstract-after-level <newcount> + &)
   (COUNT <newcount>
  ^num [initial precondition level at which to begin abstracting])
   (Goal <g> ^top-lkahead-sel-spce t ^subgoal-count <scount>)
   ( count  <scount> ^num 0)
   )

;;; Initialize info about assumption counting when a new
;;; evaluate-object subgoal is generated.  This rule would fire at the same
;;; time any initialization occurs for the domain eval.  (A domain evaluation
;;; initialization is shown in comments as an example).
(Sp eval*select-state*role-operator*init-acount
   (Goal <g> ^problem-space <p> ^superoperator <o2> )
   (Operator <o2> ^name evaluate-object ^role Operator ^object <o>
^superproblem-space <p> ^superstate <s>)
   (Goal <g> ^state <snew> + )
   (State <snew> ^duplicate-of <s>)
   -->
   (State <snew> ;^domain-eval <ncount>
      ^assumption-count <ancount>)
   ;;(COUNT <ncount> ^num 0)
   (COUNT <ancount> ^num 0)
   )

;;; copy down to a new subgoal the info about the number of subgoals to
;;; achieve during lookahead search.
(sp copy-down-goal-achievem-req
    (goal <g> ^object <sg> )
    (goal <sg> ^goal-achievem-req <count>)
    -->
    (goal <g> ^goal-achievem-req <count> + &)
    )
;;; Copy down to a new subgoal the info about how many levels of
;;; preconditions to work on before beginning to abstract.
(sp copy-down-abstract-after-level
    (goal <g> ^object <sg>  )
    (goal <sg> ^abstract-after-level <count>)
    -->
    (goal <g> ^abstract-after-level <count> + &)
    )

;;; If a new evaluate-object subgoal is generated for an op, copy down the
;;; "goal achievement count" from the selection space state above (that is,
;;; the number of goals that have been achieved thus far during lookahead).
;;; The number is relative to the initial, or top-level operator tie-- it's
;;; set to 0 then, and added to or subtracted from when appropriate.
```

```
(Sp eval*select-state*role-operator*copy-down-goal-achievem-req
   (Goal <g> ^problem-space <p> ^superoperator <o2>
  ^object <supergoal>)
   (Goal <supergoal> ^problem-space <sp> ^state <ss>)
   (Problem-Space <sp> ^name selection)
   (State <ss> ^goal-achievem-req <acount>)
   (Operator <o2> ^name evaluate-object ^role Operator ^object <o>
^superproblem-space <p> ^superstate <s>)
   -->
   (Goal <g> ^goal-achievem-req <acount> + &)
   )
;;; If a new evaluate-object subgoal is generated, copy down the
;;; "abstract-after-level" information from the selection space state to
;;; the new goal.
(Sp eval*select-state*role-operator*copy-down-abstract-after-level
   (Goal <g> ^problem-space <p> ^superoperator <o2>
  ^object <supergoal>)
   (Goal <supergoal> ^problem-space <sp> ^state <ss>)
   (Problem-Space <sp> ^name selection)
   (State <ss> ^abstract-after-level <acount>)
   (Operator <o2> ^name evaluate-object ^role Operator ^object <o>
^superproblem-space <p> ^superstate <s>)
   -->
   (Goal <g> ^abstract-after-level <acount> + &)
   )
;;; If the selection space goal already has information about goal
;;; achievement iteration (passed from the goal above) then copy this
;;; information to the selection space's state (where it will be used
;;; by the selection space's 'iterate' operator.)
(sp select*create-state*add-g-ach-req*have-g-ach
   (goal <g> ^problem-space <p> ^state <s> )
   (problem-space <p> ^name selection)
   (Goal  <g> ^goal-achievem-req <acount>)
   (count <acount> ^num <anum>)
   -->
   (State <s> ^goal-achievem-req <acount> + &)
   (goal <g> ^not-top-lkahead-sel-spce t)
   )
;;; The same as above, except copy the information used by the iterative
;;; abstraction method increment.
(sp select*create-state*add-abs-after-level*compute*have-abs-after
   (goal <g> ^problem-space <p>
 ^state <s>  - ^not-in-abstr-subgoal)
   (problem-space <p> ^name selection)
   (Goal  <g> ^abstract-after-level <acount>)
   (count <acount> ^num <anum>)
   -->
   (State <s> ^abstract-after-level <acount> + &)
   (goal <g> ^not-top-lkahead-sel-spce t)
   )

;;; If a state has a subgoal-count, add this information to the goal as
;;; well.
(Sp abstraction*pass-new-implementation-subgoal-count-to-goal
    (Goal <g> ^problem-space <p> ^state <s> )
    (Problem-Space <p> ^type domain-problem-space)
    (State <s> ^subgoal-count <count>)
    -->
    (Goal <g>  ^subgoal-count <count>)
    )

;;; This production is necessary because of the "extended plan use" method
;;; increment.  All possible goal achievement counts are added to the
;;; initial goal, so that chunks built at any plan iteration level have the
;;; potential to fire in the "execution" space.  However, once a lookahead
;;; search is begun, we want to get rid of the 'dummy' goal achievement
;;; counts in the search spaces.
(sp reject-top-level-goal-achievem-req
    (goal <g> ^object <sg> ^goal-achievem-req <count>
  ^problem-space <p> ^state <s>)
    (goal <sg> ^goal-achievem-req <count> ^not-in-abstr-subgoal)
    (goal <g> - ^not-in-abstr-subgoal)
    -->
```

```
    (goal <g> ^goal-achievem-req <count> - )
    )
;;; As with the rule above, this rule is necessary because of the "extended
;;; plan use" method increment.  Reject any 'dummy' goal-achievem-req
;;; information which is copied from the execution space to the state in
;;; the selection space.
(sp reject-top-level-goal-achievem-req-state
    (goal <g> ^object <sg>
  ^problem-space <p> ^state <s>)
    (goal <sg> ^goal-achievem-req <count> ^not-in-abstr-subgoal)
    (goal <g> - ^not-in-abstr-subgoal)
    (state <s> ^goal-achievem-req <count>)
    -->
    (state <s> ^goal-achievem-req <count> - )
    )
;;; Again, necessary for the "extended plan use" method.  If a new subgoal
;;; is within lookahead search and is therefore a goal in which
;;; abstraction may occur, then remove the from the subgoal the 'dummy'
;;; "abstract-after-level" information which it obtained from the upper
;;; (execution-space) goal.
(sp reject-top-level-abstract-after-level
    (goal <g> ^object <supergoal> ^abstract-after-level <count>
  ^problem-space <p> ^state <s>)
    (goal <supergoal> ^abstract-after-level <count> ^not-in-abstr-subgoal)
    (goal <g> - ^not-in-abstr-subgoal)
    -->
    (goal <g> ^abstract-after-level <count> - )
    )
;;; This rule is also used because of the "extended plan use" method
;;; increment, and removes execution-space "abstract-after-level" information
;;; from the selection space state in the new subgoal (it will have been
;;; copied to the selection space state from the goal above).
(sp reject-top-level-abstract-after-level-state
    (goal <g> ^object <supergoal>
  ^problem-space <p> ^state <s>)
    (goal <supergoal> ^abstract-after-level <count> ^not-in-abstr-subgoal)
    (goal <g> - ^not-in-abstr-subgoal)
    (state <s> ^abstract-after-level <count>)
    -->
    (state <s> ^abstract-after-level <count> - )
    )
```

A.3 Evaluation Rules

The rules in this section implement the meta-evaluation function used in our experimental tests, which combines information about assumption counts and domain evaluations in lexicographic order. To modify the meta-evaluation, these rules would be modified.

```
;;; excise the following default rule-- it is replaced with several new
;;; rules.
(Excise  eval*equal-eval-indifferent-preference)

;;; If two evaluations both have a zero assumption count and the same
;;; domain evaluation, then they are given indifferent preferences.
;;; This rule assumes that the domain evaluation is numeric, and that the
;;; evaluation has two components: "numeric-value" and "assump-count-num".
(sp equal-eval-indiff*zero-acount
    (goal <g> ^problem-space <p> ^state <s> ^attribute <role> ^object <g2>
  ^impasse tie)
    (problem-space <p> ^name selection)
    (state <s> ^evaluation <e1> { <> <e1> <e2> } )
    (evaluation <e1> ^numeric-value <nv1> ^object <op1>)
    (numeric-value <nv1> ^eval-num <en> ^assump-count-num 0)
    (evaluation <e2> ^numeric-value <nv2> ^object { <> <op1> <op2> } )
    (numeric-value <nv2> ^eval-num <en> ^assump-count-num 0)
    -->
    (goal <g2> ^<role> <op1> = <op2> )
```

```
   )

;;; There are two sets of the following rules to translate evaluations to
;;; preferences: the first set are the rules to create preferences for ties
;;; generated at the top, or execution level, and the second are the rules to
;;; create preferences for ties generated at lower levels.  They differ in
;;; that for the lower levels, information is deliberately included about
;;; what iterations level the preference was created in.  This information is
;;; then used for the "extended plan use" method increment to reason about
;;; which preferences, from which iteration, are most accurate.  In contrast,
;;; the preferences learned for the top-level tie are iteration-independent.
;;; The rules assume that the domain evaluations are numeric;
;;; this should be changed for other evaluation schemes.
;;; Given two evaluations, if one has a lower assumption count, then give it a
;;; better preference.
(Sp eval*prefer-lower-evaluation*assumption-count*top-level
   (goal <g> ^problem-space <p> ^state <s> ^attribute <role> ^object <g2>
 ^impasse tie)
    -{ (goal <g> ^operator <iterate>)
       (operator <iterate> ^name iterate) }
   (Problem-Space <p> ^name selection)
   ;; use this rule for the "top level" tie.
   (goal <g> - ^not-top-lkahead-sel-spce)
   (goal <g2> ^problem-space <p2> ^state <s2> ^{ << desired-state
 desired-op Desired >> <des> } <d>)
   ;; don't want to fire the resulting chunk if the op is already
   ;; selected (at least in soar5.0.2 ...)
   (goal <g2> - ^operator <o2>)
   (goal <g2> ^op-set <op-set>)
   (op-set <op-set> ^operator <o2> { <> <o2> <o1> })
   (State <s> ^evaluation <e1> ^evaluation { <> <e1> <e2> })
   ;; don't add the preference until evals for all ops are obtained.
   (goal <g> - ^need-eval )
   (evaluation <e1> ^object <o1> ^numeric-value <v>
      ^<des> <d>)
   (numeric-value <v> ^assump-count-num <an>)
   (evaluation <e2> ^object { <> <o1> <o2> } ^numeric-value  <v2>
      ^<des> <d>)
   (numeric-value <v2> ^assump-count-num { > <an> <an2> })
   (operator <o1> ^checked-general-count true)
   (operator <o2> ^checked-general-count true)
   -->
   (Goal <g2> ^operator <o2> < <o1> )
   )

;;; The same as the rule above, except it fires at ties below the top-level
;;; tie and adds information about iteration level.
(Sp eval*prefer-lower-evaluation*assumption-count
   (goal <g> ^problem-space <p> ^state <s> ^attribute <role> ^object <g2>
 ^impasse tie)
    -{ (goal <g> ^operator <iterate>)
       (operator <iterate> ^name iterate) }
   (Problem-Space <p> ^name selection)
   ;; This rule is used for ties below the "top-level" tie.
   (goal <g> - ^top-lkahead-sel-spce)
   (state <s> ^abstract-after-level <aacount>)
   (count <aacount> ^num <aanum>)
   (goal <g2> ^problem-space <p2> ^state <s2> ^{ << desired-state
 desired-op Desired >> <des> } <d>)
   (problem-space <p2> ^abstraction-level <naacount>)
   (count <naacount> ^num <aanum>)
   ;; don't want to fire the resulting chunk if the op is already
   ;; selected (at least in soar5.0.2 ...)
   (goal <g2> - ^operator <o2>)
   (goal <g2> ^op-set <op-set>)
   ;;(state <s2> ^g-ach-count <gachcount>)
   (op-set <op-set> ^operator <o2> { <> <o2> <o1> })
   (State <s> ^evaluation <e1> ^evaluation { <> <e1> <e2> })
   (goal <g> - ^need-eval )
   (evaluation <e1> ^object <o1> ^numeric-value <v>
      ^<des> <d>)
   (numeric-value <v> ^assump-count-num <an>)
   (evaluation <e2> ^object { <> <o1> <o2> } ^numeric-value  <v2>
```

```
      ^<des> <d>)
   (numeric-value <v2> ^assump-count-num { > <an> <an2> })
   (operator <o1> ^checked-general-count true)
   (operator <o2> ^checked-general-count true)
   -->
   (Goal <g2> ^operator <o2> < <o1> )
   (state <s2> ^abstraction-level  <naacount> + &)
   )
;;; The rule fires at the top-level tie if two evaluations have the same
;;; assumption counts but different domain evaluations.
;;; This rule assumes that the evaluation is numeric and that lower evals
;;; are better-- the rule can be changed accordingly if that is not the
;;; case.
(Sp eval*prefer-lower-evaluation*top-evaluation
    (Goal <g> ^problem-space <p> ^state <s> ^attribute <role> ^object <g2> ^impasse tie)
    -{ (goal <g> ^operator <iterate>)
       (operator <iterate> ^name iterate) }
    (goal <g> - ^not-top-lkahead-sel-spce)
    (Problem-Space <p> ^name selection)
    (Goal <g2> ^problem-space <p2> ^state <s2> ^{ << desired-state
  desired-op Desired >> <des> } <d>)
    ;; don't want to fire the resulting chunk if the op is already
    ;; selected (at least in soar5.0.2 ...)
    (goal <g2> - ^operator <o2>)
    (goal <g2> ^op-set <op-set>)
    (op-set <op-set> ^operator <o2> { <> <o2> <o1> })
    (State <s> ^evaluation <e1> ^evaluation { <> <e1> <e2> })
    ;; Wait until all operator evals are generated before making
    ;; comparisons.
    (goal <g> - ^need-eval )
    (desired <d> ^better lower)
    (evaluation <e1> ^object <o1> ^numeric-value <v>
^<des> <d>)
    (numeric-value <v> ^assump-count-num <an> ^eval-num <en>)
    (evaluation <e2> ^object { <> <o1> <o2> } ^numeric-value  <v2>
^<des> <d>)
    (numeric-value <v2> ^assump-count-num  <an> ^eval-num { > <en> <en2> })
    (operator <o1> ^checked-general-count true)
    (operator <o2> ^checked-general-count true)
    -->
    (Goal <g2> ^operator <o2> < <o1> )
    )

;;; The same as the rule above, except that this rule fires for the ties
;;; below the top-level tie and adds information about iteration level.
(Sp eval*prefer-lower-evaluation*evaluation
    (Goal <g> ^problem-space <p> ^state <s> ^attribute <role> ^object <g2> ^impasse tie)
    (goal <g> - ^top-lkahead-sel-spce)
    -{ (goal <g> ^operator <iterate>)
    (operator <iterate> ^name iterate) }
    (Problem-Space <p> ^name selection)
    (state <s> ^abstract-after-level <aacount>)
    (count <aacount> ^num <aanum>)
    (Goal <g2> ^problem-space <p2> ^state <s2> ^{ << desired-state
  desired-op Desired >> <des> } <d>)
    (problem-space <p2> ^abstraction-level <naacount>)
    (count <naacount> ^num <aanum>)
    ;; don't want to fire the resulting chunk if the op is already
    ;; selected (at least in soar5.0.2 ...)
    (goal <g2> - ^operator <o2>)
    (goal <g2> ^op-set <op-set>)
    (op-set <op-set> ^operator <o2> { <> <o2> <o1> })
    (State <s> ^evaluation <e1> ^evaluation { <> <e1> <e2> })
    (goal <g> - ^need-eval )
    (desired <d> ^better lower)
    (evaluation <e1> ^object <o1> ^numeric-value <v>
^<des> <d>)
    (numeric-value <v> ^assump-count-num <an> ^eval-num <en>)
    (evaluation <e2> ^object { <> <o1> <o2> } ^numeric-value  <v2>
^<des> <d>)
    (numeric-value <v2> ^assump-count-num  <an> ^eval-num { > <en> <en2> })
    (operator <o1> ^checked-general-count true)
    (operator <o2> ^checked-general-count true)
```

```
    -->
    (Goal <g2> ^operator <o2> < <o1> )
    ;; add information about the abstraction, or iteration, level
    (state <s2> ^abstraction-level <naacount> + &)
    )
;;; This rule is used if two evaluations are equal-- if they both have the
;;; same domain evaluation, and had the same number of assumptions made.
;;; It's only used for ties below the top-level tie, since at the top
;;; level, the system iterates until it can distinguish between the
;;; operators in the tie,  or until no assumptions are made in at least
;;; one of the evaluations.
(Sp eval*equal-eval-indifferent*not-top-lkahead-sel-spce
    (Goal <g> ^problem-space <p> ^state <s> ^attribute <role> ^object <g2>
  ^impasse tie)
    -{ (goal <g> ^operator <iterate>)
       (operator <iterate> ^name iterate) }
    (Problem-Space <p> ^name selection)
    (state <s> ^abstract-after-level <aacount>)
    (count <aacount> ^num <aanum>)
    (Goal <g>  - ^top-lkahead-sel-spce
  - ^need-eval)
    (State <s> ^evaluation <e1> ^evaluation { <> <e1> <e2> })
    (Goal <g2> ^problem-space <p2> ^state <s2> ^{ << Desired-state
  desired-op Desired >> <des> } <d>)
   (problem-space <p2> ^abstraction-level <naacount>)
   (count <naacount> ^num <aanum>)
    (goal <g2> ^op-set <op-set>)
    (evaluation <e1> ^object <x> ^numeric-value <v> ^<des> <d>)
    (numeric-value <v> ^assump-count-num <an> ^eval-num <cn>)
    (Evaluation <e2> ^object <y> ^numeric-value <v2> ^<des> <d>)
    ;; for the evaluations to be the same,  the assumption-count and the
    ;; problem-space evaluation have to be the same.
    (numeric-value <v2> ^assump-count-num <an> ^eval-num <cn>)
    (operator <x> ^checked-general-count true)
    (operator <y> ^checked-general-count true)
    (op-set <op-set> ^operator <x> { <> <x> <y> })
    -->
    (Goal <g2> ^<role> <x> = <y>)
    ;; add information about the abstraction, or iteration, level
    (state <s2> ^abstraction-level <naacount> + &)
    )
```

A.4 Detection of Search Completion

```
;;; ================================================
;;; detection of successful search
;;; ================================================

;;; The following two rules are modifications of the default operator
;;; subgoaling rule which detects that within a lookahead search, a duplicate
;;; of the "desired" (i.e., subgoaled-upon) operator has been able to apply.
;;; The rules create an evaluation which contains both the domain evaluation
;;; and the number of assumptions made.  The rules here assume that domain
;;; evaluation is represented in the selection space in a particular way
;;; (e.g., with ``^domain-eval'').  Thus, the exact form of these two rules
;;; will vary with the operator subgoaling scheme and domain evaluation used
;;; by a domain; the point is that the evaluation built from the search needs
;;; to include the assumption count.
;;; There are two rules here instead of one for chunking purposes-- the
;;; second rule fires if the application of the duplicate "desired" operator
;;; has generated a desired goal state for the task.  (If this is the case,
;;; we would like it to learn from this fact).
(Sp opsub*detect-indirect-opsub-success*1
     (Goal <g-eval> ^problem-space <p> ^state <s> ^desired-op <o>
   ^operator <o-dup>
    ^applied <o-dup> ^superoperator <sq> ^object <sg> ^desired <d>
    - ^op-sub-reject <op-mark>)
     (Operator <o-dup> ^duplicate-of <o>
 )
```

```
     (state <s> ^op-mark <op-mark>)
     (goal <sg> ^problem-space <sp> ^state <ss>)
     (problem-space <sp> ^name selection)
     (operator <sq> ^name evaluate-object ^evaluation <e>)
     (state <s>
    ;; wait until this tag is present-- it indicates that all the
    ;; counts (e.g.  goal conjunct achievement count, assumption
    ;; count) have been updated, and that there has been time to
    ;; check whether or not the current state is the desired goal
    ;; state for the task.
    ^checked-for-desired-state <o-dup>
    ^domain-eval <count> ^assumption-count <acount>
    ;; the current state was not the desired goal state for the task
    - ^found-success )
     (count <count> ^num <count-num>)
     (count <acount> ^num <assump-count-num>)
     -->
     (evaluation <e> ^found-success true ^numeric-value <nv> + &,
 ^desired <d>)
     (numeric-value <nv> ^assump-count-num <assump-count-num>
    ^eval-num <count-num>)
     )

(Sp opsub*detect-indirect-opsub-success*2
     (Goal <g-eval> ^problem-space <p> ^state <s> ^desired-op <o>
    ^applied <o-dup> ^superoperator <sq> ^object <sg>
    ^desired-state <d>
    - ^op-sub-reject <op-mark>)
     (Operator <o-dup> ^duplicate-of <o>
 )
     (state <s> ^op-mark <op-mark>)
     (goal <sg> ^problem-space <sp> ^state <ss>)
     (problem-space <sp> ^name selection)
     (operator <sq> ^name evaluate-object ^evaluation <e>)
     (state <s>
    ^checked-for-desired-state <o-dup>
    ^domain-eval <count> ^assumption-count <acount>
    ;; the current state WAS the desired goal state for the task
    ^found-success <d>)
     (count <count> ^num <count-num>)
     (count <acount> ^num <assump-count-num>)
     -->
     (evaluation <e> ^found-success true ^numeric-value <nv> + &,
 ^desired <d>)
     (numeric-value <nv> ^assump-count-num <assump-count-num>
    ^eval-num <count-num>)
     )

;;; The following two productions detect that all required goal conjuncts
;;; of a task (those listed in "goal-conjuncts") have been completed.
;;; The particular productions here test, as an example, for completion
;;; four-goal-conjunct tasks.  These productions could  be modified to
;;; be made more general and to deal with a variable number of task goal
;;; conjuncts.

;;; This rule checks for completion in the top, or execution, space.  It
;;; doesn't involve the abstraction method, but is included for comparison
;;; with the rule below.
(sp toh*exit-test*detect-desired-state*top
    (goal <g> ^problem-space <p> ^state <s> ^desired-state <d> ^object nil
  ^operator <q> ^applied <q>)
    (problem-space <p> ^name toh-domain)
    (desired <d> ^goal-conjuncts <gc>)
    (goal-conjuncts <gc> ^<goal-a> <xx> ^<goal-b> { <> <xx> <yy> }
    ^<goal-c> { <> <xx> <> <yy> <zz>}
    ^<goal-d> { <> <xx> <> <yy> <> <zz> <ww>}
    )
    (<goal-a> <xx> ^goal-name <gn1>)
    (<goal-b> <yy> ^goal-name <gn2>)
    (<goal-c> <zz> ^goal-name <gn3>)
    (<goal-d> <ww> ^goal-name <gn4>)
    (state <s> ^goal-true <gn1> <gn2> <gn3> <gn4>)
    -->
    (state <s> ^found-success <d> <d> &)
    )
```

```
;;; This rule tests for task completion in the lookahead search spaces.
;;; Again, this rule would be used for a task with four goal conjuncts.  The
;;; exact syntax of the goal conjunct representation is not
;;; important-- however, the bookkeeping checks related to the abstraction
;;; process are necessary.
(sp toh*exit-test*detect-desired-state*lookahead
    (goal <g> ^problem-space <p> ^state <s> ^desired-state <d>
  ^operator <q> ^applied <q>)
    ;; if operator subgoaling, make this test for the highest of the goals
    ;; with state <s>.
    - { (goal <g> ^object <sg>)
    (goal <sg> ^state <s> ^operator ) }
    (problem-space <p> ^name toh-domain)
    ;; check that all bookkeeping needed for the evaluation is done.
    (state <s> - ^update-domain-eval - ^add-to-assump-ct)
    (desired <d> ^goal-conjuncts <gc>)
    (goal-conjuncts <gc> ^<goal-a> <xx> ^<goal-b> { <> <xx> <yy> }
    ^<goal-c> { <> <xx> <> <yy> <zz>}
    ^<goal-d> { <> <xx> <> <yy> <> <zz> <ww>}
    )
    (<goal-a> <xx> ^goal-name <gn1>)
    (<goal-b> <yy> ^goal-name <gn2> )
    (<goal-c> <zz> ^goal-name <gn3>)
    (<goal-d> <ww> ^goal-name <gn4>)
    ;; The following test is a hack due to a Soar5.0.2 problem-- chunks are
    ;; built by tracing from the last "found-success" attribute posted, so
    ;; wait until have checked for success due to the goal achievement
    ;; iteration count (see below) BEFORE checking for success due to
    ;; completion of all goals.  This way, chunks will be learned based on
    ;; the fact that all goals were completed.
    (state <s> ^checked-for-g-ach-success <q>
   ^goal-true <gn1> <gn2> <gn3> <gn4>)
    -->
    (state <s> ^found-success <d> <d> & ^all-goals t)
    (state <s> ^checked-for-desired-state <q>)
    )

;;; Used with the method increment for goal achievement iteration, this
;;; production checks to see if enough goals have been achieved to stop the
;;; search and evaluate.  The production assumes the existence of a domain
;;; state attribute called "g-ach-count", which contains the number of goal
;;; conjuncts achieved thus far in the search.  The rules checks that no
;;; changes to this count are waiting to be made, by checking the existence
;;; of two variables, 'add-to-g-ach-ct' and 'sub-from-g-ach-ct'.  These tests
;;; (and the attribute names) can be changed to fit the needs of a particular
;;; domain; since the code to count the number of goals achieved is not
;;; intrinsically related to abstraction, it is not included here.
(sp g-ach-abstraction*detect-desired-state*2
   (Goal <g> ^problem-space <p> ^state <s> ^desired-state <d>)
   ;; Don't use this method in the "execution" space-- it's only used
   ;; during lookahead.
   (goal <g> - ^not-in-abstr-subgoal)
   (Problem-Space <p> ^type domain-problem-space)
   (Goal <g> ^goal-achievem-req <xxx>)
   (COUNT <xxx> ^num <num>)
   ;; if operator subgoaling, want this rule to fire only at the highest
   ;; operator subgoal which has the state <s>
    - { (goal <g> ^object <sg>)
    (goal <sg> ^state <s> ^operator ) }
   (State <s>
  ;; check that that the "g-ach-count" is up to date
  - ^add-to-g-ach-ct - ^sub-from-g-ach-ct
  ;; check that the domain evaluation and the assumption count have
  ;; been brought up to date as well.
  - ^update-domain-eval - ^add-to-assump-ct)
   ;; if the number of goals achieved is >= the number required to be
   ;; achieved.
   (State <s> ^g-ach-count <gc>)
   (COUNT <gc> ^num >= <num>)
   -->
   (State <s> ^found-success <d> <d> &)
   )

;;; This is a "timing" production-- it fires under the conditions when the
```

```
;;; production above fires, except it doesn't check the g-ach-count.  It is
;;; used to 'time' the firing of the production above which checks for
;;; completion of ALL goals (``toh*exit-test*detect-desired-state*lookahead'')
;;; -- for soar5.0.2 chunking reasons, we would like that rule to fire AFTER
;;; the one directly above has had a chance to fire. (The chunks will be
;;; based on the last, or most recent, reason for detecting lookahead
;;; success.)
(Sp abstraction*checked-for-g-ach-success
    (Goal <g> ^problem-space <p> ^state <s> ^desired-state <d>
   ^operator <q> ^applied <q>)
    (Problem-Space <p> ^type domain-problem-space)
     - { (goal <g> ^object <sg>)
     (goal <sg> ^state <s> ^operator ) }
    (State <s> - ^update-domain-eval - ^add-to-g-ach-ct
   - ^sub-from-g-ach-ct - ^add-to-assump-ct)
    -->
    (State <s> ^checked-for-g-ach-success <q> <q> &)
    )

;;; This is also a timing production; it fires when the system has had time
;;; to check for success both due to achieving N goal conjuncts in the
;;; particular lookahead search (if goal achievement iteration is being
;;; used), and due to achieving all task goal conjuncts.  At this point,
;;; the production following this one, which creates the lookahead search
;;; evaluation, can fire.
(sp abstraction*checked-for-desired-state*lookahead
    (Goal <g> ^problem-space <p> ^state <s> ^desired-state <d>
  ^operator <q> ^applied <q>)
    (Problem-Space <p> ^type domain-problem-space)
    (State <s> - ^add-to-assump-ct - ^update-domain-eval
   ^checked-for-g-ach-success <q>)
    -->
    (State <s> ^checked-for-desired-state <q>)
    )

;;; This rule creates an evaluation for a successful lookahead search.
;;; It assumes that the domain evaluation is numeric.
(Sp abstraction*detect-numeric-success
    (Goal <g> ^problem-space <p> ^state <s> ^operator <q>
  ^superoperator <sq> ^object <sg> ^desired <d>)
    (Problem-Space <p> ^type domain-problem-space)
    (Goal <sg> ^problem-space <sp> ^state <ss>
  )
    (Problem-Space <sp> ^name selection)
    (Operator <sq> ^name evaluate-object ^evaluation <e>)
    (State <s> ^found-success <d>
   ;; see above-- this test is necessary in soar5.0.2.
   ;; The systems waits until all tests for success have been made,
   ;; to ensure that useful learning will occur.
   ^checked-for-desired-state <q>
   ^domain-eval <count> ^assumption-count <acount>)
    (COUNT <count> ^num <count-num>)   ;the domain evaluation
    (COUNT <acount> ^num <assump-count-num>) ;the number of assumptions made.
    -->
    (evaluation <e> ^found-success true
^numeric-value <nv> <nv> &
^desired <d>)
    ;; keep two values separate so that can combine them lexicographically
    ;; when comparing evaluations.
    (numeric-value <nv> ^assump-count-num <assump-count-num> ^eval-num <count-num>)
    )

;;; If success was based on achieving all goals, add this information to
;;; the evaluation.
(Sp abstraction*detect-numeric-success*note-all-goals
    (Goal <g> ^problem-space <p> ^state <s> ^operator <q>
  ^superoperator <sq> ^object <sg> ^desired <d>)
    (Problem-Space <p> ^type domain-problem-space)
    (Goal <sg> ^problem-space <sp> ^state <ss>
  )
    (Problem-Space <sp> ^name selection)
    (Operator <sq> ^name evaluate-object ^evaluation <e>)
    (State <s> ^found-success <d> ^checked-for-desired-state <q>
   ^all-goals t)
    -->
```

```
    (evaluation <e> ^all-goals t)
    )
```

A.5 Method Increments

A.5.1 Iterative abstraction and Goal Achievement Iteration

See also the section on "Detection of Search Completion" (Section A.4), in which the goal achievement iteration method is additionally implemented by declaring that a search has been successful when some number of the task's goal conjuncts have been achieved.

```
;;; This production produces an evaluate-object operator, in the selection
;;; space, for each item (e.g., operator) in a tie.  This production replaces
;;; an original default rule, and is is different from the original in that
;;; here, new evaluate-object operators are proposed EACH TIME the selection
;;; space iterates on abstraction level.
(Sp eval*select-evaluate
   (Goal <g> ^problem-space <p> ^state <s> ^item <x>)
   (Problem-Space <p> ^name selection)
   ;;want to propose new eval ops for each change of the
   ;;'abstract-after-level' (i.e., iteration level).
   (State <s> ^abstract-after-level <count>
  )
   -->
   (Operator <o> ^state <s> ^name evaluate-object ^object <x>)
   (Goal <g> ^operator <o> = )
   (Goal <g> ^operator <o> + ))

;;; The following two rules allow the system to notice that it needs an
;;; evaluation for an operator -- the first rule postulates that an eval is needed
;;; for each operator  (for each abstraction iteration), and the second
;;; retracts the information if that is not true.
(sp eval*notice-need-evals
    (goal <g> ^problem-space <p> ^item <op> ^state <s>)
    (problem-space <p> ^name selection)
    (state <s> ^abstract-after-level <n> )
    -->
    (goal <g> ^need-eval <op> + &)
    )

(sp eval*notice-have-eval
    (goal <g> ^problem-space <p> ^state <s>)
    (problem-space <p> ^name selection)
    (state <s> ^evaluation <e>)
    (evaluation <e> ^object <op> ^<< cutoff numeric-value >> <nv>)
    -->
    (goal <g> ^need-eval <op> - )
    )

;;; Propose a new "iterate" operator for each set of "abstract-after-level"
;;; and "goal-achievem-req" values in the top-level selection space.  This
;;; operator is made worse than the eval-object operators, and if they
;;; have all applied but the impasse has not been resolved, the
;;; iterate operator causes a new abstraction iteration to begin.
(Sp eval*propose*iterate
   (Goal <g> ^problem-space <p> ^state <s>
 ;; if this is the top-level selection space
```

```
  ^top-lkahead-sel-spce t)
   (Problem-Space <p> ^name selection)
   (State <s> ^abstract-after-level <count> ^goal-achievem-req <gcount>)
   -->
   (Goal <g> ^operator <o> + )
   (operator <o> ^abstract-after-level <count> + &, ^goal-achievem-req <gcount> + &)
   (Operator <o> ^name iterate)
   )

;;; The 'save-old-evals' operator saves the operator evaluations from the
;;; previous iteration to the new one.  This can be useful for comparative
;;; purposes.  As it turns out, this information is not used by the current
;;; methods.  However, the productions which implement it are listed here
;;; since they would in fact be used by proposed extensions of the method
;;; increments.
;;; Propose the 'save-old-evals' operator.
(Sp eval*propose*save-old-evals
   (Goal <g> ^problem-space <p> ^state <s>
  ^top-lkahead-sel-spce t)
   (Problem-Space <p> ^name selection)
   (State <s> ^abstract-after-level <count> ^goal-achievem-req <gcount>)
   -->
   (Goal <g> ^operator <o> + )
   (operator <o> ^abstract-after-level <count> + &, ^goal-achievem-req <gcount> + &)
   (Operator <o> ^name save-old-evals)
   )

;;; make the 'save-old-evals' operator worse than the eval-object
;;; operators ...
(Sp eval*preference*save-old-evals*worse
     (Goal <g> ^problem-space <p> ^state <s> ^operator <q> +
    ^operator { <> <q> <q-eval> } +
    )
     (Problem-Space <p> ^name selection)
     (Operator <q> ^name save-old-evals)
     (Operator <q-eval> ^name evaluate-object)
     -->
     (Goal <g> ^operator <q> < <q-eval>)
     )

;;; ...  but better than the iterate operator, so it will be done
;;; before the iterate operation.
(sp eval*preference*save-old-evals*better*iterate
     (Goal <g> ^problem-space <p> ^state <s> ^operator <q> +
    ^operator { <> <q> <iterate> } +
    )
     (Problem-Space <p> ^name selection)
     (Operator <q> ^name save-old-evals)
     (operator <iterate> ^name iterate)
     -->
     (goal <g> ^operator <q> > <iterate>)
     )
;;; the iterate operator is also worse than the eval-object operators.
(Sp eval*preference*iterate*worse
     (Goal <g> ^problem-space <p> ^state <s> ^operator <q> +
    ^operator { <> <q> <q-eval> } +
    ;^op-set <opss>
    )
     (Problem-Space <p> ^name selection)
     (Operator <q> ^name iterate)
     (Operator <q-eval> ^name evaluate-object)
     -->
     (Goal <g> ^operator <q> < <q-eval>)
     )
;; 'reconsider' the 'save-old-evals' operator once it is proposed.
(sp eval*reconsider-save-old-evals
    (goal <g> ^problem-space <p> ^state <s> ^item <x> ^operator <q>
  )
    (problem-space <p> ^name selection)
    (operator <q> ^name save-old-evals)
    -->
    (goal <g> ^operator <q> @)
    )
```

```
;;; In the selection space, if the operator is 'save-old-evals', then
;;; move the evaluations to the "old-evaluations".
(sp eval*apply-evaluate*move-evals-to-old-evals
   (Goal <g> ^problem-space <p> ^operator <o> ^state <s>
 - ^not-in-abstr-subgoal)
   (Problem-Space <p> ^name selection)
   (Operator <o> ^name save-old-evals)
   (State <s> ^evaluation <old-eval>)
   ;; wait until have moved the previous "old evals" from the state.
   (state <s> ^removed-old-evals <o>)
   -->
   (State <s> ^old-evaluation <old-eval> + &)
   (state <s> ^evaluation <old-eval> - )
   (state <s> ^changed-new-to-old-evals <o>)
   (State <s> ^old-e-not-ag t -)
     )

;; when the operator is "save-old-evals", remove the previous "old
;; evals" from the state in preparation to shifting the current evals to
;; the "old evals".
(sp eval*apply-evaluate*remove-old-old-evals
   (Goal <g> ^problem-space <p> ^operator <o> ^state <s>)
   (Problem-Space <p> ^name selection)
   (Operator <o> ^name save-old-evalss)
   (State <s> ^old-evaluation <old-eval>
  - ^removed-old-evals <o>)
   -->
   (State <s> ^old-evaluation <old-eval> -)
   )

;;; This production applies at the same time as the one above, and
;;; indicates that there has been time to remove the previous "old evals"
;;; from the state.
(sp eval*apply-evaluate*remove-old-old-evals*tag
   (Goal <g> ^problem-space <p> ^operator <o> ^state <s>)
   (Problem-Space <p> ^name selection)
   (Operator <o> ^name save-old-evals)
   -->
   (state <s> ^removed-old-evals <o> + &)
   )

;;; This rule is used with the goal achievement iteration method.  If the
;;; operator is 'iterate' (i.e., if iterating), then decrement the
;;; 'goal-achievem-req' count.  Don't want to decrement it if it's already
;;; at '1'.
(Sp eval*apply*g-iterate*decr-goal-ach-req
     (Goal <g> ^problem-space <p> ^state <s> ^operator <q>
   ^object <sg>
   )
     (Problem-Space <p> ^name selection)
     (Operator <q> ^name iterate)
     (Operator <q> ^goal-achievem-req <count>)
     (state <s> ^changed-new-to-old-evals <q>)
     ;;; don't decrement if already one-- always want to achieve at
     ;;; least one goal
     (COUNT <count> ^num { <> 1 <num>} )
     -->
     (State <s> ^goal-achievem-req <ncount> + &)
     (State <s> ^goal-achievem-req <count> - )
     (COUNT <ncount> ^num (compute <num> - 1))
     (Goal <g> ^operator <q> - )
     (goal <g> ^operator <q> @)
     )

;;; similarly, when the operator is 'iterate', increment the
;;; 'abstract-after' level for the iterative abstraction method.
(Sp eval*apply*iterate*incr-abstract-level
     (Goal <g> ^problem-space <p> ^state <s> ^operator <q>
   ^object <sg>
   )
     (Problem-Space <p> ^name selection)
     (Operator <q> ^name iterate)
     (Operator <q> ^abstract-after-level <count>)
     (COUNT <count> ^num <num>)
     -->
     (State <s> ^abstract-after-level <ncount> + &)
```

```
     (State <s> ^abstract-after-level <count> - )
     (COUNT <ncount> ^num (compute 1 + <num>))
     (Goal <g> ^operator <q> - )
     (goal <g> ^operator <q> @)
     )

;;; make sure that the two previous rules are considered operator
;;; applications.
(operator-applications '(eval*apply*iterate*decr-goal-ach-req
 eval*apply*g-iterate*incr-abstract-level))

;;; if there are no items left to eval (all been rejected) then no need to
;;; iterate.
(Sp eval*preference*reject*iterate*no-items-to-eval
   (Goal <g> ^problem-space <p> ^state <s> ^quiescence t
  - ^item)
   (Problem-Space <p> ^name selection)
   (Goal <g> ^operator <o> + )
   (Operator <o> ^name iterate)
   -->
   (Goal <g> ^operator <o> - )
   )
```

A.5.2 Assumption Counting

```
;;; If a precondition was abstracted, note that an assumption was made.
;;; As the rules are presented here, it is necessary to have a
;;; different such rule for each precondition of each operator, but the
;;; process could be generalized.
(Sp [ps-domain-name]*apply*[operator-name]*abs-prec-chk*[precondition-name]*assump-ct
    (Goal <g> ^problem-space <p> ^state <s> ^in-abstr-subgoal
  ;; if precondition was abstracted
  ^operator <q> ^[precondition-name]-precond-abs <q>)
    (state <s> ^op-mark <om>)
    (Goal <g> - ^applied <q>)
    ;; don't make preconds met of operator that was subgoaled on.
    - { (Goal <g> ^desired-op <des> ^operator <q>)
    (Operator <q>  ^duplicate-of <des>) }
    (Problem-Space <p> ^name [ps-domain-name])
    (Operator <q> ^name [operator-name])
    -->
    ;; note that had to make an assumption for the operator.
    ;; The assumption-count tags are given both an acceptable and an
    ;; indiff. preference.  If there are more than one, then one will
    ;; randomly become the value of ^add-to-assump-ct.  Then, once it's
    ;; tallied and rejected, another value will pop into place.
    (State <s> ^add-to-assump-ct [operator-name]-[precondition-name])
    (State <s> ^add-to-assump-ct [operator-name]-[precondition-name] = )
    ;; A hack for soar 5.0.2: keep a list of all the assumption-counts as
    ;; well, since don't want them to be retracted and disappear forever
    ;; before they are made acceptable.  (this supposedly won't be
    ;; necessary in soar5.1).
    (state <s> ^add-to-assump-ct-list [operator-name]-[precondition-name] + & )
    )

(operator-applications
 '(
   [ps-domain-name]*apply*[operator-name]*abs-no-prec-chk*[precondition-name]*assump-ct
   )
 )

;;; Again, a hack necessary for Soar 5.0.2: keep assumption-counts that
;;; haven't been made acceptable yet 'alive' so that they can be selected.
(sp abstraction*propose*assump-ct*keep-accept
    (goal <g> ^problem-space <p> ^state <s>)
    (problem-space <p> ^type domain-problem-space)
     - { (Goal <g> ^object <sg>)
     (Goal <sg> ^state <s> ^operator ) }
    (state <s> ^add-to-assump-ct-list <item>
   - ^add-to-assump-ct <item>)
    -->
```

```
    (state <s> ^add-to-assump-ct <item> <item> = )
    )

;;; For each "add-to-assump-ct" value, add 1 to the assumption-count value
;;; for the current lookahead search.  Note that these rules, as they
;;; stand, don't tally any information about the number of assumptions made
;;; PER OPERATOR, since the "assumption counting" method increment only
;;; uses information about the entire search.  The idea is that this
;;; production will fire once for each "add-to-assump-ct" there is (they
;;; are indifferent to each other, and as each one is tallied, it is
;;; rejected, giving the next one a chance to pop up.)
(Sp [ps-domain-name]*add-to-assumption-count
    (Goal <g> ^problem-space <p> ^state <s> ^operator <q>)
    (Problem-Space <p> ^name [ps-domain-name])
    (Operator <q> ^type [ps-domain-name]-op)
     - { (Goal <g> ^object <sg>)
     (Goal <sg> ^state <s> ^operator ) }
    (State <s> ^add-to-assump-ct <xxx>)
    (State <s> ^assumption-count <count>)
    (COUNT <count> ^num <num>)
    -->
    (State <s> ^assumption-count <newcount> <newcount> &)
    (State <s> ^assumption-count <count> - )
    (COUNT <newcount> ^num (Compute 1 + <num>))
    (State <s> ^add-to-assump-ct <xxx> - )
    (state <s> ^add-to-assump-ct-list <xxx> - )
    )
```

A.5.3 The Abstraction-Gradient Method Increment

```
;;; This production is loaded in when using the abstraction-gradient method increment in
;;; which abstr level is increased as a search progresses.  When an
;;; operator subgoal is resolved, the "abstract-after-level", which determines
;;; how many levels of preconditions are subgoaled upon before abstracting,
;;; is decremented.  This means that the next operator applied in the
;;; search will be more abstract.
(sp opsub*decr-abstraction-level*1
   (goal <subg>
 ^name operator-subgoal ^in-eval-obj-subgoal
 ^object <g> ^problem-space <subp> ^state <subs> )
   (goal <g> ^problem-space <p> ^state <s> ^operator <q> ^applied <q>
 ^abstract-after-level <count> - ^passed-back-abs-info <q>)
   (count <count> ^num <num>)
 -->
   (goal <g> ^abstract-after-level <count> - )
   (goal <g> ^abstract-after-level <ncount>  + &, ^passed-back-abs-info <q> + &,)
   (count <ncount> ^num (compute <num> - 1 ))
   ;;(write1 (crlf) "passed back abstr info for goal" <g> ".")
   )

;;; This production is loaded in for a variation on the method increment
;;; above, and is used in conjunction with the the method increment which does
;;; iteration on goal achievement.  In this variation, the count which
;;; tells the problem solver how many goal conjuncts must be achieved in
;;; the lookahead search ("goal-achievem-req") is incremented under the
;;; same conditions in which the "abstract-after-level" is decremented.  This
;;; "backs off" the search so that now the same conditions exist as in the
;;; previous iteration, and the chunks learned in the previous iteration
;;; can now apply.  (This does not necessarily provide the best
;;; performance!)
(sp opsub*decr-abstraction-levelthen*2
   (goal <subg>
 ^name operator-subgoal ^in-eval-obj-subgoal
               ^object <g> ^problem-space <subp> ^state <subs> )
   (goal <g> ^problem-space <p> ^state <s> ^operator <q> ^applied <q>
 ^goal-achievem-req <count>
 - ^passed-back-abs-info <q>)
   (count <count> ^num <num>)
 -->
```

```
   (goal <g> ^goal-achievem-req <count> - )
   (goal <g> ^goal-achievem-req <ncount>  + &, ^passed-back-abs-info <q> + &,)
   (count <ncount> ^num (compute <num> + 1 ))
   ;;(write1 (crlf) "passed back g-abstr info for goal" <g> ".")
   )
;; ensure that the two rules above are considered "operator applications".
(operator-applications '(opsub*decr-abstraction-level*1
opsub*decr-abstraction-level*2))
```

A.5.4 The Extended Plan Use method increment

```
;;; the following productions implement the use and retraction of the plans
;;; learned during iterative abstract search, at the execution level.  All
;;; plans from all iterations are allowed to fire in the execution space,
;;; and add to the state their "abstraction level".  They fire because
;;; there exists an "abstract-after-level" tag on the goal with their
;;; particular abstraction level.  Then, the new "abstraction-level"
;;; augmentations to the state are compared.  Only the highest-level plans
;;; are retained-- the rest are RETRACTED by retracting the
;;; "abstract-after-level" tag for their level; when
;;; the tag is retracted for a level, the conditions of the plans for that
;;; level are no longer met.
;;; This rules compares the "abstraction-levels" which exist in an
;;; execution-space state.  Retract the corresponding "abstract-after-level"
;;; tags (attached to the goal) for all but the highest level.
(sp top-level-reject-abs-afters
    (goal <g> ^problem-space <p> ^state <s> ^not-in-abstr-subgoal)
    (state <s>  ^abstraction-level <a1> <a2>)
    (goal <g> ^abstract-after-level <abs>)
    (count <a1> ^num <num1>)
    (count <a2> ^num { < <num1> <num2> } )
    -->
    (count <abs> ^num <num2> - )
    )
;;; If an operator generates a tie impasse, retract the
;;; "abstract-after-level" augmentations from the goal which generated the
;;; tie.  Want to keep them removed until the tie is resolved.
(sp retract-abstract-after-level
    (goal <subg> ^object <g> ^quiescence t
  ^impasse tie ^attribute operator)
    (goal <g> ^problem-space <p> ^state <s> - ^operator
  ^not-in-abstr-subgoal)
    (goal <g> ^abstract-after-level <abs>)
    -->
    (goal <g> ^abstract-after-level <abs> - )
    )

;;; When an operator has applied, replace the "abstract-after-level"
;;; augmentations to the goal, so that new abstract plans can fire.
(sp replace*abstract-after-level
    (goal <g> ^problem-space <p> ^state <s> ^operator <o>
  ^applied <o>
  ;; do when in execution space
  ^not-in-abstr-subgoal)
    (goal <g> ^abstract-after-level <count> - ^replaced-aal <o>)
    -->
    (goal <g> ^abstract-after-level <count> -)
    (goal <g> ^abstract-after-level <abs>  ^replaced-aal <o> + &)
    ;; The replacement numbers of the abstract-after-level augmentations
    ;; should match the initialization numbers added originally in the
    ;; production which initialized the initial state, etc. (see section on
    ;; 'Task Initialization' above.)
    (count <abs> ^num 1 + &, 2 + &, 3 + &, 4 + &,
    5 + &, 6 + &, 7 + &, 8 + &, 9 + &, 10 + &)
    )

;;; This rule also replaces the "abstract-after-level" augmentations to the
;;; goal, but this occurs when each new operator is selected, IF there are
;;; no abstract-after-level augmentations present at all.  This means that
```

```
;;; they were removed while working in an operator tie subgoal, and now
;;; that the subgoal is resolved and an operator selected, they need to be
;;; replaced.
(sp add-back-in*abstract-after-level
    (goal <g> ^problem-space <p> ^state <s>  ^operator <o>
  - ^applied <o>
  ;; do this when in execution space
  ^not-in-abstr-subgoal)
    (goal <g> - ^abstract-after-level )
    -->
    (goal <g> ^abstract-after-level <abs> )
    (count <abs> ^num 1 + &, 2 + &, 3 + &, 4 + &,
    5 + &, 6 + &, 7 + &, 8 + &, 9 + &, 10 + &)
    )
;;; each time an operator applies in the execution space, want to remove
;;; the "abstraction level" tags added via previous chunks.  This way, when
;;; new chunks fire, we can again figure out the highest iteration level of
;;; the chunks, and retract the ones not from that level.
(sp remove-from-state*abstraction-levels
    (goal <g> ^problem-space <p> ^state <s> ^operator <o>
  ^applied <o>
  ;; do when in execution space
  ^not-in-abstr-subgoal)
    (goal <g> - ^removed-abs-level <o>
  - ^replaced-aal <o>)
    (state <s> ^abstraction-level <abslevel>)
    -->
    (state <s> ^abstraction-level <abslevel> - )
    (goal <g> ^removed-abs-level <o> + &)
    )

;; ensure that the productions above are considered operator applications.
(operator-applications '(
 replace*abstract-after-level
 remove-from-state*abstraction-levels
 add-back-in*abstract-after-level))
```

A.6 Conflict Resolution

Because *conflicts* can only be resolved by rejecting all but one item in the conflict (in the version of Soar used for this paper), there need to be separate rules for tie resolution and conflict resolution. However, analogously to the rules in Section A.3, these rules look at both domain evaluation and assumption count to make their decision, and implement the same lexicographic ordering.

```
(operator-applications
 '( eval*prefer-lower-evaluation*assumption-count*conflict
     eval*prefer-lower-evaluation*evaluation*conflict
      eval*equal-eval-indifferent*not-first-abs-count*conflict
       robot*eval*equal-eval-indiff*same-evals*acount*conflict
        robot*eval*equal-eval-indiff*same-evals*eval-num*conflict
))

(Sp eval*prefer-lower-evaluation*assumption-count*conflict
   (Goal <g> ^problem-space <p> ^state <s> ^attribute <role> ^object <g2>
 ^impasse conflict ^quiescence t
 )
   (Problem-Space <p> ^name selection)
   (Goal <g2> ^problem-space <p2> ^state <s2> ^{ << desired-state desired-op Desired >> <des> } <d>)
   (State <s> ^evaluation <e1> ^evaluation { <> <e1> <e2> })
   (desired <d> ^better lower)
```

```
   (evaluation <e1> ^object <o1> ^numeric-value <v>
       ;;^<< desired-state desired-op Desired >> <d>
       ^<des> <d>)
   (numeric-value <v> ^acount-num <an>)
   (evaluation <e2> ^object { <> <o1> <o2> } ^numeric-value  <v2>
       ;;^<< desired-state desired-op Desired >> <d>
       ^<des> <d>)
   (numeric-value <v2> ^acount-num { > <an> <an2> })
   -->
   ;;(Goal <g2> ^operator <o2> < <o1> )
   (goal <g2> ^operator <o2> -)
   )

;;; only if the assumption counts are the same, look at the evaluations.
(Sp eval*prefer-lower-evaluation*evaluation*conflict
    (Goal <g> ^problem-space <p> ^state <s> ^attribute <role> ^object <g2> ^impasse conflict
  ^quiescence t
  )
    (Problem-Space <p> ^name selection)
    (Goal <g2> ^problem-space <p2> ^state <s2> ^{ << desired-state desired-op Desired >> <des> } <d>)
    (State <s> ^evaluation <e1> ^evaluation { <> <e1> <e2> })
    (desired <d> ^better lower)
    (evaluation <e1> ^object <o1> ^numeric-value <v>
;;^<< desired-state desired-op Desired >> <d>
       ^<des> <d>)
    (numeric-value <v> ^acount-num <an> ^eval-num <en>)
    (evaluation <e2> ^object { <> <o1> <o2> } ^numeric-value  <v2>
;;^<< desired-state desired-op Desired >> <d>
       ^<des> <d>)
    (numeric-value <v2> ^acount-num  <an> ^eval-num { > <en> <en2> })
    -->
    ;;(Goal <g2> ^operator <o2> < <o1> )
    (goal <g2> ^operator <o2> -)
    )

(Sp eval*equal-eval-indifferent*not-first-abs-count*conflict
   (Goal <g> ^problem-space <p> ^state <s> ^attribute <role> ^object <g2>
 ^impasse conflict ^operator <eval> ^quiescence t)
   (Problem-Space <p> ^name selection)
   (Goal <g> ^not-first-abs-count)
   (operator <eval> ^name evaluate-object ^object <y>)
   (State <s> ^evaluation <e1> ^evaluation { <> <e1> <e2> })
   (Goal <g2> ^problem-space <p2> ^state <s2> ^{ << Desired-state desired-op Desired >> <des> } <d>)
   (evaluation <e1> ^object <x> ^numeric-value <v> ^<des> <d>)
   (numeric-value <v> ^acount-num <an> ^eval-num <cn>)
   (Evaluation <e2> ^object <y> ^numeric-value <v2> ^<des> <d>)
   ;; for the evaluations to be the same,  the assumption-count and the
   ;; problem-space evaluation have to be the same.
   (numeric-value <v2> ^acount-num <an> ^eval-num <cn>)
   -->
   (Goal <g2> ^<role> <y> - )
   )

(sp robot*eval*equal-eval-indiff*zero-acount*conflict
    (goal <g> ^problem-space <p> ^state <s> ^attribute <role> ^object <g2>
  ^impasse conflict ^operator <eval> ^quiescence t)
    (operator <eval> ^name evaluate-object  ^object <op2>)
    (problem-space <p> ^name selection)
    (state <s> ^evaluation <e1> { <> <e1> <e2> } )
    (evaluation <e1> ^numeric-value <nv1> ^object <op1>)
    (numeric-value <nv1> ^eval-num <en> ^acount-num 0)
    (evaluation <e2> ^numeric-value <nv2> ^object { <> <op1> <op2> } )
    (numeric-value <nv2> ^eval-num <en> ^acount-num 0)
    -->
    ;;(goal <g2> ^<role> <op1> = <op2> )
    (goal <g2> ^<role> <op2> -)
    )

(sp robot*eval*equal-eval-indiff*same-evals*acount*conflict
    (goal <g> ^problem-space <p> ^state <s> ^attribute <role> ^object <g2>
  ^impasse conflict ^operator <eval> ^quiescence t)
    (problem-space <p> ^name selection)
    (operator <eval> ^name evaluate-object  ^object <op2>)
    (state <s> ^evaluation <e1> { <> <e1> <e2> }
   ^old-evaluation <oe1> { <> <oe1> <oe2> } )
```

```
    (evaluation <e1> ^numeric-value <nv1> ^object <op1>)
    (numeric-value <nv1> ^eval-num <en> ^acount-num <an>)
    (evaluation <e2> ^numeric-value <nv2> ^object { <> <op1> <op2> } )
    (numeric-value <nv2> ^eval-num <en> ^acount-num <an>)
    (evaluation <oe1> ^numeric-value <onv1> ^object <op1>)
    (numeric-value <onv1> ^eval-num <en> ^acount-num <an>)
    (evaluation <oe2> ^numeric-value <onv2> ^object <op2>)
    (numeric-value <onv2> ^eval-num <en> ^acount-num <an>)
    -{(state <s> ^evaluation <e3> ^old-evaluation <oe3>)
    (evaluation <e3> ^numeric-value <nv3> ^object <op3>)
    (evaluation <oe3> ^numeric-value <onv3> ^object <op3>)
    (numeric-value <nv3> ^acount-num <nv3-a>)
    (numeric-value <onv3> - ^acount-num <nv3-a>)}
    -->
    ;(goal <g2> ^<role> <op1> = <op2> )
    (goal <g2> ^<role> <op2> -)
    )

(sp robot*eval*equal-eval-indiff*same-evals*eval-num*conflict
    (goal <g> ^problem-space <p> ^state <s> ^attribute <role> ^object <g2>
  ^impasse conflict ^operator <eval> ^quiescence t)
    (problem-space <p> ^name selection)
    (operator <eval> ^name evaluate-object  ^object <op2>)
    (state <s> ^evaluation <e1> { <> <e1> <e2> }
   ^old-evaluation <oe1> { <> <oe1> <oe2> } )
    (evaluation <e1> ^numeric-value <nv1> ^object <op1>)
    (numeric-value <nv1> ^eval-num <en> ^acount-num <an>)
    (evaluation <e2> ^numeric-value <nv2> ^object { <> <op1> <op2> } )
    (numeric-value <nv2> ^eval-num <en> ^acount-num <an>)
    (evaluation <oe1> ^numeric-value <onv1> ^object <op1>)
    (numeric-value <onv1> ^eval-num <en> ^acount-num <an>)
    (evaluation <oe2> ^numeric-value <onv2> ^object <op2>)
    (numeric-value <onv2> ^eval-num <en> ^acount-num <an>)
    -{(state <s> ^evaluation <e3> ^old-evaluation <oe3>)
    (evaluation <e3> ^numeric-value <nv3> ^object <op3>)
    (evaluation <oe3> ^numeric-value <onv3> ^object <op3>)
    (numeric-value <nv3> ^acount-num <nv3-a> ^eval-num <nv3-e>)
    (numeric-value <onv3>  ^acount-num <nv3-a> - ^eval-num <nv3-e> )}
    -->
    ;;(goal <g2> ^<role> <op1> = <op2> )
    (goal <g2> ^<role> <op2> -)
    )
```

Appendix B

Eight-Puzzle Domain

The Eight-Puzzle is a sliding tile puzzle in which there are eight tiles and one empty space in a 9x9 grid of 'cells', as shown in Figure B.1. The tiles are numbered, and arbitrarily ordered at the beginning of the task. The task goal is to slide the tiles from cell to cell until they show a given goal configuration. In our tests, such goals were specified by a conjunction of tile/cell pairs.

In our formulation, the EP domain has one operator, which moves a tile from square to square. The operator is instantiated with a tile, and the source and destination squares. The operator has one precondition — the destination square must be blank. The condition that the tiles be adjacent is *critical*, in that it is represented as part of the conditions for proposing the operator. (There is no reason that experiments couldn't be run with this condition explicitly represented as a precondition as well. This is left for future work.) Thus, during abstract search, the tile may be moved to an adjacent square without regard to whether or not there is already a tile on the square. Measuring the number of steps in the shortest such search produces the "Manhattan Distance" heuristic [Pearl, 1983].

The search control for the EP domain prefers operators which move a tile towards its goal spot along either the x- or y-axis of the grid of cells. Of those operators which move a tile towards its goal, search control prefers those for which all preconditions are met. If an operator is selected and can not directly apply because its target cell is occupied, operator subgoaling search control prefers operators which clear the target

5	2	7
8	3	1
6	4	

Figure B.1: Example state in eight-puzzle task. The dark square represents the empty spot in the grid, and the white squares represent tiles.

cell, by moving the tile already on the target cell in any direction. Additional search control prevents operator subgoaling loops: if an operator is proposed to move Tile X to a target cell, and an operator subgoal is generated to achieve this operator, Tile X can not be moved to a cell other than that target cell as part of that operator subgoal.

B.1 Operator Application Rules

Below are listed the Soar rules for applying the EP operator. Rules which add a "reconsider" preference ("@") for an operator signal that the operator application has completed. For a full specification of the syntax of the version of Soar used in this paper (version 5.0.2), see [Laird *et al.*, 1989]. In the following, "binding" relations specify which tile is currently on which cell. In the move-tile operator instantiations, the "tile-cell" is the cell currently containing the tile being moved; the "other-cell" is the target cell; and the "tile-binding" is the binding of the tile being moved. In the state, "tile-cell" contains the target cell of the most recent move; "tile-binding" is the cell binding created by the most recent move; and the "prev-cell" is the cell most recently moved from. The blank is represented as Tile 0.

```
(sp eight*apply*move-tile*change-old-tile-cell-bindings
  (goal <g> ^problem-space <p> ^state <s> ^operator <o> )
  (problem-space <p> ^name eight-puzzle)
  (Goal <g> ^all-preconds-met <o>)
  (state <s> ^binding <b1> )
  (operator <o> ^tile-cell <cell>
```

```
    ^name move-tile  ^tile-binding <b2>)
  (binding <b2> ^tile <t2> ^cell <cell>)
  (binding <b1> ^tile <t2> ^cell <cell>)
  -->
  (state <s> ^binding <b1> - ^prev-cell <cell> + & )
  )
(sp eight*apply*move-tile*change-old-tile-cell-bindings2
  (goal <g> ^problem-space <p> ^state <s> ^operator <o> )
  (problem-space <p> ^name eight-puzzle)
  (Goal <g> ^all-preconds-met <o>)
  (operator <o> ^name move-tile ^other-cell <other-cell> ^tile-binding <b2>)
  (binding <b2> ^tile <t2>)
  -->
  (state <s> ^tile-cell <other-cell> + &, ^tile-binding <b3>+ &,
       ^binding <b3> + & )
  (binding <b3> ^tile <t2> ^cell <other-cell>)
  )
(Sp eight*apply*move-tile*remove-old-tile-binding-info
    (goal <g> ^problem-space <p> ^state <s> ^operator <o>)
    (problem-space <p> ^name eight-puzzle)
    (operator <o> ^name move-tile ^other-cell <other-cell>
      )
    (Goal <g> ^all-preconds-met <o>)
    (State <s> ^tile-binding <some-binding>)
    (binding <some-binding> ^cell <> <other-cell> )
    -->
    (state <s> ^tile-binding <some-binding> - )
    )

(sp eight*apply*move-tile*remove-old-tile-cell-info
  (goal <g> ^problem-space <p> ^state <s> ^operator <o>)
  (problem-space <p> ^name eight-puzzle)
  (operator <o> ^name move-tile ^other-cell <other-cell>)
  (Goal <g> ^all-preconds-met <o>)
  (State <s> ^tile-cell {<> <other-cell> <something>} )
  -->
  (state <s> ^tile-cell <something> -)
  )

(sp eight*apply*move-tile*remove-old-prev-cell-info
  (goal <g> ^problem-space <p> ^state <s> ^operator <o>)
  (problem-space <p> ^name eight-puzzle)
  (operator <o> ^name move-tile ^tile-cell <tile-cell>)
  (Goal <g> ^all-preconds-met <o>)
  (State <s> ^prev-cell {<> <tile-cell> <something>} )
  -->
  (state <s> ^prev-cell <something> -)
  )
;; If a tile is moved onto the blank cell, remove the 'blank' annotation.
(sp eight*apply*move-tile*remove-blank
   (goal <g1> ^state <d2> ^problem-space <p1>)
   (problem-space <p1> ^name eight-puzzle)
     - { (goal <g1> ^object <sg>)
     (goal <sg> ^state <d2>
   ) }
   (state <d2> ^binding <d1> { <> <d1> <b1> })
   (binding <d1> ^cell <c1> ^tile <t1>)
   (tile <t1> ^name 0)
   (binding <b1> ^cell <c1> ^tile { <> <t1> <t2> })
 -->
   (state <d2> ^binding <d1> -))

(sp eight*apply*move-tile*change-tile-cell-bindings*make-blank
  (goal <g> ^problem-space <p> ^state <s> ^operator <o> )
  (problem-space <p> ^name eight-puzzle)
  (operator <o> ^tile-cell <cell>  ^name move-tile )
  (Goal  <g> ^all-preconds-met <o>)
  -->
  (state <s> ^binding <b4> <b4> &)
  ;;; the cell the tile was moved FROM becomes blank.
  (binding <b4> ^tile <t-blank> ^cell <cell>)
  (tile <t-blank> ^name 0)
  )
```

```
(sp eight*reconsider-move-tile-op
     (Goal <g> ^problem-space <p> ^state <s> ^operator <q>)
     (Problem-Space <p> ^name eight-puzzle)
     (Operator <q> ^name move-tile ^tile-cell <cell> ^other-cell <ocell>
       ^tile-binding <tb>)
     (binding <tb> ^tile <tile>)
     (State <s> ^prev-cell <cell> ^tile-cell <ocell>)
     (state <s> ^binding <bind>)
     (binding <bind> ^tile <tile> ^cell <ocell>)
     -->
     (Goal <g> ^operator <q> @ )
     )
```

B.2 EP tasks

For the tests described here, the tasks each required 10 tile movements from initial to goal state. The tasks were randomly generated. The initial and goal states for each task are listed below, using Soar syntax.

```
Tile <t0> represents the blank, or empty, cell.
    ;; task #1, initial state.
    (binding <bb0> ^cell <c11> ^tile <t6>)
    (binding <bb1> ^cell <c12> ^tile <t8>)
    (binding <bb2> ^cell <c13> ^tile <t2>)
    (binding <bb3> ^cell <c21> ^tile <t4>)
    (binding <bb4> ^cell <c22> ^tile <t5>)
    (binding <bb5> ^cell <c23> ^tile <t7>)
    (binding <bb6> ^cell <c31> ^tile <t3>)
    (binding <bb7> ^cell <c32> ^tile <t1>)
    (binding <bb8> ^cell <c33> ^tile <t0>)
    ;; task #1, desired state
    (binding <d2> ^cell <c12> ^tile <t8>)
    (binding <d3> ^cell <c13> ^tile <t7>)
    (binding <d8> ^cell <c21> ^tile <t6>)
    (binding <d0> ^cell <c22> ^tile <t2>)
    (binding <d4> ^cell <c23> ^tile <t5>)
    (binding <d7> ^cell <c31> ^tile <t4>)
    (binding <d6> ^cell <c32> ^tile <t3>)
    (binding <d5> ^cell <c33> ^tile <t1>)
    ;; task #2, initial state.
    (binding <bb0> ^cell <c11> ^tile <t5>)
    (binding <bb1> ^cell <c12> ^tile <t7>)
    (binding <bb2> ^cell <c13> ^tile <t3>)
    (binding <bb3> ^cell <c21> ^tile <t6>)
    (binding <bb4> ^cell <c22> ^tile <t8>)
    (binding <bb5> ^cell <c23> ^tile <t1>)
    (binding <bb6> ^cell <c31> ^tile <t4>)
    (binding <bb7> ^cell <c32> ^tile <t0>)
    (binding <bb8> ^cell <c33> ^tile <t2>)

    ;; task #2, desired state
    (binding <d1> ^cell <c11> ^tile <t7>)
    (binding <d2> ^cell <c12> ^tile <t3>)
    (binding <d3> ^cell <c13> ^tile <t1>)
    (binding <d8> ^cell <c21> ^tile <t5>)
    (binding <d0> ^cell <c22> ^tile <t4>)
    (binding <d7> ^cell <c31> ^tile <t6>)
    (binding <d6> ^cell <c32> ^tile <t2>)
    (binding <d5> ^cell <c33> ^tile <t8>)
```

```
;; task #3, initial state.
(binding <bb0> ^cell <c11> ^tile <t5>)
(binding <bb1> ^cell <c12> ^tile <t0>)
(binding <bb2> ^cell <c13> ^tile <t3>)
(binding <bb3> ^cell <c21> ^tile <t1>)
(binding <bb4> ^cell <c22> ^tile <t7>)
(binding <bb5> ^cell <c23> ^tile <t4>)
(binding <bb6> ^cell <c31> ^tile <t6>)
(binding <bb7> ^cell <c32> ^tile <t8>)
(binding <bb8> ^cell <c33> ^tile <t2>)

;; task #3, desired state
(binding <d1> ^cell <c11> ^tile <t5>)
(binding <d2> ^cell <c12> ^tile <t3>)
(binding <d3> ^cell <c13> ^tile <t4>)
(binding <d8> ^cell <c21> ^tile <t8>)
(binding <d0> ^cell <c22> ^tile <t6>)
(binding <d7> ^cell <c31> ^tile <t1>)
(binding <d6> ^cell <c32> ^tile <t2>)
(binding <d5> ^cell <c33> ^tile <t7>)
;; task #4,  initial state.
(binding <bb0> ^cell <c11> ^tile <t2>)
(binding <bb1> ^cell <c12> ^tile <t3>)
(binding <bb2> ^cell <c13> ^tile <t1>)
(binding <bb3> ^cell <c21> ^tile <t0>)
(binding <bb4> ^cell <c22> ^tile <t6>)
(binding <bb5> ^cell <c23> ^tile <t8>)
(binding <bb6> ^cell <c31> ^tile <t7>)
(binding <bb7> ^cell <c32> ^tile <t5>)
(binding <bb8> ^cell <c33> ^tile <t4>)

;; task #4, desired state
(binding <d1> ^cell <c11> ^tile <t3>)
(binding <d2> ^cell <c12> ^tile <t1>)
(binding <d3> ^cell <c13> ^tile <t8>)
(binding <d8> ^cell <c21> ^tile <t5>)
(binding <d0> ^cell <c22> ^tile <t7>)
(binding <d4> ^cell <c23> ^tile <t6>)
(binding <d7> ^cell <c31> ^tile <t2>)
(binding <d5> ^cell <c33> ^tile <t4>)
;; task #5,  initial state.
(binding <bb0> ^cell <c11> ^tile <t2>)
(binding <bb1> ^cell <c12> ^tile <t7>)
(binding <bb2> ^cell <c13> ^tile <t8>)
(binding <bb3> ^cell <c21> ^tile <t4>)
(binding <bb4> ^cell <c22> ^tile <t6>)
(binding <bb5> ^cell <c23> ^tile <t0>)
(binding <bb6> ^cell <c31> ^tile <t3>)
(binding <bb7> ^cell <c32> ^tile <t5>)
(binding <bb8> ^cell <c33> ^tile <t1>)
;; task #5, desired state
(binding <d1> ^cell <c11> ^tile <t4>)
(binding <d2> ^cell <c12> ^tile <t2>)
(binding <d3> ^cell <c13> ^tile <t7>)
(binding <d8> ^cell <c21> ^tile <t5>)
(binding <d0> ^cell <c22> ^tile <t3>)
(binding <d4> ^cell <c23> ^tile <t8>)
(binding <d7> ^cell <c31> ^tile <t6>)
(binding <d5> ^cell <c33> ^tile <t1>)
;; task #6, initial state.
(binding <bb0> ^cell <c11> ^tile <t2>)
(binding <bb1> ^cell <c12> ^tile <t8>)
(binding <bb2> ^cell <c13> ^tile <t0>)
(binding <bb3> ^cell <c21> ^tile <t6>)
(binding <bb4> ^cell <c22> ^tile <t4>)
(binding <bb5> ^cell <c23> ^tile <t3>)
(binding <bb6> ^cell <c31> ^tile <t7>)
(binding <bb7> ^cell <c32> ^tile <t1>)
(binding <bb8> ^cell <c33> ^tile <t5>)
;; task #6, desired state
```

```
(binding <d1> ^cell <c11> ^tile <t6>)
(binding <d2> ^cell <c12> ^tile <t2>)
(binding <d3> ^cell <c13> ^tile <t3>)
(binding <d8> ^cell <c21> ^tile <t8>)
(binding <d0> ^cell <c22> ^tile <t4>)
(binding <d4> ^cell <c23> ^tile <t5>)
(binding <d6> ^cell <c32> ^tile <t7>)
(binding <d5> ^cell <c33> ^tile <t1>)

;; task #7.  initial state.
(binding <bb0> ^cell <c11> ^tile <t4>)
(binding <bb1> ^cell <c12> ^tile <t2>)
(binding <bb2> ^cell <c13> ^tile <t8>)
(binding <bb3> ^cell <c21> ^tile <t6>)
(binding <bb4> ^cell <c22> ^tile <t7>)
(binding <bb5> ^cell <c23> ^tile <t3>)
(binding <bb6> ^cell <c31> ^tile <t1>)
(binding <bb7> ^cell <c32> ^tile <t0>)
(binding <bb8> ^cell <c33> ^tile <t5>)
;; task #7, desired state
(binding <d1> ^cell <c11> ^tile <t4>)
(binding <d2> ^cell <c12> ^tile <t2>)
(binding <d3> ^cell <c13> ^tile <t8>)
(binding <d8> ^cell <c21> ^tile <t6>)
(binding <d0> ^cell <c22> ^tile <t7>)
(binding <d4> ^cell <c23> ^tile <t1>)
(binding <d7> ^cell <c31> ^tile <t5>)
(binding <d5> ^cell <c33> ^tile <t3>)
;; task #8, initial state.
(binding <bb0> ^cell <c11> ^tile <t3>)
(binding <bb1> ^cell <c12> ^tile <t0>)
(binding <bb2> ^cell <c13> ^tile <t6>)
(binding <bb3> ^cell <c21> ^tile <t1>)
(binding <bb4> ^cell <c22> ^tile <t5>)
(binding <bb5> ^cell <c23> ^tile <t4>)
(binding <bb6> ^cell <c31> ^tile <t2>)
(binding <bb7> ^cell <c32> ^tile <t8>)
(binding <bb8> ^cell <c33> ^tile <t7>)

;; task #8, desired state
(binding <d1> ^cell <c11> ^tile <t3>)
(binding <d2> ^cell <c12> ^tile <t8>)
(binding <d3> ^cell <c13> ^tile <t6>)
(binding <d8> ^cell <c21> ^tile <t1>)
(binding <d0> ^cell <c22> ^tile <t2>)
(binding <d4> ^cell <c23> ^tile <t4>)
(binding <d7> ^cell <c31> ^tile <t5>)
(binding <d5> ^cell <c33> ^tile <t7>)
;; task #9,  initial state.
(binding <bb0> ^cell <c11> ^tile <t2>)
(binding <bb1> ^cell <c12> ^tile <t3>)
(binding <bb2> ^cell <c13> ^tile <t4>)
(binding <bb3> ^cell <c21> ^tile <t1>)
(binding <bb4> ^cell <c22> ^tile <t5>)
(binding <bb5> ^cell <c23> ^tile <t0>)
(binding <bb6> ^cell <c31> ^tile <t7>)
(binding <bb7> ^cell <c32> ^tile <t6>)
(binding <bb8> ^cell <c33> ^tile <t8>)

;; task #9, desired state
(binding <d1> ^cell <c11> ^tile <t2>)
(binding <d2> ^cell <c12> ^tile <t8>)
(binding <d3> ^cell <c13> ^tile <t3>)
(binding <d8> ^cell <c21> ^tile <t1>)
(binding <d0> ^cell <c22> ^tile <t5>)
(binding <d4> ^cell <c23> ^tile <t4>)
(binding <d7> ^cell <c31> ^tile <t7>)
(binding <d5> ^cell <c33> ^tile <t6>)

;; task #10, initial state.
(binding <bb0> ^cell <c11> ^tile <t0>)
```

```
(binding <bb1> ^cell <c12> ^tile <t5>)
(binding <bb2> ^cell <c13> ^tile <t1>)
(binding <bb3> ^cell <c21> ^tile <t8>)
(binding <bb4> ^cell <c22> ^tile <t7>)
(binding <bb5> ^cell <c23> ^tile <t3>)
(binding <bb6> ^cell <c31> ^tile <t4>)
(binding <bb7> ^cell <c32> ^tile <t6>)
(binding <bb8> ^cell <c33> ^tile <t2>)

;; task #10, desired state
(binding <d2> ^cell <c12> ^tile <t5>)
(binding <d3> ^cell <c13> ^tile <t3>)
(binding <d8> ^cell <c21> ^tile <t8>)
(binding <d0> ^cell <c22> ^tile <t1>)
(binding <d4> ^cell <c23> ^tile <t2>)
(binding <d7> ^cell <c31> ^tile <t4>)
(binding <d6> ^cell <c32> ^tile <t7>)
(binding <d5> ^cell <c33> ^tile <t6>)
```

Appendix C

Tower of Hanoi Domain

The Tower of Hanoi puzzle has three pegs, and N disks of increasing size. For our experiments, the 3- and 4-disk versions of the puzzle were used. The rules of the puzzle state that only the top disk on a stack can be moved to another peg, and that a larger disk can not be moved on top of a smaller one. Figure C.1 shows selected states during the solution of a typical TOH task; here the goal is to stack all disks on the third peg.

In the formulation of the TOH domain used here, there is one operator, which moves a disk to a peg. The preconditions of the operator encode the puzzle rules: the disk must be *clear* (no other disks on top of it) and the top disk on the target peg must not be smaller than the disk being moved. Thus, during abstract search when preconditions are ignored, it is possible for a disk to be moved on top of a smaller disk, or for a disk to be moved without clearing out the disks on top of it.

The TOH domain was provided with the following two search control rules. First, if a disk to be moved is not clear, then move the disk on top of it out of the way. Second, if a disk on a peg is smaller than the disk to be moved to that peg, then move the first disk to a different peg (note that this second rule does not specify the order in which the pegs on a disk should be moved off). With this search control, search is still required to determine which potential ordering of disks can create a legal stack. Search control was also provided to prevent operator subgoaling loops: if an operator is proposed to move Disk X to a target peg, and an operator subgoal is generated to

Figure C.1: Selected steps from a typical Tower of Hanoi task.

achieve this operator, Disk X can not be moved to a peg other than that target peg as part of that same operator subgoal.

C.1 Operator Application Rules

Below are listed the operator application rules for the TOH domain. Rules which add a "reconsider" preference ("@") for an operator signal that the operator application has completed. For a full specification of the syntax of the version of Soar used in this paper (version 5.0.2), see [Laird *et al.*, 1989]. In the following, each disk-to-peg operator is instantiated with the disk being moved and its target peg. Included in the states are "on-peg" relations (which specify which peg a disk is on), and "on-disk" relations (which specify that one disk is on top of another).

```
(sp toh*apply*disk-to-peg*remove-on-peg
    (Goal <g> ^problem-space <p> ^state <s> ^operator <q>)
    (Goal <g> ^all-preconds-met <q> - ^applied <q>)
    (Problem-Space <p> ^name toh-domain)
    (Operator <q> ^name disk-to-peg ^disk <disk> ^peg <peg>)
    (State <s> ^on-peg <onp>)
    (on-peg <onp> ^disk <disk> ^peg { <> <peg> <peg2>})
    -->
    (State <s> ^on-peg <onp> - )
    )
(sp toh*apply*disk-to-peg*remove-on-disk*1
    (Goal <g> ^problem-space <p> ^state <s> ^operator <q>)
    (Goal <g> ^all-preconds-met <q> - ^applied <q>)
    (Problem-Space <p> ^name toh-domain)
    (Operator <q> ^name disk-to-peg ^disk <disk> ^peg <peg>)
    (State <s> ^on-disk <ond1> )
    ;; the disk is on another disk
    (on-disk <ond1> ^top-disk <disk> ^bottom-disk <diskb>)
    ;; the disk is not already on the new peg
    -{(state <s> ^on-peg <onp>)
    (on-peg <onp> ^disk <disk> ^peg <peg>)}
    -->
    (State <s> ^on-disk <ond1> - )
    )
(sp toh*apply*disk-to-peg*remove-on-disk*2
    (Goal <g> ^problem-space <p> ^state <s> ^operator <q>)
    (Goal <g> ^all-preconds-met <q> - ^applied <q>)
    (Problem-Space <p> ^name toh-domain)
    (Operator <q> ^name disk-to-peg ^disk <disk>)
    (State <s> ^on-disk <ond1> )
    (on-disk <ond1> ^bottom-disk <disk> ^top-disk <diskb>)
    -->
    (State <s> ^on-disk <ond1> - )
    )
(sp toh*apply*disk-to-peg*add-other-on-disk
    (Goal <g> ^problem-space <p> ^state <s> ^operator <q>)
    (Goal <g> ^all-preconds-met <q> - ^applied <q>)
    (Problem-Space <p> ^name toh-domain)
    (Operator <q> ^name disk-to-peg ^disk <disk>)
    (State <s> ^on-disk <ond1> <ond2>)
    (on-disk <ond1> ^top-disk <disk> ^bottom-disk <diskb>)
    (on-disk <ond2> ^bottom-disk <disk> ^top-disk <diskt>)
    -{(state <s> ^on-peg <onp>)
```

```
    (on-peg <onp> ^disk <disk> ^peg <peg>)}
    -->
    (State <s> ^on-disk <ondnew> + & )
    (on-disk <ondnew> ^top-disk <diskt> ^bottom-disk <diskb>)
    )
(sp toh*apply*disk-to-peg*on-peg
    (Goal <g> ^problem-space <p> ^state <s> ^operator <q>)
    (Goal <g> ^all-preconds-met <q> - ^applied <q>)
    (Problem-Space <p> ^name toh-domain)
    (Operator <q> ^name disk-to-peg ^disk <disk> ^peg <peg>)
    (state <s> ^op-mark <om>)
    -{(state <s> ^on-peg <onpx>)
    (on-peg <onpx> ^disk <disk> ^peg <peg>)}
    -->
    (state <s> ^on-peg <onp> + &)
    (on-peg <onp> ^disk <disk> ^peg <peg>)
    )
(sp toh*apply*disk-to-peg*on-disk
    (Goal <g> ^problem-space <p> ^state <s> ^operator <q>)
    (Goal <g> ^all-preconds-met <q> - ^applied <q>)
    (Problem-Space <p> ^name toh-domain)
    (Operator <q> ^name disk-to-peg ^disk <disk> ^peg <peg>)
    (state <s> ^on-peg <onp>)
    ;; another disk already on the peg
    (on-peg <onp> ^peg <peg> ^disk { <> <disk> <disk2> })
    ;; and it's the top disk on that peg
    -{(state <s> ^on-disk <on>)
    (on-disk <on> ^bottom-disk <disk2>)}
    -{(state <s> ^on-disk <ondx>)
    (on-disk <ondx> ^bottom-disk <disk2> ^top-disk <disk>)}
    -->
    (state <s> ^on-disk <ond2> + &)
    (on-disk <ond2> ^bottom-disk <disk2> ^top-disk <disk>)
    )

(Sp toh*reconsider*disk-to-peg
    (Goal <g> ^problem-space <p> ^state <s> ^operator <q>
   - ^applied <q>)
    (Problem-Space <p> ^name toh-domain)
    (Operator <q> ^name disk-to-peg ^disk <disk> ^peg <peg>)
    (State <s> ^on-peg <onp>)
    (on-peg <onp> ^disk <disk> ^peg <peg>)
    -->
    (Goal <g> ^operator <q> @ )
    )
```

C.2 Tower of Hanoi Tasks

The initial and goal states for each task used in Chp 6 are listed below, using Soar syntax. For each task, the listing first describes the relevant initial state augmentations for the task, and then the task goal. Note that the goal statements describe which disk should end on which peg, but do not specify an ordering of disks on pegs.

```
;;; Task 1
    ;; initial state augmentations:
    (on-peg <onp1> ^disk <disk1> ^peg <peg1>)
    (on-peg <onp2> ^disk <disk2> ^peg <peg1>)
    (on-peg <onp3> ^disk <disk3> ^peg <peg1>)
    (on-peg <onp4> ^disk <disk4> ^peg <peg1>)
    (on-disk <ond3> ^bottom-disk <disk4> ^top-disk <disk3>)
```

```
    (on-disk <ond1> ^bottom-disk <disk3> ^top-disk <disk2>)
    (on-disk <ond2> ^bottom-disk <disk2> ^top-disk <disk1>)
    ;; goal statement
    (goal-conjuncts <gc> ^disk-on-peg <g-onp1> + &, <g-onp2> + &,
    <g-onp3> + &, <g-onp4> + &,)
    (disk-on-peg <g-onp1> ^disk <disk1> ^peg <peg3>
 ^args two ^goal-name <gname1>)
    (disk-on-peg <g-onp2> ^disk <disk2> ^peg <peg3>
 ^args two ^goal-name <gname2>)
    (disk-on-peg <g-onp3> ^disk <disk3> ^peg <peg3>
 ^args two ^goal-name <gname3>)
    (disk-on-peg <g-onp4> ^disk <disk4> ^peg <peg3>
 ^args two ^goal-name <gname4>)
;; Task 2
    ;; initial state augmentations:
    (on-peg <onp1> ^disk <disk1> ^peg <peg2>)
    (on-peg <onp2> ^disk <disk2> ^peg <peg1>)
    (on-peg <onp3> ^disk <disk3> ^peg <peg1>)
    (on-peg <onp4> ^disk <disk4> ^peg <peg2>)
    (on-disk <ond2> ^bottom-disk <disk4> ^top-disk <disk1>)
    (on-disk <ond1> ^bottom-disk <disk3> ^top-disk <disk2>)
    ;; goal statement
    (goal-conjuncts <gc> ^disk-on-peg <g-onp1> + &, <g-onp2> + &,
    <g-onp3> + &, <g-onp4> + &,)
    (disk-on-peg <g-onp1> ^disk <disk1> ^peg <peg3>
 ^args two ^goal-name <gname1>)
    (disk-on-peg <g-onp2> ^disk <disk2> ^peg <peg3>
 ^args two ^goal-name <gname2>)
    (disk-on-peg <g-onp3> ^disk <disk3> ^peg <peg3>
 ^args two ^goal-name <gname3>)
    (disk-on-peg <g-onp4> ^disk <disk4> ^peg <peg3>
 ^args two ^goal-name <gname4>)

;;; Task 3
    ;; initial state augmentations:
    (on-peg <onp1> ^disk <disk1> ^peg <peg1>)
    (on-peg <onp2> ^disk <disk2> ^peg <peg3>)
    (on-peg <onp3> ^disk <disk3> ^peg <peg3>)
    (on-peg <onp4> ^disk <disk4> ^peg <peg1>)
    (on-disk <ond2> ^bottom-disk <disk4> ^top-disk <disk1>)
    (on-disk <ond1> ^bottom-disk <disk3> ^top-disk <disk2>)
    ;; goal statement
    (goal-conjuncts <gc> ^disk-on-peg <g-onp1> + &, <g-onp2> + &,
    <g-onp3> + &, <g-onp4> + &,)
    (disk-on-peg <g-onp1> ^disk <disk1> ^peg <peg3>
 ^args two ^goal-name <gname1>)
    (disk-on-peg <g-onp2> ^disk <disk2> ^peg <peg3>
 ^args two ^goal-name <gname2>)
    (disk-on-peg <g-onp3> ^disk <disk3> ^peg <peg3>
 ^args two ^goal-name <gname3>)
    (disk-on-peg <g-onp4> ^disk <disk4> ^peg <peg3>
 ^args two ^goal-name <gname4>)

;; Task 4
    ;; initial state augmentations:
    (on-peg <onp1> ^disk <disk1> ^peg <peg2>)
    (on-peg <onp2> ^disk <disk2> ^peg <peg2>)
    (on-peg <onp3> ^disk <disk3> ^peg <peg1>)
    (on-peg <onp4> ^disk <disk4> ^peg <peg2>)
    (on-disk <ond2> ^bottom-disk <disk4> ^top-disk <disk2>)
    (on-disk <ond1> ^bottom-disk <disk2> ^top-disk <disk1>)
    ;; goal statement
    (goal-conjuncts <gc> ^disk-on-peg <g-onp1> + &, <g-onp2> + &,
    <g-onp3> + &, <g-onp4> + &,)
    (disk-on-peg <g-onp1> ^disk <disk1> ^peg <peg3>
 ^args two ^goal-name <gname1>)
    (disk-on-peg <g-onp2> ^disk <disk2> ^peg <peg3>
 ^args two ^goal-name <gname2>)
    (disk-on-peg <g-onp3> ^disk <disk3> ^peg <peg3>
```

```
^args two ^goal-name <gname3>)
   (disk-on-peg <g-onp4> ^disk <disk4> ^peg <peg3>
^args two ^goal-name <gname4>)
```

Appendix D

Robot Domain

The Robot Domain was developed from the domain used in the original ABStrips work [Sacerdoti, 1974]. In this domain, a simulated robot manipulates boxes within a group of rooms. Some of the rooms are connected by doors which are initially either opened or closed; the robot has the ability to close and open the doors as well. In addition, the robot is able to move to various locations, and is able to move (push) the boxes to specified target locations.

Two room configurations were used– the original ABStrips room configuration, and a more complex configuration. The two configurations were shown in Figures 6.1 and 6.2 of Chapter 6.

A task in the Robot Domain is specified by providing the problem solver with an initial state and a set of goal conjuncts to achieve. The initial state description describes the placement and status of the robot, boxes, and doors. The goal conjuncts describe aspects of the desired state of the task, e.g. "Robot next to Door-A", "Door-B closed", or "Box-A next to Box-B". There may be any number of goal conjuncts (goal disjuncts were not used for these experiments, but nothing in the domain prohibits such goal specifications as well).

Some of the Robot Domain tasks were tested both with and without the use of domain knowledge about goal invariants, manifested as additional goal conjunct(s). For example, in this domain, for a task in which a goal is to move one box (boxA) next to another (boxB), the two boxes will always be in the same room when this

goal is achieved. Thus, the goal conjunct "in-same-room(boxA, boxB)" may always be added to such a task with no loss of generality. In the task listings below, any goal invariants are so marked. Then, in the detailed results of Appendix E, versions of a task which use the additional goal invariant are marked with an "x".

MEA search control was used for our tests. Precondition-testing rules are listed in Section D.2, as are the rules which create task subgoals for unmet preconditions. Triggered by the task subgoals, MEA rules then fire to propose operators whose effects achieve the task subgoals. Knowledge was also provided about how to detect operator subgoaling loops.

D.1 Operator Application Rules

Below are listed the operator application rules for the Robot Domain. The rules which add a "reconsider" preference ("@") for an operator signal that the operator application has completed. For a full specification of the syntax of the version of Soar used in this paper (version 5.0.2), see [Laird *et al.*, 1989].

```
;=========goto-box operator applications:
(sp robot*apply*goto-box*loc
    (Goal <g> ^problem-space <p> ^state <s> ^operator <q>)
    (Goal <g> ^all-preconds-met <q> - ^applied <q>)
    (Problem-Space <p> ^name robot-domain)
    (State <s> ^at <at> )
    (Operator <q> ^name goto-box)
    (at <at> ^obj <robot> )
    (robot <robot> ^type robot)
    -->
    (State <s> ^at <at> - )
    )
(sp robot*apply*goto-box*other-box
    (Goal <g> ^problem-space <p> ^state <s> ^operator <q>)
    (Goal <g> ^all-preconds-met <q> - ^applied <q>)
    (Problem-Space <p> ^name robot-domain)
    (State <s> ^next-to <nt> )
    (next-to <nt> ^obj1 <robot> ^obj2 { <> <box> <box1>}
     - ^obj2 <box>)
    (Operator <q> ^name goto-box ^box <box>)
    (robot <robot> ^type robot)
    (box <box1> ^type box)
    -->
    (State <s> ^next-to <nt> - )
    )
(sp robot*apply*goto-box*door
    (Goal <g> ^problem-space <p> ^state <s> ^operator <q>)
    (Goal <g> ^all-preconds-met <q> - ^applied <q>)
    (Problem-Space <p> ^name robot-domain)
    (State <s> ^next-to <nt> )
    (next-to <nt> ^obj1 <robot> ^obj2 <door1>)
    (Operator <q> ^name goto-box)
    (robot <robot> ^type robot)
    (door <door1> ^type door)
```

```
    -->
    (State <s> ^next-to <nt> - )
    )
(sp robot*apply*goto-box
    (Goal <g> ^problem-space <p> ^state <s> ^operator <q>)
    (Goal <g> ^all-preconds-met <q> - ^applied <q>)
    (Problem-Space <p> ^name robot-domain)
    (Operator <q> ^name goto-box ^box <box>)
    (State <s> ^robot <robot>)
    -{(state <s> ^next-to <snt>)
    (next-to <snt> ^obj1 <robot> ^obj2 <box>)}
    -->
    (State <s> ^next-to <nt> <nt> &)
    (next-to <nt> ^obj1 <robot> ^obj2 <box>)
    )

(Sp robot*reconsider*goto-box
    (Goal <g> ^problem-space <p> ^state <s> ^operator <q>
   - ^applied <q>)
    (Problem-Space <p> ^name robot-domain)
    (Operator <q> ^name goto-box ^box <box>)
    (State <s> ^next-to <nt> ^robot <robot>)
    (next-to <nt> ^obj1 <robot> ^obj2 <box>)
    -->
    (Goal <g> ^operator <q> @ )
    )

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; goto-door operator applications

(sp robot*apply*goto-door
    (Goal <g> ^problem-space <p> ^state <s> ^operator <q>)
    (Goal <g> ^all-preconds-met <q> - ^applied <q>)
    (Problem-Space <p> ^name robot-domain)
    (Operator <q> ^name goto-door ^door <door>)
    (State <s> ^robot <robot>)
    -{(state <s> ^next-to <snt>)
    (next-to <snt> ^obj1 <robot> ^obj2 <door>)}
    -->
    (State <s> ^next-to <nt> <nt> &)
    (next-to <nt> ^obj1 <robot> ^obj2 <door>)
    )

(sp robot*apply*goto-door*loc
    (Goal <g> ^problem-space <p> ^state <s> ^operator <q>)
    (Goal <g> ^all-preconds-met <q> - ^applied <q>)
    (Problem-Space <p> ^name robot-domain)
    (Operator <q> ^name goto-door )
    (State <s> ^at <at>)
    (at <at> ^obj <robot>)
    (robot <robot> ^type robot)
    -->
    (State <s> ^at <at> - )
    )
(sp robot*apply*goto-door*box
    (Goal <g> ^problem-space <p> ^state <s> ^operator <q>)
    (Goal <g> ^all-preconds-met <q> - ^applied <q>)
    (Problem-Space <p> ^name robot-domain)
    (Operator <q> ^name goto-door )
    (State <s> ^next-to <nt>)
    (next-to <nt> ^obj1 <robot> ^obj2 <box1>)
    (robot <robot> ^type robot)
    (box <box1> ^type box)
    -->
    (State <s> ^next-to <nt> - )
    )
(sp robot*apply*goto-door*other-door
    (Goal <g> ^problem-space <p> ^state <s> ^operator <q>)
    (Goal <g> ^all-preconds-met <q> - ^applied <q>)
    (Problem-Space <p> ^name robot-domain)
    (Operator <q> ^name goto-door ^door <door>)
    (State <s> ^next-to <nt>)
    (next-to <nt> ^obj1 <robot> ^obj2 {<> <door> <door1>}
     - ^obj2 <door>)
```

```
    (robot <robot> ^type robot)
    (door <door1> ^type door)
    -->
    (State <s> ^next-to <nt> - )
    )
(Sp robot*reconsider*goto-door
    (Goal <g> ^problem-space <p> ^state <s> ^operator <q>
   - ^applied <q>)
    (Problem-Space <p> ^name robot-domain)
    (Operator <q> ^name goto-door ^door <door>)
    (State <s> ^next-to <nt> ^robot <robot>)
    (next-to <nt> ^obj1 <robot> ^obj2 <door>)
    -->
    (Goal <g> ^operator <q> @ )
    )

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; goto-loc operator application

(sp robot*apply*goto-loc
    (Goal <g> ^problem-space <p> ^state <s> ^operator <q>)
    (Goal <g> ^all-preconds-met <q> - ^applied <q>)
    (Problem-Space <p> ^name robot-domain)
    (Operator <q> ^name goto-loc ^loc <loc>)
    (State <s> ^robot <robot>)
    -{(state <s> ^at <sat>)
    (at <sat> ^obj <robot> ^loc <loc>)}
    -->
    (State <s> ^at <at> <at> &)
    (at <at> ^obj <robot> ^loc <loc>)
    )

(sp robot*apply*goto-loc*other-loc
    (Goal <g> ^problem-space <p> ^state <s> ^operator <q>)
    (Goal <g> ^all-preconds-met <q> - ^applied <q>)
    (Problem-Space <p> ^name robot-domain)
    (Operator <q> ^name goto-loc ^loc <loc>)
    (State <s> ^at <at>)
    (at <at> ^obj <robot> - ^loc <loc>)
    (robot <robot> ^type robot)
    -->
    (State <s> ^at <at> - )
    )
(sp robot*apply*goto-loc*box
    (Goal <g> ^problem-space <p> ^state <s> ^operator <q>)
    (Goal <g> ^all-preconds-met <q> - ^applied <q>)
    (Problem-Space <p> ^name robot-domain)
    (Operator <q> ^name goto-loc )
    (State <s> ^next-to <nt>)
    (next-to <nt> ^obj1 <robot> ^obj2 <box1>)
    (robot <robot> ^type robot>)
    (box <box1> ^type <box1>)
    -->
    (State <s> ^next-to <nt> - )
    )
(sp robot*apply*goto-loc*door
    (Goal <g> ^problem-space <p> ^state <s> ^operator <q>)
    (Goal <g> ^all-preconds-met <q> - ^applied <q>)
    (Problem-Space <p> ^name robot-domain)
    (Operator <q> ^name goto-loc )
    (State <s> ^next-to <nt>)
    (next-to <nt> ^obj1 <robot> ^obj2 <door1>)
    (robot <robot> ^type robot>)
    (door <door1> ^type door)
    -->
    (State <s> ^next-to <nt> - )
    )

(Sp robot*reconsider*goto-loc
    (Goal <g> ^problem-space <p> ^state <s> ^operator <q>
   - ^applied <q>)
    (Problem-Space <p> ^name robot-domain)
    (Operator <q> ^name goto-loc ^loc <loc>)
```

```
    (State <s> ^at <at> ^robot <robot>)
    (at <at> ^obj <robot> ^loc <loc>)
    -->
    (Goal <g> ^operator <q> @ )
    )

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; push-box operator application -- a robot moves
;;; one box to another box. The operator
;;; is instantiated with a ``move-box'' (the box being
;;; moved) and a ``to-box'' (the target box).
(sp robot*apply*push-box*add-next-to*1
    (Goal <g> ^problem-space <p> ^state <s> ^operator <q>)
    (Goal <g> ^all-preconds-met <q> - ^applied <q>)
    (Problem-Space <p> ^name robot-domain)
    (Operator <q> ^name push-box ^move-box <box-x> ^to-box <box-y>)
    (State <s> ^robot <robot>)
    -{(state <s> ^next-to <snt>)
    (next-to <snt> ^obj1 <box-y> ^obj2 <box-x>)}
    -->
    (State <s> ^next-to <nt1> <nt1> &, )
    (next-to <nt1> ^obj1 <box-y> ^obj2 <box-x>)
    )

(sp robot*apply*push-box*add-next-to*2
    (Goal <g> ^problem-space <p> ^state <s> ^operator <q>)
    (Goal <g> ^all-preconds-met <q> - ^applied <q>)
    (Problem-Space <p> ^name robot-domain)
    (Operator <q> ^name push-box ^move-box <box-x> ^to-box <box-y>)
    (State <s> ^robot <robot>)
    -{(state <s> ^next-to <snt>)
    (next-to <snt> ^obj1 <box-x> ^obj2 <box-y>)}
    -->
    (State <s> ^next-to <nt2> <nt2> &,
   )
    (next-to <nt2> ^obj1 <box-x> ^obj2 <box-y>)
    )

(sp robot*apply*push-box*add-next-to*3
    (Goal <g> ^problem-space <p> ^state <s> ^operator <q>)
    (Goal <g> ^all-preconds-met <q> - ^applied <q>)
    (Problem-Space <p> ^name robot-domain)
    (Operator <q> ^name push-box ^move-box <box-x> ^to-box <box-y>)
    (State <s> ^robot <robot>)
    -{(state <s> ^next-to <snt>)
    (next-to <snt> ^obj1 <robot> ^obj2 <box-x>)}
    -->
    (State <s> ^next-to <nt3> <nt3> &
   )
    (next-to <nt3> ^obj1 <robot> ^obj2 <box-x>)
    )


(sp robot*apply*push-box*loc
    (Goal <g> ^problem-space <p> ^state <s> ^operator <q>)
    (Goal <g> ^all-preconds-met <q> - ^applied <q>)
    (Problem-Space <p> ^name robot-domain)
    (Operator <q> ^name push-box )
    (State <s> ^at <at>)
    (at <at> ^obj <robot> )
    (robot <robot> ^type robot)
    -->
    (State <s> ^at <at> - )
    )
(sp robot*apply*push-box*other-box
    (Goal <g> ^problem-space <p> ^state <s> ^operator <q>)
    (Goal <g> ^all-preconds-met <q> - ^applied <q>)
    (Problem-Space <p> ^name robot-domain)
    (Operator <q> ^name push-box ^move-box <box-x>)
    (State <s> ^next-to <nt>)
    (next-to <nt> ^obj1 <robot> ^obj2 {<> <box-x> <box1> }
     - ^obj2 <box-x>)
    (robot <robot> ^type robot)
    (box <box1> ^type box)
    -->
```

```
    (State <s> ^next-to <nt> - )
    )
(sp robot*apply*push-box*robot-door
    (Goal <g> ^problem-space <p> ^state <s> ^operator <q>)
    (Goal <g> ^all-preconds-met <q> - ^applied <q>)
    (Problem-Space <p> ^name robot-domain)
    (Operator <q> ^name push-box )
    (State <s> ^next-to <nt>)
    (next-to <nt> ^obj1 <robot> ^obj2 <door1>)
    (robot <robot> ^type robot)
    (door <door1> ^type door)
    -->
    (State <s> ^next-to <nt> - )
    )
(sp robot*apply*push-box*box-loc
    (Goal <g> ^problem-space <p> ^state <s> ^operator <q>)
    (Goal <g> ^all-preconds-met <q> - ^applied <q>)
    (Problem-Space <p> ^name robot-domain)
    (Operator <q> ^name push-box ^move-box <box-x>)
    (State <s> ^at <at>)
    (at <at> ^obj <box-x> )
    -->
    (State <s> ^at <at> - )
    )
(sp robot*apply*push-box*box-x-nt*1
    (Goal <g> ^problem-space <p> ^state <s> ^operator <q>)
    (Goal <g> ^all-preconds-met <q> - ^applied <q>)
    (Problem-Space <p> ^name robot-domain)
    (Operator <q> ^name push-box ^move-box <box-x> ^to-box <box-y>)
    (State <s> ^next-to <nt>)
    (next-to <nt> ^obj1 <box-x> ^obj2 {<> <box-y> <box3> }
     - ^obj2 <box-y>)
    (box <box3> ^type box )
    -->
    (State <s> ^next-to <nt> - )
    )
(sp robot*apply*push-box*box-x-nt*2
    (Goal <g> ^problem-space <p> ^state <s> ^operator <q>)
    (Goal <g> ^all-preconds-met <q> - ^applied <q>)
    (Problem-Space <p> ^name robot-domain)
    (Operator <q> ^name push-box ^move-box <box-x> ^to-box <box-y>)
    (State <s> ^next-to <nt>)
    (next-to <nt> ^obj1 {<> <box-y> <box3> } - ^obj1 <box-y>
     ^obj2 <box-x> )
    (box <box3> ^type box )
    -->
    (State <s> ^next-to <nt> - )
    )
(sp robot*apply*push-box*box-door
    (Goal <g> ^problem-space <p> ^state <s> ^operator <q>)
    (Goal <g> ^all-preconds-met <q> - ^applied <q>)
    (Problem-Space <p> ^name robot-domain)
    (Operator <q> ^name push-box ^move-box <box-x>)
    (State <s> ^next-to <nt>)
    (next-to <nt> ^obj1 <box-x> ^obj2 <door1>)
    (door <door1> ^type door)
    -->
    (State <s> ^next-to <nt> - )
    )

(Sp robot*reconsider*push-box
    (Goal <g> ^problem-space <p> ^state <s> ^operator <q>
   - ^applied <q>)
    (Problem-Space <p> ^name robot-domain)
    (State <s>  ^robot <robot>)
    (Operator <q> ^name push-box ^move-box <box-x> ^to-box <box-y>)
    (State <s> ^next-to <nt1> { <> <nt1> <nt2> } { <> <nt1> <> <nt2> <nt3> })
    (next-to <nt1> ^obj1 <box-y> ^obj2 <box-x>)
    (next-to <nt2> ^obj1 <box-x> ^obj2 <box-y>)
    (next-to <nt3> ^obj1 <robot> ^obj2 <box-x>)
    -->
    (Goal <g> ^operator <q> @ )
```

```
    )

;;; ;;;;;;;;;;;;;;;;;;;;
;;; push-to-door operator application--- pushes a box to a door.
(sp robot*apply*push-to-door*add-next-to*1
    (Goal <g> ^problem-space <p> ^state <s> ^operator <q>)
    (Goal <g> ^all-preconds-met <q> - ^applied <q>)
    (Problem-Space <p> ^name robot-domain)
    (Operator <q> ^name push-to-door ^box <box> ^door <door>)
    -{(state <s> ^next-to <snt>)
    (next-to <snt> ^obj1 <box> ^obj2 <door>)}
    -->
    (State <s> ^next-to <nt1> <nt1> &,
   )
    (next-to <nt1> ^obj1 <box> ^obj2 <door>)
    )

(sp robot*apply*push-to-door-add-next-to*2
    (Goal <g> ^problem-space <p> ^state <s> ^operator <q>)
    (Goal <g> ^all-preconds-met <q> - ^applied <q>)
    (Problem-Space <p> ^name robot-domain)
    (Operator <q> ^name push-to-door ^box <box>
      )
    (State <s> ^robot <robot>)
    -{(state <s> ^next-to <snt>)
    (next-to <snt> ^obj1 <robot> ^obj2 <box>)}
    -->
    (State <s> ^next-to
   <nt2> <nt2> &
   )
    (next-to <nt2> ^obj1 <robot> ^obj2 <box>)
    )

(sp robot*apply*push-to-door*robot-loc
    (Goal <g> ^problem-space <p> ^state <s> ^operator <q>)
    (Goal <g> ^all-preconds-met <q> - ^applied <q>)
    (Problem-Space <p> ^name robot-domain)
    (Operator <q> ^name push-to-door )
    (State <s> ^at <at>)
    (at <at> ^obj <robot>)
    (robot <robot> ^type robot)
    -->
    (State <s> ^at <at> - )
    )
(sp robot*apply*push-to-door*other-box
    (Goal <g> ^problem-space <p> ^state <s> ^operator <q>)
    (Goal <g> ^all-preconds-met <q> - ^applied <q>)
    (Problem-Space <p> ^name robot-domain)
    (Operator <q> ^name push-to-door ^box <box>)
    (State <s> ^next-to <nt>)
    (next-to <nt> ^obj1 <robot> ^obj2 { <> <box> <box1> }
     - ^obj2 <box>)
    (robot <robot> ^type robot)
    (box <box1> ^type box )
    -->
    (State <s> ^next-to <nt> - )
    )
(sp robot*apply*push-to-door*other-door
    (Goal <g> ^problem-space <p> ^state <s> ^operator <q>)
    (Goal <g> ^all-preconds-met <q> - ^applied <q>)
    (Problem-Space <p> ^name robot-domain)
    (Operator <q> ^name push-to-door )
    (State <s> ^next-to <nt>)
    (next-to <nt> ^obj1 <robot> ^obj2 <door1>)
    (robot <robot> ^type robot)
    (door <door1> ^type door )
    -->
    (State <s> ^next-to <nt> - )
    )
(sp robot*apply*push-to-door*box-loc
    (Goal <g> ^problem-space <p> ^state <s> ^operator <q>)
    (Goal <g> ^all-preconds-met <q> - ^applied <q>)
    (Problem-Space <p> ^name robot-domain)
    (Operator <q> ^name push-to-door ^box <box>)
```

```
    (State <s> ^at <at>)
    (at <at> ^obj <box>)
    -->
    (State <s> ^at <at> - )
    )
(sp robot*apply*push-to-door*box-nt-b*1
    (Goal <g> ^problem-space <p> ^state <s> ^operator <q>)
    (Goal <g> ^all-preconds-met <q> - ^applied <q>)
    (Problem-Space <p> ^name robot-domain)
    (Operator <q> ^name push-to-door ^box <box>)
    (State <s> ^next-to <nt>)
    (next-to <nt> ^obj1 <box> ^obj2 <other-box>)
    (box <other-box> ^type box)
    -->
    (State <s> ^next-to <nt> - )
    )
(sp robot*apply*push-to-door*box-nt-b*2
    (Goal <g> ^problem-space <p> ^state <s> ^operator <q>)
    (Goal <g> ^all-preconds-met <q> - ^applied <q>)
    (Problem-Space <p> ^name robot-domain)
    (Operator <q> ^name push-to-door ^box <box>)
    (State <s> ^next-to <nt>)
    (next-to <nt> ^obj1 <other-box> ^obj2 <box>)
    (box <other-box> ^type box)
    -->
    (State <s> ^next-to <nt> - )
    )
(sp robot*apply*push-to-door*box-other-d
    (Goal <g> ^problem-space <p> ^state <s> ^operator <q>)
    (Goal <g> ^all-preconds-met <q> - ^applied <q>)
    (Problem-Space <p> ^name robot-domain)
    (Operator <q> ^name push-to-door ^box <box> ^door <door>)
    (State <s> ^next-to <nt>)
    (next-to <nt> ^obj1 <box> ^obj2 { <> <door> <door1> }
     - ^obj2 <door>)
    (door <door1> ^type door)
    -->
    (State <s> ^next-to <nt> - )
    )

(Sp robot*reconsider*push-to-door
    (Goal <g> ^problem-space <p> ^state <s> ^operator <q>
   - ^applied <q>)
    (Problem-Space <p> ^name robot-domain)
    (State <s>  ^robot <robot>)
    (Operator <q> ^name push-to-door ^box <box> ^door <door>)
    (State <s> ^next-to <nt1> { <> <nt1> <nt2> } )
    (next-to <nt1> ^obj1 <box> ^obj2 <door>)
    (next-to <nt2> ^obj1 <robot> ^obj2 <box>)
    -->
    (Goal <g> ^operator <q> @ )
    )

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; push-to-loc operator application
(sp robot*apply*push-to-loc*add-at
    (Goal <g> ^problem-space <p> ^state <s> ^operator <q>)
    (Goal <g> ^all-preconds-met <q> - ^applied <q>)
    (Problem-Space <p> ^name robot-domain)
    (Operator <q> ^name push-to-loc ^box <box> ^loc <loc>)
    (State <s> ^robot <robot>)
    -{(state <s> ^at <sat>)
    (at <sat> ^obj <box> ^loc <loc>)}
    -->
    (State <s> ^at <at> <at> & )
    (at <at> ^obj <box> ^loc <loc>)
    )

(sp robot*apply*push-to-loc*add-next-to
    (Goal <g> ^problem-space <p> ^state <s> ^operator <q>)
    (Goal <g> ^all-preconds-met <q> - ^applied <q>)
    (Problem-Space <p> ^name robot-domain)
    (Operator <q> ^name push-to-loc ^box <box>
```

```
      )
    (State <s> ^robot <robot>)
    -{(state <s> ^next-to <snt>)
    (next-to <snt> ^obj1 <robot> ^obj2 <box>)}
    -->
    (State <s> ^next-to <nt1> <nt1> &
   )
    (Next-to <nt1> ^obj1 <robot> ^obj2 <box>)
    )

(Sp robot*reconsider*push-to-loc
    (Goal <g> ^problem-space <p> ^state <s> ^operator <q>
   - ^applied <q>)
    (Problem-Space <p> ^name robot-domain)
    (State <s> ^robot <robot>)
    (Operator <q> ^name push-to-loc ^box <box> ^loc <loc>)
    (State <s> ^next-to <nt1>  ^at <at> )
    (at <at> ^obj <box> ^loc <loc>)
    (Next-to <nt1> ^obj1 <robot> ^obj2 <box>)
    -->
    (Goal <g> ^operator <q> @ )
    )

(sp robot*apply*push-to-loc*other-loc
    (Goal <g> ^problem-space <p> ^state <s> ^operator <q>)
    (Goal <g> ^all-preconds-met <q> - ^applied <q>)
    (Problem-Space <p> ^name robot-domain)
    (Operator <q> ^name push-to-loc )
    (State <s> ^at <at>)
    (at <at> ^obj <robot>)
    (robot <robot> ^type robot)
    -->
    (State <s> ^at <at> - )
    )
(sp robot*apply*push-to-loc*box
    (Goal <g> ^problem-space <p> ^state <s> ^operator <q>)
    (Goal <g> ^all-preconds-met <q> - ^applied <q>)
    (Problem-Space <p> ^name robot-domain)
    (Operator <q> ^name push-to-loc ^box <box>)
    (State <s> ^next-to <nt>)
    (next-to <nt> ^obj1 <robot> ^obj2 { <> <box> <box1> }
     - ^obj2 <box>)
    (robot <robot> ^type robot)
    (box <box1> ^type box)
    -->
    (State <s> ^next-to <nt> - )
    )
(sp robot*apply*push-to-loc*door
    (Goal <g> ^problem-space <p> ^state <s> ^operator <q>)
    (Goal <g> ^all-preconds-met <q> - ^applied <q>)
    (Problem-Space <p> ^name robot-domain)
    (Operator <q> ^name push-to-loc )
    (State <s> ^next-to <nt>)
    (next-to <nt> ^obj1 <robot> ^obj2 <door1>)
    (robot <robot> ^type robot)
    (door <door1> ^type door)
    -->
    (State <s> ^next-to <nt> - )
    )
(sp robot*apply*push-to-loc*box-loc
    (Goal <g> ^problem-space <p> ^state <s> ^operator <q>)
    (Goal <g> ^all-preconds-met <q> - ^applied <q>)
    (Problem-Space <p> ^name robot-domain)
    (Operator <q> ^name push-to-loc ^box <box> ^loc <loc>)
    (State <s> ^at <at>)
    (at <at> ^obj <box> - ^loc <loc>)
    -->
    (State <s> ^at <at> - )
    )

(sp robot*apply*push-to-loc*box-nt-b*1
    (Goal <g> ^problem-space <p> ^state <s> ^operator <q>)
    (Goal <g> ^all-preconds-met <q> - ^applied <q>)
    (Problem-Space <p> ^name robot-domain)
```

```
    (Operator <q> ^name push-to-loc ^box <box>)
    (State <s> ^next-to <nt>)
    (next-to <nt> ^obj1 <box> ^obj2 <box3>)
    (box <box3> ^type box)
    -->
    (State <s> ^next-to <nt> - )
    )
(sp robot*apply*push-to-loc*box-nt-b*2
    (Goal <g> ^problem-space <p> ^state <s> ^operator <q>)
    (Goal <g> ^all-preconds-met <q> - ^applied <q>)
    (Problem-Space <p> ^name robot-domain)
    (Operator <q> ^name push-to-loc ^box <box>)
    (State <s> ^next-to <nt>)
    (next-to <nt> ^obj1 <box3> ^obj2 <box>)
    (box <box3> ^type box)
    -->
    (State <s> ^next-to <nt> - )
    )
(sp robot*apply*push-to-loc*box-nt-d
    (Goal <g> ^problem-space <p> ^state <s> ^operator <q>)
    (Goal <g> ^all-preconds-met <q> - ^applied <q>)
    (Problem-Space <p> ^name robot-domain)
    (Operator <q> ^name push-to-loc ^box <box>)
    (State <s> ^next-to <nt>)
    (next-to <nt> ^obj1 <box> ^obj2 <door2>)
    (door <door2> ^type door)
    -->
    (State <s> ^next-to <nt> - )
    )

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; go-through-door operator application-- moves a robot through
;;; a door.

(sp robot*apply*go-through-door
    (Goal <g> ^problem-space <p> ^state <s> ^operator <q>)
    (Goal <g> ^all-preconds-met <q> - ^applied <q>)
    (Problem-Space <p> ^name robot-domain)
    (Operator <q> ^name go-through-door ^door <door> ^into-room <room-x>)
    (State <s> ^robot <robot>)
    -{(state <s> ^in-room <sinr>)
    (in-room <sinr> ^obj <robot> ^room <room-x>)}
    -->
    (State <s> ^in-room <inr> <inr> &)
    (in-room <inr> ^obj <robot> ^room <room-x>)
    )

(sp robot*apply*go-through-door*loc
    (Goal <g> ^problem-space <p> ^state <s> ^operator <q>)
    (Goal <g> ^all-preconds-met <q> - ^applied <q>)
    (Problem-Space <p> ^name robot-domain)
    (Operator <q> ^name go-through-door )
    (State <s> ^at <at>)
    (at <at> ^obj <robot>)
    (robot <robot> ^type robot)
    -->
    (State <s> ^at <at> - )
    )
(sp robot*apply*go-through-door*box
    (Goal <g> ^problem-space <p> ^state <s> ^operator <q>)
    (Goal <g> ^all-preconds-met <q> - ^applied <q>)
    (Problem-Space <p> ^name robot-domain)
    (Operator <q> ^name go-through-door )
    (State <s> ^next-to <nt>)
    (next-to <nt> ^obj1 <robot> ^obj2 <box>)
    (robot <robot> ^type robot)
    (box <box> ^type box)
    -->
    (State <s> ^next-to <nt> - )
    )
(sp robot*apply*go-through-door*other-door
    (Goal <g> ^problem-space <p> ^state <s> ^operator <q>)
    (Goal <g> ^all-preconds-met <q> - ^applied <q>)
    (Problem-Space <p> ^name robot-domain)
```

```
    (Operator <q> ^name go-through-door )
    (State <s> ^next-to <nt>)
    (next-to <nt> ^obj1 <robot> ^obj2 <door>)
    (robot <robot> ^type robot)
    (door <door> ^type door)
    -->
    (State <s> ^next-to <nt> - )
    )
(sp robot*apply*go-through-door*other-room
    (Goal <g> ^problem-space <p> ^state <s> ^operator <q>)
    (Goal <g> ^all-preconds-met <q> - ^applied <q>)
    (Problem-Space <p> ^name robot-domain)
    (Operator <q> ^name go-through-door ^into-room <room-x>)
    (State <s> ^in-room <inr>)
    (in-room <inr> ^obj <robot> - ^room <room-x>)
    (robot <robot> ^type robot)
    -->
    (State <s> ^in-room <inr> - )
    )

(Sp robot*reconsider*go-through-door
    (Goal <g> ^problem-space <p> ^state <s> ^operator <q>
   - ^applied <q>)
    (Problem-Space <p> ^name robot-domain)
    (Operator <q> ^name go-through-door ^door <door> ^into-room <room-x>)
    (State <s> ^in-room <inr> ^robot <robot>)
    (in-room <inr> ^obj <robot> ^room <room-x>)
    -->
    (Goal <g> ^operator <q> @ )
    )

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; push-through-door operator application--- a robot pushes a box
;;; through a door.

(sp robot*apply*push-through-door*add-next-to
    (Goal <g> ^problem-space <p> ^state <s> ^operator <q>)
    (Goal <g> ^all-preconds-met <q> - ^applied <q>)
    (Problem-Space <p> ^name robot-domain)
    (Operator <q> ^name push-through-door ^box <box>
      )
    (State <s> ^robot <robot>)
    -{(state <s> ^next-to <snt>)
    (next-to <snt> ^obj1 <robot> ^obj2 <box>)}
    -->
    (State <s>
 ^next-to <nt> <nt> &)
    (next-to <nt> ^obj1 <robot> ^obj2 <box>)
    )

(sp robot*apply*push-through-door*add-inr-box
    (Goal <g> ^problem-space <p> ^state <s> ^operator <q>)
    (Goal <g> ^all-preconds-met <q> - ^applied <q>)
    (Problem-Space <p> ^name robot-domain)
    (Operator <q> ^name push-through-door ^box <box>
    ^into-room <room-x>)
    -{(state <s> ^in-room <sinr>)
    (in-room <sinr> ^obj <box> ^room <room-x>)}
    -->
    (State <s> ^in-room <inr> <inr> &, )
    (in-room <inr> ^obj <box> ^room <room-x>)
    )

(sp robot*apply*push-through-door*add-inr-robot
    (Goal <g> ^problem-space <p> ^state <s> ^operator <q>)
    (Goal <g> ^all-preconds-met <q> - ^applied <q>)
    (Problem-Space <p> ^name robot-domain)
    (Operator <q> ^name push-through-door
    ^into-room <room-x>)
    (State <s> ^robot <robot>)
    -{(state <s> ^in-room <sinr>)
    (in-room <sinr> ^obj <robot> ^room <room-x>)}
    -->
    (State <s> ^in-room
```

```
   <inr2> <inr2> &,
   )
    (in-room <inr2> ^obj <robot> ^room <room-x>)
    )

(sp robot*apply*push-through-door*loc
    (Goal <g> ^problem-space <p> ^state <s> ^operator <q>)
    (Goal <g> ^all-preconds-met <q> - ^applied <q>)
    (Problem-Space <p> ^name robot-domain)
    (Operator <q> ^name push-through-door )
    (State <s> ^at <at>)
    (at <at> ^obj <robot> )
    (robot <robot> ^type robot)
    -->
    (State <s> ^at <at> - )
    )
(sp robot*apply*push-through-door*box
    (Goal <g> ^problem-space <p> ^state <s> ^operator <q>)
    (Goal <g> ^all-preconds-met <q> - ^applied <q>)
    (Problem-Space <p> ^name robot-domain)
    (Operator <q> ^name push-through-door ^box <box>)
    (State <s> ^next-to <nt>)
    (next-to <nt> ^obj1 <robot> ^obj2 { <> <box> <box1> }
     - ^obj2 <box>)
    (robot <robot> ^type robot)
    (box <box1> ^type box)
    -->
    (State <s> ^next-to <nt> - )
    )
(sp robot*apply*push-through-door*door
    (Goal <g> ^problem-space <p> ^state <s> ^operator <q>)
    (Goal <g> ^all-preconds-met <q> - ^applied <q>)
    (Problem-Space <p> ^name robot-domain)
    (Operator <q> ^name push-through-door )
    (State <s> ^next-to <nt>)
    (next-to <nt> ^obj1 <robot> ^obj2 <door1>)
    (robot <robot> ^type robot)
    (door <door1> ^type door)
    -->
    (State <s> ^next-to <nt> - )
    )
(sp robot*apply*push-through-door*box-loc
    (Goal <g> ^problem-space <p> ^state <s> ^operator <q>)
    (Goal <g> ^all-preconds-met <q> - ^applied <q>)
    (Problem-Space <p> ^name robot-domain)
    (Operator <q> ^name push-through-door ^box <box>)
    (State <s> ^at <at>)
    (at <at> ^obj <box>)
    -->
    (State <s> ^at <at> - )
    )
(sp robot*apply*push-through-door*box-nt-b*1
    (Goal <g> ^problem-space <p> ^state <s> ^operator <q>)
    (Goal <g> ^all-preconds-met <q> - ^applied <q>)
    (Problem-Space <p> ^name robot-domain)
    (Operator <q> ^name push-through-door ^box <box>)
    (State <s> ^next-to <nt>)
    (next-to <nt> ^obj1 <box> ^obj2 <box3>)
    (box <box3> ^type box)
    -->
    (State <s> ^next-to <nt> - )
    )
(sp robot*apply*push-through-door*box-nt-b*2
    (Goal <g> ^problem-space <p> ^state <s> ^operator <q>)
    (Goal <g> ^all-preconds-met <q> - ^applied <q>)
    (Problem-Space <p> ^name robot-domain)
    (Operator <q> ^name push-through-door ^box <box>)
    (State <s> ^next-to <nt>)
    (next-to <nt> ^obj1 <box3> ^obj2 <box>)
    (box <box3> ^type box)
    -->
    (State <s> ^next-to <nt> - )
```

```
    )
(sp robot*apply*push-through-door*box-nt-d
    (Goal <g> ^problem-space <p> ^state <s> ^operator <q>)
    (Goal <g> ^all-preconds-met <q> - ^applied <q>)
    (Problem-Space <p> ^name robot-domain)
    (Operator <q> ^name push-through-door ^box <box>)
    (State <s> ^next-to <nt>)
    (next-to <nt> ^obj1 <box> ^obj2 <door2>)
    (door <door2> ^type door)
    -->
    (State <s> ^next-to <nt> - )
    )
(sp robot*apply*push-through-door*other-room*robot
    (Goal <g> ^problem-space <p> ^state <s> ^operator <q>)
    (Goal <g> ^all-preconds-met <q> - ^applied <q>)
    (Problem-Space <p> ^name robot-domain)
    (Operator <q> ^name push-through-door ^into-room <room-x>)
    (State <s> ^in-room <inr>)
    (in-room <inr> ^obj <robot> - ^room <room-x>)
    (robot <robot> ^type robot)
    -->
    (State <s> ^in-room <inr> - )
    )
(sp robot*apply*push-through-door*other-room*box
    (Goal <g> ^problem-space <p> ^state <s> ^operator <q>)
    (Goal <g> ^all-preconds-met <q> - ^applied <q>)
    (Problem-Space <p> ^name robot-domain)
    (Operator <q> ^name push-through-door ^box <box> ^into-room <room-x>)
    (State <s> ^in-room <inr>)
    (in-room <inr> ^obj <box> - ^room <room-x>)
    -->
    (State <s> ^in-room <inr> - )
    )

(Sp robot*reconsider*push-through-door
    (Goal <g> ^problem-space <p> ^state <s> ^operator <q>
   - ^applied <q>)
    (Problem-Space <p> ^name robot-domain)
    (State <s>  ^robot <robot>)
    (Operator <q> ^name push-through-door ^box <box> ^door <door> ^into-room <room-x>
      ^from-room <f-room>)
    (State <s> ^in-room <inr> { <> <inr> <inr2> }
 ^next-to <nt>)
    (next-to <nt> ^obj1 <robot> ^obj2 <box>)
    (in-room <inr> ^obj <box> ^room <room-x>)
    (in-room <inr2> ^obj <robot> ^room <room-x>)
    -->
    (Goal <g> ^operator <q> @ )
    )
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; open-door operator application

(sp robot*apply*open-door
    (Goal <g> ^problem-space <p> ^state <s> ^operator <q>)
    (Goal <g> ^all-preconds-met <q> - ^applied <q>)
    (Problem-Space <p> ^name robot-domain)
    (Operator <q> ^name open-door ^door <door>)
    -{(state <s> ^door-status <sds>)
    (door-status <sds> ^door <door> ^status open)}
    -->
    (State <s> ^door-status <ds> <ds> &)
    (door-status <ds> ^door <door> ^status open)
    )

(sp robot*apply*open-door*other-status
    (Goal <g> ^problem-space <p> ^state <s> ^operator <q>)
    (Goal <g> ^all-preconds-met <q> - ^applied <q>)
    (Problem-Space <p> ^name robot-domain)
    (Operator <q> ^name open-door ^door <door>)
    (State <s> ^door-status <ds>)
    (door-status <ds> ^door <door> ^status closed)
    -->
    (State <s> ^door-status <ds> - )
    )
```

```
(Sp robot*reconsider*open-door
    (Goal <g> ^problem-space <p> ^state <s> ^operator <q>
   - ^applied <q>)
    (Problem-Space <p> ^name robot-domain)
    (Operator <q> ^name open-door ^door <door>)
    (State <s> ^door-status <ds>)
    (door-status <ds> ^door <door> ^status open)
    -->
    (Goal <g> ^operator <q> @ )
    )

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; close-door operator application
(sp robot*apply*close-door
    (Goal <g> ^problem-space <p> ^state <s> ^operator <q>)
    (Goal <g> ^all-preconds-met <q> - ^applied <q>)
    (Problem-Space <p> ^name robot-domain)
    (Operator <q> ^name close-door ^door <door>)
    -{(state <s> ^door-status <sds>)
    (door-status <sds> ^door <door> ^status closed)}
    -->
    (State <s> ^door-status <ds> <ds> &)
    (door-status <ds> ^door <door> ^status closed)
    )

(sp robot*apply*close-door*other-status
    (Goal <g> ^problem-space <p> ^state <s> ^operator <q>)
    (Goal <g> ^all-preconds-met <q> - ^applied <q>)
    (Problem-Space <p> ^name robot-domain)
    (Operator <q> ^name close-door ^door <door>)
    (State <s> ^door-status <ds>)
    (door-status <ds> ^door <door> ^status open)
    -->
    (State <s> ^door-status <ds> - )
    )

(Sp robot*reconsider*close-door
    (Goal <g> ^problem-space <p> ^state <s> ^operator <q>
   - ^applied <q>)
    (Problem-Space <p> ^name robot-domain)
    (Operator <q> ^name close-door ^door <door>)
    (State <s> ^door-status <ds>)
    (door-status <ds> ^door <door> ^status closed)
    -->
    (Goal <g> ^operator <q> @ )
    )
```

D.2 Operator precondition-testing rules

Below are the the precondition-testing rules for each operator in the Robot Domain. Not shown are the rules which test that all preconditions of an operator are met, and add the "operator-may-apply" flag for that operator (as described in Section 3.2).

```
;; =========goto-box precond tests
(Sp robot*propose*goto-box*check-precond*type-box
    (Goal <g> ^problem-space <p> ^state <s>
   ^op-set <opss> - ^applied <q>)
    (op-set <opss> ^operator <q>)
    (Problem-Space <p> ^name robot-domain)
    (Operator <q> ^name goto-box ^box <box>)
    (State <s> ^robot <robot> ^in-room <inr1> { <> <inr1> <inr2> } )
    (in-room <inr1> ^obj <robot> ^room <some-room>)
    (in-room <inr2> ^obj <box> ^room <some-room>)
```

```
    -->
    (Goal <g> ^in-same-room-precond <q> <q> & )
    )
;; =========goto-door precond tests
(Sp robot*propose*goto-door*check-precond*in-same-room-door
    (Goal <g> ^problem-space <p> ^state <s>
   ^op-set <opss> - ^applied <q>)
    (op-set <opss> ^operator <q>)
    (Problem-Space <p> ^name robot-domain)
    (Operator <q> ^name goto-door ^door <door>)
    ;;(state <s>  )
    (State <s> ^connects <conn> ^in-room <inr> ^robot <robot>)
    (connects <conn> ^door <door> ^room <some-room>)
    (in-room <inr> ^obj <robot> ^room <some-room>)
    -->
    (Goal <g> ^in-same-room-door-precond <q> <q> & )
    )
;; =========goto-loc precond tests
(Sp robot*propose*goto-loc*check-precond*in-same-room
    (Goal <g> ^problem-space <p> ^state <s>
   ^op-set <opss> - ^applied <q>)
    (op-set <opss> ^operator <q>)
    (Problem-Space <p> ^name robot-domain)
    (Operator <q> ^name goto-loc ^loc <loc>)
    ;;(state <s>  )
    (State <s> ^loc-in-room <lir> ^in-room <inr> ^robot <robot>)
    (loc-in-room <lir> ^loc <loc> ^room <some-room>)
    (in-room <inr> ^obj <robot> ^room <some-room>)
    -->
    (Goal <g> ^in-same-room-precond <q> <q> & )
    )

;; =========push-box precond tests
(Sp robot*propose*push-box*check-precond*in-same-room
    (Goal <g> ^problem-space <p> ^state <s>
   ^op-set <opss> - ^applied <q>)
    (op-set <opss> ^operator <q>)
    (Problem-Space <p> ^name robot-domain)
    (Operator <q> ^name push-box ^move-box <box-x>
      ^to-box <box-y>  )
    ;;(state <s>  )
    (State <s> ^robot <robot> ^in-room <inr1> { <> <inr1> <inr2> } { <> <inr1> <> <inr2> <inr3> } )
    (in-room <inr1> ^obj <box-y> ^room <some-room>)
    (in-room <inr2> ^obj <box-x> ^room <some-room>)
    (in-room <inr3> ^obj <robot> ^room <some-room>)
    -->
    (Goal <g> ^in-same-room-precond <q> <q> & )
    )
(Sp robot*propose*push-box*check-precond*next-to
    (Goal <g> ^problem-space <p> ^state <s>
   ^op-set <opss> - ^applied <q>)
    (op-set <opss> ^operator <q>)
    (Problem-Space <p> ^name robot-domain)
    (Operator <q> ^name push-box ^move-box <box-x>
      ;^to-box <box-y>
      )
    ;;(state <s>  )
    (State <s> ^robot <robot> ^next-to <nt>)
    (next-to <nt> ^obj1 <robot> ^obj2 <box-x>)
    -->
    (Goal <g> ^next-to-precond <q> <q> & )
    )
;; =========push-to-door precond tests
(Sp robot*propose*push-to-door*check-precond*in-same-room-door
    (Goal <g> ^problem-space <p> ^state <s>
   ^op-set <opss> - ^applied <q>)
    (op-set <opss> ^operator <q>)
    (Problem-Space <p> ^name robot-domain)
    (Operator <q> ^name push-to-door ^box <box> ^door <door>)
    ;;(state <s>  )
    (State <s> ^robot <robot> ^connects <conn> ^in-room <inr1> { <> <inr1> <inr2> } )
```

```
    (connects <conn> ^door <door> ^room <some-room>)
    (in-room <inr1> ^obj <robot> ^room <some-room>)
    (in-room <inr2> ^obj <box> ^room <some-room>)
    -->
    (Goal <g> ^in-same-room-door-precond <q> <q> & )
    )
(Sp robot*propose*push-to-door*check-precond*next-to
    (Goal <g> ^problem-space <p> ^state <s>
   ^op-set <opss> - ^applied <q>)
    (op-set <opss> ^operator <q>)
    (Problem-Space <p> ^name robot-domain)
    (Operator <q> ^name push-to-door ^box <box> )
    ;;(state <s>  )
    (State <s> ^robot <robot> ^next-to <nt>)
    (next-to <nt> ^obj1 <robot> ^obj2 <box>)
    -->
    (Goal <g> ^next-to-precond <q> <q> & )
    )

;; ========= push-to-loc precond tests
(Sp robot*propose*push-to-loc*check-precond*in-same-room
    (Goal <g> ^problem-space <p> ^state <s>
   ^op-set <opss> - ^applied <q>)
    (op-set <opss> ^operator <q>)
    (Problem-Space <p> ^name robot-domain)
    (Operator <q> ^name push-to-loc ^box <box> ^loc <loc>)
    ;;(state <s>  )
    (State <s> ^robot <robot> ^loc-in-room <lir> ^in-room <inr1> { <> <inr1> <inr2> } )
    (loc-in-room <lir> ^loc <loc> ^room <some-room>)
    (in-room <inr1> ^obj <robot> ^room <some-room>)
    (in-room <inr2> ^obj <box> ^room <some-room>)
    -->
    (Goal <g> ^in-same-room-precond <q> <q> & )
    )
(Sp robot*propose*push-to-loc*check-precond*next-to
    (Goal <g> ^problem-space <p> ^state <s>
   ^op-set <opss> - ^applied <q>)
    (op-set <opss> ^operator <q>)
    (Problem-Space <p> ^name robot-domain)
    (Operator <q> ^name push-to-loc ^box <box>)
    ;;(state <s>  )
    (State <s> ^robot <robot> ^next-to <nt>)
    (next-to <nt> ^obj1 <robot> ^obj2 <box>)
    -->
    (Goal <g> ^next-to-precond <q> <q> & )
    )
;; ========= go-through-door precond tests
(Sp robot*propose*go-through-door*check-precond*in-same-connective-room
    (Goal <g> ^problem-space <p> ^state <s>
   ^op-set <opss> - ^applied <q>)
    (op-set <opss> ^operator <q>)
    (Problem-Space <p> ^name robot-domain)
    (Operator <q> ^name go-through-door ^door <door> ^into-room <room>)
    ;;(state <s>  )
    (State <s> ^robot <robot> ^connects <conn> ^in-room <inr>)
    (connects <conn> ^door <door> ^room <room> { <> <room> <conn-room> } )
    (in-room <inr> ^obj <robot> ^room <conn-room>)
    -->
    (Goal <g> ^in-same-conn-room-precond <q> <q> & )
    )
(Sp robot*propose*go-through-door*check-precond*next-to
    (Goal <g> ^problem-space <p> ^state <s>
   ^op-set <opss> - ^applied <q>)
    (op-set <opss> ^operator <q>)
    (Problem-Space <p> ^name robot-domain)
    (Operator <q> ^name go-through-door ^door <door> )
    ;;(state <s>  )
    (State <s> ^robot <robot> ^next-to <nt>)
    (next-to <nt> ^obj1 <robot> ^obj2 <door>)
    -->
    (Goal <g> ^next-to-precond <q> <q> & )
    )
```

```
(Sp robot*propose*go-through-door*check-precond*door-open
    (Goal <g> ^problem-space <p> ^state <s>
   ^op-set <opss> - ^applied <q>)
    (op-set <opss> ^operator <q>)
    (Problem-Space <p> ^name robot-domain)
    (Operator <q> ^name go-through-door ^door <door> )
    ;;(state <s>  )
    (State <s> ^door-status <ds>)
    (door-status <ds> ^door <door> ^status open)
    -->
    (Goal <g> ^door-open-precond <q> <q> & )
    )

;; ========= push-through-door precond tests
(Sp robot*propose*push-through-door*check-precond*in-same-connective-room
    (Goal <g> ^problem-space <p> ^state <s>
   ^op-set <opss> - ^applied <q>)
    (op-set <opss> ^operator <q>)
    (Problem-Space <p> ^name robot-domain)
    (Operator <q> ^name push-through-door ^box <box> ^door <door> ^into-room <room>)
    ;;(state <s>  )
    (State <s> ^robot <robot> ^connects <conn> ^in-room <inr> { <> <inr> <inr2> } )
    (connects <conn> ^door <door> ^room <room> { <> <room> <conn-room> } )
    (in-room <inr> ^obj <robot> ^room <conn-room>)
    (in-room <inr2> ^obj <box> ^room <conn-room>)
    -->
    (Goal <g> ^in-same-conn-room-precond <q> <q> & )
    )
(Sp robot*propose*push-through-door*check-precond*next-to-robot
    (Goal <g> ^problem-space <p> ^state <s>
   ^op-set <opss> - ^applied <q>)
    (op-set <opss> ^operator <q>)
    (Problem-Space <p> ^name robot-domain)
    (Operator <q> ^name push-through-door ^box <box>)
    ;;(state <s>  )
    (State <s> ^robot <robot> ^next-to <nt>)
    (next-to <nt> ^obj1 <robot> ^obj2 <box>)
    -->
    (Goal <g> ^next-to-robot-precond <q> <q> & )
    )
(Sp robot*propose*push-through-door*check-precond*next-to-door
    (Goal <g> ^problem-space <p> ^state <s>
   ^op-set <opss> - ^applied <q>)
    (op-set <opss> ^operator <q>)
    (Problem-Space <p> ^name robot-domain)
    (Operator <q> ^name push-through-door ^box <box> ^door <door>)
    ;;(state <s>  )
    (State <s> ;;^robot <robot>
   ^next-to <nt>)
    (next-to <nt> ^obj1 <box> ^obj2 <door>)
    -->
    (Goal <g> ^next-to-door-precond <q> <q> & )
    )
(Sp robot*propose*push-through-door*check-precond*door-open
    (Goal <g> ^problem-space <p> ^state <s>
   ^op-set <opss> - ^applied <q>)
    (op-set <opss> ^operator <q>)
    (Problem-Space <p> ^name robot-domain)
    (Operator <q> ^name push-through-door ^door <door> )
    (State <s> ^door-status <ds>)
    (door-status <ds> ^door <door> ^status open)
    -->
    (Goal <g> ^door-open-precond <q> <q> & )
    )
;;========= open-door precond tests
(Sp robot*propose*open-door*check-precond*in-same-room-door
    (Goal <g> ^problem-space <p> ^state <s>
   ^op-set <opss> - ^applied <q>)
    (op-set <opss> ^operator <q>)
    (Problem-Space <p> ^name robot-domain)
```

```
    (Operator <q> ^name open-door ^door <door>)
    ;;(state <s>  )
    (State <s> ^robot <robot> ^in-room <inr1> ^connects <conn>)
    (in-room <inr1> ^obj <robot> ^room <room>)
    (connects <conn> ^door <door> ^room <room>)
    -->
    (Goal <g> ^in-same-room-door-precond <q> <q> & )
    )

(Sp robot*propose*open-door*check-precond*next-to
    (Goal <g> ^problem-space <p> ^state <s>
   ^op-set <opss> - ^applied <q>)
    (op-set <opss> ^operator <q>)
    (Problem-Space <p> ^name robot-domain)
    (Operator <q> ^name open-door ^door <door>)
    ;;(state <s>  )
    (State <s> ^robot <robot> ^next-to <nt>)
    (next-to <nt> ^obj1 <robot> ^obj2 <door>)
    -->
    (Goal <g> ^next-to-precond <q> <q> & )
    )

(Sp robot*propose*open-door*check-precond*door-closed
    (Goal <g> ^problem-space <p> ^state <s>
   ^op-set <opss> - ^applied <q>)
    (op-set <opss> ^operator <q>)
    (Problem-Space <p> ^name robot-domain)
    (Operator <q> ^name open-door ^door <door>)
    ;;(state <s>  )
    (State <s> ^door-status <ds>)
    (door-status <ds> ^door <door> ^status closed)
    -->
    (Goal <g> ^door-closed-precond <q> <q> & )
    )

;; ========= close-door precond tests
(Sp robot*propose*close-door*check-precond*in-same-room-door
    (Goal <g> ^problem-space <p> ^state <s>
  ^op-set <opss> - ^applied <q>)
    (op-set <opss> ^operator <q>)
    (Problem-Space <p> ^name robot-domain)
    (Operator <q> ^name close-door ^door <door>)
    ;;(state <s>  )
    (State <s> ^robot <robot> ^in-room <inr> ^connects <conn>)
    (connects <conn> ^door <door> ^room <room>)
    (in-room <inr> ^obj <robot> ^room <room>)
    -->
    (Goal <g> ^in-same-room-door-precond <q> <q> & )
    )

(Sp robot*propose*close-door*check-precond*next-to
    (Goal <g> ^problem-space <p> ^state <s>
   ^op-set <opss> - ^applied <q>)
    (op-set <opss> ^operator <q>)
    (Problem-Space <p> ^name robot-domain)
    (Operator <q> ^name close-door ^door <door>)
    ;;(state <s>  )
    (State <s> ^robot <robot> ^next-to <nt>)
    (next-to <nt> ^obj1 <robot> ^obj2 <door>)
    -->
    (Goal <g> ^next-to-precond <q> <q> & )
    )

(Sp robot*propose*close-door*check-precond*door-open
    (Goal <g> ^problem-space <p> ^state <s>
   ^op-set <opss> - ^applied <q>)
    (op-set <opss> ^operator <q>)
    (Problem-Space <p> ^name robot-domain)
    (Operator <q> ^name close-door ^door <door>)
    ;;(state <s>  )
    (State <s> ^door-status <ds>)
    (door-status <ds> ^door <door> ^status open)
    -->
    (Goal <g> ^door-open-precond <q> <q> & )
```

```
    )

;;Generation of Task Subgoals for Operator Preconditions:
;;When an operator is proposed but can not apply, an operator subgoal is
;;generated.  Below are listed the rules which create task subgoals
;;(representational, not architectural) for each precondition of the
;;subgoaled-upon operator.  Once the task subgoals are posted, MEA rules
;;will fire, to propose operators whose effects achieve the subgoals.
;;Other rules also test whether or not such a task subgoal is met,
;;and delete it if that is the case, or re-generate it if it becomes unachieved
;;during the course of further problem solving.
;;; ==========================================
(Sp robot*opsub*create-desired-goal-conjunct*goto-box
     (Goal <g> ^problem-space <p> ^name operator-subgoal ^object <g2>)
     (Goal <g2> ^operator <q> ^state <s>)
     (Operator <q> ^name goto-box ^box <box>)
     (State <s> ^robot <robot>)
     -->
     (Goal <g> ^desired <des>)
     (desired <des> ^goal-conjuncts <gc> ^for-op <q>
      ^goal-literals <box>)
     (goal-conjuncts <gc> ^in-same-room <ins>)
     (in-same-room <ins> ^box <box> ^robot <robot> ^args two)
     )
(Sp robot*opsub*create-desired-goal-conjunct*goto-door
     (Goal <g> ^problem-space <p> ^name operator-subgoal ^object <g2>)
     (Goal <g2> ^operator <q> ^state <s>)
     (Operator <q> ^name goto-door ^door <door>)
     (State <s> ^robot <robot>)
     -->
     (Goal <g> ^desired <des>)
     (desired <des> ^goal-conjuncts <gc> ^for-op <q>
      ^goal-literals <door>)
     (goal-conjuncts <gc> ^in-same-room-door <ins>)
     (in-same-room-door <ins> ^door <door> ^robot <robot> ^args two)
     )
(Sp robot*opsub*create-desired-goal-conjunct*goto-loc
     (Goal <g> ^problem-space <p> ^name operator-subgoal ^object <g2>)
     (Goal <g2> ^operator <q> ^state <s>)
     (Operator <q> ^name goto-loc ^loc <loc>)
     (State <s> ^robot <robot>)
     -->
     (Goal <g> ^desired <des>)
     (desired <des> ^goal-conjuncts <gc> ^for-op <q>
      ^goal-literals <loc>)
     (goal-conjuncts <gc> ^in-same-room <ins>)
     (in-same-room <ins> ^loc <loc> ^robot <robot> ^args two)
     )
(Sp robot*opsub*create-desired-goal-conjunct*push-box
     (Goal <g> ^problem-space <p> ^name operator-subgoal ^object <g2>)
     (Goal <g2> ^operator <q> ^state <s>)
     (Operator <q> ^name push-box ^move-box <box-x> ^to-box <box-y>)
     (State <s> ^robot <robot>)
     -->
     (Goal <g> ^desired <des>)
     (desired <des> ^goal-conjuncts <gc> ^for-op <q>
      ^goal-literals <box-x> + &, <box-y> + &)
     (goal-conjuncts <gc> ^in-same-room <ins>)
     (in-same-room <ins>  ^robot <robot> ^move-box <box-x>
   ^to-box <box-y> ^next-goal <nt> ^args three)
     (next-to <nt> ^robot <robot> ^box <box-x> ^args two)
     )
(Sp robot*opsub*create-desired-goal-conjunct*push-to-door
    (Goal <g> ^problem-space <p> ^name operator-subgoal ^object <g2>)
    (Goal <g2> ^operator <q> ^state <s>)
    (Operator <q> ^name push-to-door ^box <box> ^door <door>)
    (State <s> ^robot <robot>)
    -->
    (Goal <g> ^desired <des>)
    (desired <des> ^goal-conjuncts <gc> ^for-op <q>
     ^goal-literals <box> + &, <door> + &)
```

```
    (goal-conjuncts <gc> ^in-same-room-door <ins>)
    (in-same-room-door <ins>  ^robot <robot> ^box <box> ^door <door>
       ^next-goal <nt> ^args three)
    (next-to <nt> ^robot <robot> ^box <box> ^args two)
    )
(Sp robot*opsub*create-desired-goal-conjunct*push-to-loc
     (Goal <g> ^problem-space <p> ^name operator-subgoal ^object <g2>)
     (Goal <g2> ^operator <q> ^state <s>)
     (Operator <q> ^name push-to-loc ^box <box> ^loc <loc>)
     (State <s> ^robot <robot>)
     -->
     (Goal <g> ^desired <des>)
     (desired <des> ^goal-conjuncts <gc> ^for-op <q>
      ^goal-literals <box> + &, <loc> + &)
     (goal-conjuncts <gc> ^in-same-room <ins>)
     (in-same-room <ins>  ^robot <robot> ^box <box> ^loc <loc>
    ^next-goal <nt> ^args three)
     (next-to <nt> ^robot <robot> ^box <box> ^args two)
     )
(Sp robot*opsub*create-desired-goal-conjunct*go-through-door
     (Goal <g> ^problem-space <p> ^name operator-subgoal ^object <g2>)
     (Goal <g2> ^operator <q> ^state <s>)
     (Operator <q> ^name go-through-door ^door <door>  ^into-room <room>)
     (State <s> ^robot <robot>)
     -->
     (Goal <g> ^desired <des>)
     (desired <des> ^goal-conjuncts <gc> ^for-op <q>
      ^goal-literals <door> + &, <room> + &)
     (goal-conjuncts <gc> ^in-same-conn-room <ins>)
     (in-same-conn-room <ins>  ^robot <robot> ^door <door> ^into-room <room>
    ^next-goal <nt> ^args three)
     (next-to <nt> ^robot <robot> ^door <door> ^next-goal <do> ^args two)
     (door-open <do> ^door <door> ^args one)
     )
(Sp robot*opsub*create-desired-goal-conjunct*push-through-door
     (Goal <g> ^problem-space <p> ^name operator-subgoal ^object <g2>)
     (Goal <g2> ^operator <q> ^state <s>)
     (Operator <q> ^name push-through-door ^box <box> ^door <door>  ^into-room <room>)
     (State <s> ^robot <robot>)
     -->
     (Goal <g> ^desired <des>)
     (desired <des> ^goal-conjuncts <gc> ^for-op <q>
      ^goal-literals <box> + &, <door> + &, <room> + &)
     (goal-conjuncts <gc> ^in-same-conn-room <ins>)
     (in-same-conn-room <ins>  ^robot <robot> ^box <box> ^door <door>
^into-room <room> ^next-goal <nt> ^args four)
     (next-to <nt> ^box <box> ^door <door> ^next-goal <nt2> ^args two)
     (next-to <nt2> ^robot <robot> ^box <box> ^next-goal <do> ^args two)
     (door-open <do> ^door <door> ^args one)
     )
(Sp robot*opsub*create-desired-goal-conjunct*open-door
     (Goal <g> ^problem-space <p> ^name operator-subgoal ^object <g2>)
     (Goal <g2> ^operator <q> ^state <s>)
     (Operator <q> ^name open-door ^door <door>)
     (State <s> ^robot <robot>)
     -->
     (Goal <g> ^desired <des>)
     (desired <des> ^goal-conjuncts <gc> ^for-op <q>
      ^goal-literals <door>)
     (goal-conjuncts <gc> ^in-same-room-door <insrd> )
     (in-same-room-door <insrd> ^robot <robot> ^door <door> ^args two
^next-goal <nt>)
     (next-to <nt>  ^robot <robot> ^door <door> ^next-goal <dc> ^args two)
     (door-closed <dc> ^door <door> ^args one)
     )
(Sp robot*opsub*create-desired-goal-conjunct*close-door
     (Goal <g> ^problem-space <p> ^name operator-subgoal ^object <g2>)
     (Goal <g2> ^operator <q> ^state <s>)
     (Operator <q> ^name close-door ^door <door>)
     (State <s> ^robot <robot>)
     -->
```

```
     (Goal <g> ^desired <des>)
     (desired <des> ^goal-conjuncts <gc> ^for-op <q>
      ^goal-literals <door>)
     (goal-conjuncts <gc> ^in-same-room-door <insrd> )
     (in-same-room-door <insrd> ^robot <robot> ^door <door> ^args two
^next-goal <nt>)
     (next-to <nt>  ^robot <robot> ^door <door> ^next-goal <do> ^args two)
     (door-open <do> ^door <door> ^args one)
     )
```

D.3 2-goal-conjunct tasks

The following pieces of Soar code describe the initial and goal states for the 2-goal-conjunct Robot Domain tasks used in Chapter 6. For each task, the listing first describes the relevant initial state augmentations for the task, and then the task goal.

```
;; Task 1
    ;; initial state augmentations
    (next-to <nt1> ^obj1 <box-b> ^obj2 <Room2Room4>)
    (door-status <ds1> ^door <Room4Room3> ^status open)
    (door-status <ds2> ^door <Room7Room4> ^status open)
    (door-status <ds3> ^door <Room2Room4> ^status closed)
    (door-status <ds4> ^door <Room5Room4> ^status closed)
    (door-status <ds5> ^door <Room5Room7> ^status open)
    (door-status <ds6> ^door <Room2Room1> ^status open)
    (door-status <ds7> ^door <Room5Room2> ^status closed)
    (door-status <ds8> ^door <Room6Room5> ^status open)
    (in-room <inr1> ^obj <box-e> ^room <Room6>)
    (in-room <inr2> ^obj <box-d> ^room <Room3>)
    (in-room <inr3> ^obj <box-c> ^room <Room1>)
    (in-room <inr4> ^obj <box-b> ^room <Room4>)
    (in-room <inr5> ^obj <box-a> ^room <Room5>)
    (in-room <inr6> ^obj <robot> ^room <Room7>)
    ;; goal statement
    (goal <g> ^desired-state <d>)
    (desired <d> ^goal-conjuncts <gc> ^better lower
          )
    (goal-conjuncts <gc> ^next-to <gnt> ^in-same-room <gnt2>
    )
    (next-to <gnt> ^box <box-b> <box-b> &, <box-e> <box-e> &, ^args two
     ^goal-name <gname1>)
    (in-same-room <gnt2> ^box <box-b> + &, <box-e> + &, ^args two ^goal-name <gname2>)
;; Task 2
    ;; initial state augmentations
    (next-to <nt1> ^obj1 <box-b> ^obj2 <Room5Room4>)
    (next-to <nt2> ^obj1 <box-d> ^obj2 <box-a>)
    (next-to <nt3> ^obj1 <box-a> ^obj2 <box-d>)
    (door-status <ds1> ^door <Room4Room3> ^status closed)
    (door-status <ds2> ^door <Room7Room4> ^status closed)
    (door-status <ds3> ^door <Room2Room4> ^status closed)
    (door-status <ds4> ^door <Room5Room4> ^status open)
    (door-status <ds5> ^door <Room5Room7> ^status open)
    (door-status <ds6> ^door <Room2Room1> ^status open)
    (door-status <ds7> ^door <Room5Room2> ^status closed)
    (door-status <ds8> ^door <Room6Room5> ^status open)
    (in-room <inr1> ^obj <box-e> ^room <Room7>)
    (in-room <inr2> ^obj <box-d> ^room <Room6>)
    (in-room <inr3> ^obj <box-c> ^room <Room1>)
    (in-room <inr4> ^obj <box-b> ^room <Room5>)
```

```
    (in-room <inr5> ^obj <box-a> ^room <Room6>)
    (in-room <inr6> ^obj <robot> ^room <Room5>)
    ;; goal statement
    (goal <g> ^desired-state <d>)
    (desired <d> ^goal-conjuncts <gc> ^better lower
     )
    (goal-conjuncts <gc> ^next-to <gnt> ^in-room <gnt2>
         )
    (next-to <gnt> ^box <box-b> ^door <Room4Room3> ^args two ^goal-name <gname1>)
    (in-room <gnt2> ^box <box-a> ^room <Room1> ^args two ^goal-name <gname2>)
;;  Task 3
    ;; initial state augmentations
    (next-to <nt1> ^obj1 <box-e> ^obj2 <Room7Room4>)
    (door-status <ds1> ^door <Room4Room3> ^status open)
    (door-status <ds2> ^door <Room7Room4> ^status open)
    (door-status <ds3> ^door <Room2Room4> ^status open)
    (door-status <ds4> ^door <Room5Room4> ^status closed)
    (door-status <ds5> ^door <Room5Room7> ^status open)
    (door-status <ds6> ^door <Room2Room1> ^status closed)
    (door-status <ds7> ^door <Room5Room2> ^status closed)
    (door-status <ds8> ^door <Room6Room5> ^status closed)
    (in-room <inr1> ^obj <box-e> ^room <Room4>)
    (in-room <inr2> ^obj <box-d> ^room <Room2>)
    (in-room <inr3> ^obj <box-c> ^room <Room1>)
    (in-room <inr4> ^obj <box-b> ^room <Room3>)
    (in-room <inr5> ^obj <box-a> ^room <Room7>)
    (in-room <inr6> ^obj <robot> ^room <Room7>)
    ;; goal statement
    (goal <g> ^desired-state <d>)
    (desired <d> ^goal-conjuncts <gc> ^better lower
     )
    (goal-conjuncts <gc> ^in-room <gnt> ^door-closed <gnt2>
    )
    (in-room <gnt> ^box <box-e> ^room <Room3> ^args two ^goal-name <gname1>)
    (door-closed <gnt2> ^door <Room2Room4> ^args one ^goal-name <gname2>)

;; Task 4
    ;;;; initial state augmentations
    (next-to <nt1> ^obj1 <box-e> ^obj2 <box-d>)
    (next-to <nt2> ^obj1 <box-d> ^obj2 <box-e>)
    (next-to <nt3> ^obj1 <box-c> ^obj2 <box-b>)
    (next-to <nt4> ^obj1 <box-b> ^obj2 <box-c>)
    (door-status <ds1> ^door <Room4Room3> ^status open)
    (door-status <ds2> ^door <Room7Room4> ^status open)
    (door-status <ds3> ^door <Room2Room4> ^status closed)
    (door-status <ds4> ^door <Room5Room4> ^status open)
    (door-status <ds5> ^door <Room5Room7> ^status closed)
    (door-status <ds6> ^door <Room2Room1> ^status closed)
    (door-status <ds7> ^door <Room5Room2> ^status open)
    (door-status <ds8> ^door <Room6Room5> ^status open)
    (in-room <inr1> ^obj <box-e> ^room <Room7>)
    (in-room <inr2> ^obj <box-d> ^room <Room7>)
    (in-room <inr3> ^obj <box-c> ^room <Room5>)
    (in-room <inr4> ^obj <box-b> ^room <Room5>)
    (in-room <inr5> ^obj <box-a> ^room <Room6>)
    (in-room <inr6> ^obj <robot> ^room <Room5>)
    ;; goal statement
    (goal <g> ^desired-state <d>)
    (desired <d> ^goal-conjuncts <gc> ^better lower
     )
    (goal-conjuncts <gc> ^in-room <gnt> ^door-open <gnt2>
    )
    (in-room <gnt> ^box <box-b> ^room <Room7> ^args two ^goal-name <gname1>)
    (door-open <gnt2> ^door <Room5Room7> ^args one ^goal-name <gname2>)

;; Task 5
    ;; initial state augmentations
    (next-to <nt1> ^obj1 <box-e> ^obj2 <box-a>)
    (next-to <nt2> ^obj1 <box-a> ^obj2 <box-e>)
```

```
    (next-to <nt3> ^obj1 <box-e> ^obj2 <Room4Room3>)
    (next-to <nt4> ^obj1 <box-a> ^obj2 <Room4Room3>)
    (next-to <nt5> ^obj1 <box-b> ^obj2 <Room5Room2>)
    (next-to <nt6> ^obj1 <box-d> ^obj2 <Room6Room5>)
    (door-status <ds1> ^door <Room4Room3> ^status open)
    (door-status <ds2> ^door <Room7Room4> ^status closed)
    (door-status <ds3> ^door <Room2Room4> ^status open)
    (door-status <ds4> ^door <Room5Room4> ^status open)
    (door-status <ds5> ^door <Room5Room7> ^status open)
    (door-status <ds6> ^door <Room2Room1> ^status open)
    (door-status <ds7> ^door <Room5Room2> ^status closed)
    (door-status <ds8> ^door <Room6Room5> ^status closed)
    (in-room <inr1> ^obj <box-e> ^room <Room4>)
    (in-room <inr2> ^obj <box-d> ^room <Room6>)
    (in-room <inr3> ^obj <box-c> ^room <Room2>)
    (in-room <inr4> ^obj <box-b> ^room <Room5>)
    (in-room <inr5> ^obj <box-a> ^room <Room4>)
    (in-room <inr6> ^obj <robot> ^room <Room4>)
    ;; goal statement
    (goal <g> ^desired-state <d>)
    (desired <d> ^goal-conjuncts <gc> ^better lower
     )
    (goal-conjuncts <gc> ^next-to <gnt> ^door-closed <gnt2>
    )
    (next-to <gnt> ^box <box-c> + &,  <box-d> + &, ^args two ^goal-name <gname1>)
    (door-closed <gnt2> ^door <Room5Room7> ^args one ^goal-name <gname2>)

;; Task 6
    ;; initial state augmentations
    (next-to <nt1> ^obj1 <box-d> ^obj2 <box-a>)
    (next-to <nt2> ^obj1 <box-a> ^obj2 <box-d>)
    (next-to <nt3> ^obj1 <box-d> ^obj2 <Room5Room4>)
    (next-to <nt4> ^obj1 <box-a> ^obj2 <Room4Room3>)
    (door-status <ds1> ^door <Room4Room3> ^status open)
    (door-status <ds2> ^door <Room7Room4> ^status closed)
    (door-status <ds3> ^door <Room2Room4> ^status closed)
    (door-status <ds4> ^door <Room5Room4> ^status open)
    (door-status <ds5> ^door <Room5Room7> ^status closed)
    (door-status <ds6> ^door <Room2Room1> ^status closed)
    (door-status <ds7> ^door <Room5Room2> ^status closed)
    (door-status <ds8> ^door <Room6Room5> ^status open)
    (in-room <inr1> ^obj <box-e> ^room <Room2>)
    (in-room <inr2> ^obj <box-d> ^room <Room4>)
    (in-room <inr3> ^obj <box-c> ^room <Room7>)
    (in-room <inr4> ^obj <box-b> ^room <Room6>)
    (in-room <inr5> ^obj <box-a> ^room <Room4>)
    (in-room <inr6> ^obj <robot> ^room <Room5>)
    ;; goal statement
    (goal <g> ^desired-state <d>)
    (desired <d> ^goal-conjuncts <gc> ^better lower
     )
    (goal-conjuncts <gc> ^next-to <gnt> ^door-open <gnt2>
    )
    (next-to <gnt> ^box <box-e> ^door <Room5Room2> ^args two ^goal-name <gname1>)
    (door-open <gnt2> ^door <Room5Room7> ^args one ^goal-name <gname2>)

;;; Task 7
    ;; initial state augmentations
    (next-to <nt1> ^obj1 <box-c> ^obj2 <box-a>)
    (next-to <nt2> ^obj1 <box-a> ^obj2 <box-c>)
    (next-to <nt3> ^obj1 <box-d> ^obj2 <Room2Room4>)
    (door-status <ds1> ^door <Room4Room3> ^status open)
    (door-status <ds2> ^door <Room7Room4> ^status closed)
    (door-status <ds3> ^door <Room2Room4> ^status closed)
    (door-status <ds4> ^door <Room5Room4> ^status closed)
    (door-status <ds5> ^door <Room5Room7> ^status closed)
    (door-status <ds6> ^door <Room2Room1> ^status open)
    (door-status <ds7> ^door <Room5Room2> ^status open)
    (door-status <ds8> ^door <Room6Room5> ^status closed)
    (in-room <inr1> ^obj <box-e> ^room <Room3>)
```

```
    (in-room <inr2> ^obj <box-d> ^room <Room4>)
    (in-room <inr3> ^obj <box-c> ^room <Room5>)
    (in-room <inr4> ^obj <box-b> ^room <Room1>)
    (in-room <inr5> ^obj <box-a> ^room <Room5>)
    (in-room <inr6> ^obj <robot> ^room <Room6>)
    ;; goal statement
    (goal <g> ^desired-state <d>)
    (desired <d> ^goal-conjuncts <gc> ^better lower
     )
    (goal-conjuncts <gc> ^next-to <gnt> ^door-closed <gnt2>
    )
    (next-to <gnt> ^box <box-a> ^door <Room5Room2> ^args two
     ^goal-name <gname1>)
    (door-closed <gnt2> ^door <Room5Room2> ^args one
 ^goal-name <gname2>)

;;; Task 8
    ;; initial state augmentations
    (next-to <nt1> ^obj1 <box-b> ^obj2 <Room5Room2>)
    (door-status <ds1> ^door <Room4Room3> ^status open)
    (door-status <ds2> ^door <Room7Room4> ^status open)
    (door-status <ds3> ^door <Room2Room4> ^status closed)
    (door-status <ds4> ^door <Room5Room4> ^status closed)
    (door-status <ds5> ^door <Room5Room7> ^status open)
    (door-status <ds6> ^door <Room2Room1> ^status closed)
    (door-status <ds7> ^door <Room5Room2> ^status open)
    (door-status <ds8> ^door <Room6Room5> ^status open)
    (in-room <inr1> ^obj <box-e> ^room <Room5>)
    (in-room <inr2> ^obj <box-d> ^room <Room5>)
    (in-room <inr3> ^obj <box-c> ^room <Room4>)
    (in-room <inr4> ^obj <box-b> ^room <Room2>)
    (in-room <inr5> ^obj <box-a> ^room <Room4>)
    (in-room <inr6> ^obj <robot> ^room <Room7>)
    ;; goal statement
    (goal <g> ^desired-state <d>)
    (desired <d> ^goal-conjuncts <gc> ^better lower
     )
    (goal-conjuncts <gc> ^in-room <gnt> ^door-open <gnt2>
    )
    (in-room <gnt> ^robot <robot> ^room <Room2> ^args two
     ^goal-name <gname1>)
    (door-open <gnt2> ^door <Room2Room1> ^args one ^goal-name <gname2>)

;;; Task 9
    ;; initial state augmentations
    (next-to <nt1> ^obj1 <box-b> ^obj2 <box-d>)
    (next-to <nt2> ^obj1 <box-d> ^obj2 <box-b>)
    (next-to <nt3> ^obj1 <box-b> ^obj2 <Room5Room4>)
    (door-status <ds1> ^door <Room4Room3> ^status open)
    (door-status <ds2> ^door <Room7Room4> ^status closed)
    (door-status <ds3> ^door <Room2Room4> ^status closed)
    (door-status <ds4> ^door <Room5Room4> ^status open)
    (door-status <ds5> ^door <Room5Room7> ^status open)
    (door-status <ds6> ^door <Room2Room1> ^status closed)
    (door-status <ds7> ^door <Room5Room2> ^status closed)
    (door-status <ds8> ^door <Room6Room5> ^status closed)
    (in-room <inr1> ^obj <box-e> ^room <Room6>)
    (in-room <inr2> ^obj <box-d> ^room <Room5>)
    (in-room <inr3> ^obj <box-c> ^room <Room6>)
    (in-room <inr4> ^obj <box-b> ^room <Room5>)
    (in-room <inr5> ^obj <box-a> ^room <Room3>)
    (in-room <inr6> ^obj <robot> ^room <Room6>)
    ;; goal statement
    (goal <g> ^desired-state <d>)
    (desired <d> ^goal-conjuncts <gc> ^better lower
     )
    (goal-conjuncts <gc> ^in-room <gnt> ^next-to <gnt2>
    )
    (in-room <gnt> ^box <box-c> ^room <Room3> ^args two ^goal-name <gname1>)
    (next-to <gnt2> ^box  <box-a> ^door <Room5Room2> ^args two
```

```
     ^goal-name <gname2>)

;; Task 10
    ;; initial state augmentations
    (next-to <nt1> ^obj1 <box-d> ^obj2 <Room2Room4>)
    (next-to <nt2> ^obj1 <box-b> ^obj2 <Room5Room7>)
    (next-to <nt3> ^obj1 <box-c> ^obj2 <Room6Room5>)
    (door-status <ds1> ^door <Room4Room3> ^status closed)
    (door-status <ds2> ^door <Room7Room4> ^status open)
    (door-status <ds3> ^door <Room2Room4> ^status open)
    (door-status <ds4> ^door <Room5Room4> ^status closed)
    (door-status <ds5> ^door <Room5Room7> ^status closed)
    (door-status <ds6> ^door <Room2Room1> ^status open)
    (door-status <ds7> ^door <Room5Room2> ^status closed)
    (door-status <ds8> ^door <Room6Room5> ^status open)
    (in-room <inr1> ^obj <box-e> ^room <Room3>)
    (in-room <inr2> ^obj <box-d> ^room <Room2>)
    (in-room <inr3> ^obj <box-c> ^room <Room5>)
    (in-room <inr4> ^obj <box-b> ^room <Room5>)
    (in-room <inr5> ^obj <box-a> ^room <Room2>)
    (in-room <inr6> ^obj <robot> ^room <Room4>)
    ;; goal statement
    (goal <g> ^desired-state <d>)
    (desired <d> ^goal-conjuncts <gc> ^better lower
     )
    (goal-conjuncts <gc> ^in-room <gnt> ^next-to <gnt2>
    )
    (in-room <gnt> ^box <box-d> ^room <Room6> ^args two ^goal-name <gname1>)
    (next-to <gnt2> ^box  <box-c> ^door <Room5Room2> ^args two ^goal-name <gname2>)

;; Task 11
    ;; initial state augmentations
    (next-to <nt1> ^obj1 <box-a> ^obj2 <Room2Room1>)
    (door-status <ds1> ^door <Room4Room3> ^status closed)
    (door-status <ds2> ^door <Room7Room4> ^status open)
    (door-status <ds3> ^door <Room2Room4> ^status closed)
    (door-status <ds4> ^door <Room5Room4> ^status closed)
    (door-status <ds5> ^door <Room5Room7> ^status closed)
    (door-status <ds6> ^door <Room2Room1> ^status closed)
    (door-status <ds7> ^door <Room5Room2> ^status open)
    (door-status <ds8> ^door <Room6Room5> ^status open)
    (in-room <inr1> ^obj <box-e> ^room <Room1>)
    (in-room <inr2> ^obj <box-d> ^room <Room2>)
    (in-room <inr3> ^obj <box-c> ^room <Room6>)
    (in-room <inr4> ^obj <box-b> ^room <Room3>)
    (in-room <inr5> ^obj <box-a> ^room <Room1>)
    (in-room <inr6> ^obj <robot> ^room <Room7>)
    ;; goal statement
    (goal <g> ^desired-state <d>)
    (desired <d> ^goal-conjuncts <gc> ^better lower
     )
    (goal-conjuncts <gc> ^in-room <gnt> ^next-to <gnt2>
    )
    (in-room <gnt> ^box <box-e> ^room <Room4> ^args two ^goal-name <gname1>)
    (next-to <gnt2> ^box  <box-b> + &, <box-d> + &, ^args two
     ^goal-name <gname2>)
;; Task 13
    ;; initial state augmentations
    (next-to <nt1> ^obj1 <box-c> ^obj2 <Room2Room4>)
    (next-to <nt2> ^obj1 <box-b> ^obj2 <box-e>)
    (next-to <nt3> ^obj1 <box-e> ^obj2 <box-b>)
    (door-status <ds1> ^door <Room4Room3> ^status closed)
    (door-status <ds2> ^door <Room7Room4> ^status open)
    (door-status <ds3> ^door <Room2Room4> ^status open)
    (door-status <ds4> ^door <Room5Room4> ^status open)
    (door-status <ds5> ^door <Room5Room7> ^status open)
    (door-status <ds6> ^door <Room2Room1> ^status closed)
    (door-status <ds7> ^door <Room5Room2> ^status open)
    (door-status <ds8> ^door <Room6Room5> ^status closed)
```

```
    (in-room <inr1> ^obj <box-e> ^room <Room1>)
    (in-room <inr2> ^obj <box-d> ^room <Room6>)
    (in-room <inr3> ^obj <box-c> ^room <Room4>)
    (in-room <inr4> ^obj <box-b> ^room <Room1>)
    (in-room <inr5> ^obj <box-a> ^room <Room3>)
    (in-room <inr6> ^obj <robot> ^room <Room3>)
    ;; goal statement
    (goal <g> ^desired-state <d>)
    (desired <d> ^goal-conjuncts <gc> ^better lower
     )
    (goal-conjuncts <gc> ^door-open <gnt> ^door-closed <gnt2>
    )
    (door-open <gnt> ^door <Room6Room5> ^args one ^goal-name <gname1>)
    (door-closed <gnt2> ^door <Room2Room4> ^args one ^goal-name <gname2>)
;; Task 14
    ;; initial state augmentations
    (next-to <nt1> ^obj1 <box-d> ^obj2 <Room4Room3>)
    (next-to <nt2> ^obj1 <box-e> ^obj2 <Room6Room5>)
    (door-status <ds1> ^door <Room4Room3> ^status closed)
    (door-status <ds2> ^door <Room7Room4> ^status closed)
    (door-status <ds3> ^door <Room2Room4> ^status open)
    (door-status <ds4> ^door <Room5Room4> ^status open)
    (door-status <ds5> ^door <Room5Room7> ^status closed)
    (door-status <ds6> ^door <Room2Room1> ^status open)
    (door-status <ds7> ^door <Room5Room2> ^status closed)
    (door-status <ds8> ^door <Room6Room5> ^status closed)
    (in-room <inr1> ^obj <box-e> ^room <Room5>)
    (in-room <inr2> ^obj <box-d> ^room <Room4>)
    (in-room <inr3> ^obj <box-c> ^room <Room5>)
    (in-room <inr4> ^obj <box-b> ^room <Room6>)
    (in-room <inr5> ^obj <box-a> ^room <Room3>)
    (in-room <inr6> ^obj <robot> ^room <Room1>)
    ;; goal statement
    (goal <g> ^desired-state <d>)
    (desired <d> ^goal-conjuncts <gc> ^better lower
     )
    (goal-conjuncts <gc> ^door-open <gnt> ^in-room <gnt2>
    )
    (door-open <gnt> ^door <Room7Room4> ^args one ^goal-name <gname1>)
    (in-room <gnt2> ^box <box-b> ^room <Room3> ^args two ^goal-name <gname2>)

;; Task 15
    ;; initial state augmentations
    (next-to <nt1> ^obj1 <box-d> ^obj2 <box-b>)
    (next-to <nt2> ^obj1 <box-b> ^obj2 <box-d>)
    (next-to <nt3> ^obj1 <box-c> ^obj2 <box-d>)
    (next-to <nt4> ^obj1 <box-d> ^obj2 <box-c>)
    (next-to <nt5> ^obj1 <box-c> ^obj2 <Room4Room3>)
    (door-status <ds1> ^door <Room4Room3> ^status open)
    (door-status <ds2> ^door <Room7Room4> ^status open)
    (door-status <ds3> ^door <Room2Room4> ^status open)
    (door-status <ds4> ^door <Room5Room4> ^status open)
    (door-status <ds5> ^door <Room5Room7> ^status open)
    (door-status <ds6> ^door <Room2Room1> ^status open)
    (door-status <ds7> ^door <Room5Room2> ^status open)
    (door-status <ds8> ^door <Room6Room5> ^status open)
    (in-room <inr1> ^obj <box-e> ^room <Room7>)
    (in-room <inr2> ^obj <box-d> ^room <Room3>)
    (in-room <inr3> ^obj <box-c> ^room <Room3>)
    (in-room <inr4> ^obj <box-b> ^room <Room3>)
    (in-room <inr5> ^obj <box-a> ^room <Room6>)
    (in-room <inr6> ^obj <robot> ^room <Room5>)
    ;; goal statement
    (goal <g> ^desired-state <d>)
    (desired <d> ^goal-conjuncts <gc> ^better lower
     )
    (goal-conjuncts <gc> ^door-closed <gnt> ^in-room <gnt2>
    )
    (door-closed <gnt> ^door <Room6Room5> ^args one ^goal-name <gname1>)
```

```
    (in-room <gnt2> ^box <box-c> ^room <Room7> ^args two ^goal-name <gname2>)

;; Task 16
    ;; initial state augmentations
    (next-to <nt1> ^obj1 <box-e> ^obj2 <Room5Room2>)
    (next-to <nt2> ^obj1 <box-a> ^obj2 <box-d>)
    (next-to <nt3> ^obj1 <box-d> ^obj2 <box-a>)

    (door-status <ds1> ^door <Room4Room3> ^status open)
    (door-status <ds2> ^door <Room7Room4> ^status closed)
    (door-status <ds3> ^door <Room2Room4> ^status closed)
    (door-status <ds4> ^door <Room5Room4> ^status open)
    (door-status <ds5> ^door <Room5Room7> ^status closed)
    (door-status <ds6> ^door <Room2Room1> ^status open)
    (door-status <ds7> ^door <Room5Room2> ^status closed)
    (door-status <ds8> ^door <Room6Room5> ^status open)

    (in-room <inr1> ^obj <box-e> ^room <Room5>)
    (in-room <inr2> ^obj <box-d> ^room <Room7>)
    (in-room <inr3> ^obj <box-c> ^room <Room5>)
    (in-room <inr4> ^obj <box-b> ^room <Room6>)
    (in-room <inr5> ^obj <box-a> ^room <Room7>)
    (in-room <inr6> ^obj <robot> ^room <Room3>)

    ;; goal statement
    (goal <g> ^desired-state <d>)
    (desired <d> ^goal-conjuncts <gc> ^better lower
     )
    (goal-conjuncts <gc> ^next-to <gnt> ^in-room <gnt2>
    )
    (next-to <gnt> ^box <box-c> ^door <Room5Room4> ^args two
     ^goal-name <gname1>)
    (in-room <gnt2> ^box <box-e> ^room <Room7> ^args two
     ^goal-name <gname2>)

;; Task 17
    ;; initial state augmentations
    (next-to <nt1> ^obj1 <box-b> ^obj2 <Room5psp>)
    (next-to <nt2> ^obj1 <box-a> ^obj2 <Room2Room1>)

    (door-status <ds1> ^door <Room4Room3> ^status closed)
    (door-status <ds2> ^door <Room7Room4> ^status closed)
    (door-status <ds3> ^door <Room2Room4> ^status open)
    (door-status <ds4> ^door <Room5Room4> ^status open)
    (door-status <ds5> ^door <Room5Room7> ^status open)
    (door-status <ds6> ^door <Room2Room1> ^status closed)
    (door-status <ds7> ^door <Room5Room2> ^status open)
    (door-status <ds8> ^door <Room6Room5> ^status open)

    (in-room <inr1> ^obj <box-e> ^room <Room6>)
    (in-room <inr2> ^obj <box-d> ^room <Room5>)
    (in-room <inr3> ^obj <box-c> ^room <Room4>)
    (in-room <inr4> ^obj <box-b> ^room <Room7>)
    (in-room <inr5> ^obj <box-a> ^room <Room2>)
    (in-room <inr6> ^obj <robot> ^room <Room2>)

    ;; goal statement
    (goal <g> ^desired-state <d>)
    (desired <d> ^goal-conjuncts <gc> ^better lower
     )
    (goal-conjuncts <gc> ^door-open <gnt> ^in-room <gnt2>
    )
    (door-open <gnt> ^door <Room4Room3> ^args one ^goal-name <gname1>)
    (in-room <gnt2> ^box <box-d> ^room <Room2> ^args two ^goal-name <gname2>)
;; Task 18
    ;; initial state augmentations
    (next-to <nt1> ^obj1 <box-e> ^obj2 <box-b>)
    (next-to <nt2> ^obj1 <box-b> ^obj2 <box-e>)
    (next-to <nt3> ^obj1 <box-d> ^obj2 <box-a>)
    (next-to <nt4> ^obj1 <box-a> ^obj2 <box-d>)
    (next-to <nt5> ^obj1 <box-d> ^obj2 <Room7Room4>)
    (next-to <nt6> ^obj1 <box-c> ^obj2 <Room6Room5>)

    (door-status <ds1> ^door <Room4Room3> ^status closed)
    (door-status <ds2> ^door <Room7Room4> ^status open)
    (door-status <ds3> ^door <Room2Room4> ^status closed)
```

```
    (door-status <ds4> ^door <Room5Room4> ^status closed)
    (door-status <ds5> ^door <Room5Room7> ^status open)
    (door-status <ds6> ^door <Room2Room1> ^status closed)
    (door-status <ds7> ^door <Room5Room2> ^status closed)
    (door-status <ds8> ^door <Room6Room5> ^status closed)
    (in-room <inr1> ^obj <box-e> ^room <Room1>)
    (in-room <inr2> ^obj <box-d> ^room <Room7>)
    (in-room <inr3> ^obj <box-c> ^room <Room5>)
    (in-room <inr4> ^obj <box-b> ^room <Room1>)
    (in-room <inr5> ^obj <box-a> ^room <Room7>)
    (in-room <inr6> ^obj <robot> ^room <Room1>)
    ;; goal statements
    (goal <g> ^desired-state <d>)
    (desired <d> ^goal-conjuncts <gc> ^better lower
     )
    (goal-conjuncts <gc> ^door-closed <gnt> ^door-open <gnt2>
    )
    (door-closed <gnt> ^door <Room5Room7> ^args one ^goal-name <gname1>)
    (door-open <gnt2> ^door <Room6Room5> ^args one ^goal-name <gname2>)

;; Task 19
    ;; initial state augmentations
    (next-to <nt1> ^obj1 <box-d> ^obj2 <box-c>)
    (next-to <nt2> ^obj1 <box-c> ^obj2 <box-d>)
    (next-to <nt3> ^obj1 <box-b> ^obj2 <Room4Room3>)
    (door-status <ds1> ^door <Room4Room3> ^status open)
    (door-status <ds2> ^door <Room7Room4> ^status closed)
    (door-status <ds3> ^door <Room2Room4> ^status closed)
    (door-status <ds4> ^door <Room5Room4> ^status open)
    (door-status <ds5> ^door <Room5Room7> ^status closed)
    (door-status <ds6> ^door <Room2Room1> ^status closed)
    (door-status <ds7> ^door <Room5Room2> ^status open)
    (door-status <ds8> ^door <Room6Room5> ^status open)
    (in-room <inr1> ^obj <box-e> ^room <Room7>)
    (in-room <inr2> ^obj <box-d> ^room <Room6>)
    (in-room <inr3> ^obj <box-c> ^room <Room6>)
    (in-room <inr4> ^obj <box-b> ^room <Room3>)
    (in-room <inr5> ^obj <box-a> ^room <Room5>)
    (in-room <inr6> ^obj <robot> ^room <Room4>)
    ;; goal statement
    (goal <g> ^desired-state <d>)
    (desired <d> ^goal-conjuncts <gc> ^better lower
     )
    (goal-conjuncts <gc> ^door-closed <gnt> ^in-room <gnt2>
    )
    (door-closed <gnt> ^door <Room5Room4> ^args one ^goal-name <gname1>)
    (in-room <gnt2> ^box <box-b> ^room <Room2> ^args two ^goal-name <gname2>)

;; Task 20
    ;; initial state augmentations
    (next-to <nt1> ^obj1 <box-e> ^obj2 <box-a>)
    (next-to <nt2> ^obj1 <box-a> ^obj2 <box-e>)
    (door-status <ds1> ^door <Room4Room3> ^status open)
    (door-status <ds2> ^door <Room7Room4> ^status closed)
    (door-status <ds3> ^door <Room2Room4> ^status open)
    (door-status <ds4> ^door <Room5Room4> ^status open)
    (door-status <ds5> ^door <Room5Room7> ^status open)
    (door-status <ds6> ^door <Room2Room1> ^status closed)
    (door-status <ds7> ^door <Room5Room2> ^status closed)
    (door-status <ds8> ^door <Room6Room5> ^status closed)
    (in-room <inr1> ^obj <box-e> ^room <Room6>)
    (in-room <inr2> ^obj <box-d> ^room <Room7>)
    (in-room <inr3> ^obj <box-c> ^room <Room5>)
    (in-room <inr4> ^obj <box-b> ^room <Room7>)
    (in-room <inr5> ^obj <box-a> ^room <Room6>)
    (in-room <inr6> ^obj <robot> ^room <Room3>)
    ;; goal statement
    (goal <g> ^desired-state <d>)
    (desired <d> ^goal-conjuncts <gc> ^better lower
     )
```

```
    (goal-conjuncts <gc> ^door-closed <gnt> ^next-to <gnt2>
    )
    (door-closed <gnt> ^door <Room5Room7> ^args one ^goal-name <gname1>)
    (next-to <gnt2> ^box <box-e> ^door <Room4Room3> ^args two ^goal-name <gname2>)
```

D.4 3-goal-conjunct tasks

The following pieces of Soar code describe the initial and goal states for the 3-goal-conjunct Robot Domain tasks used in Chapter 6. For each task, the listing first describes the relevant initial state augmentations for the task, and then the task goal.

```
;;; Task 1
    ;; initial state augmentations
    (door-status <ds1> ^door <Room4Room3> ^status open)
    (door-status <ds2> ^door <Room7Room4> ^status closed)
    (door-status <ds3> ^door <Room2Room4> ^status closed)
    (door-status <ds4> ^door <Room5Room4> ^status open)
    (door-status <ds5> ^door <Room5Room7> ^status open)
    (door-status <ds6> ^door <Room2Room1> ^status closed)
    (door-status <ds7> ^door <Room5Room2> ^status open)
    (door-status <ds8> ^door <Room6Room5> ^status open)
    (in-room <inr1> ^obj <box-e> ^room <Room7>)
    (in-room <inr2> ^obj <box-d> ^room <Room7>)
    (in-room <inr3> ^obj <box-c> ^room <Room6>)
    (in-room <inr4> ^obj <box-b> ^room <Room3>)
    (in-room <inr5> ^obj <box-a> ^room <Room5>)
    (in-room <inr6> ^obj <robot> ^room <Room6>)
    ;; goal statement
    (goal <g> ^desired-state <d>)
    (desired <d> ^goal-conjuncts <gc> ^better lower
     )
    (goal-conjuncts <gc> ^door-closed <gnt> + &, ^in-room <gnt2> + &, ^next-to <gnt3>
    )
    (door-closed <gnt> ^door <Room6Room5> ^args one ^goal-name <gname1>)
    (in-room <gnt2> ^box <box-d> ^room <Room2> ^args two ^goal-name <gname2>)
    (next-to <gnt3> ^box <box-d> ^door <Room5Room2> ^args two ^goal-name <gname3>)

;;; Task 2
    ;; initial state augmentations
    (next-to <nt1> ^obj1 <box-d> ^obj2 <box-a>)
    (next-to <nt2> ^obj1 <box-a> ^obj2 <box-d>)
    (next-to <nt3> ^obj1 <box-d> ^obj2 <Room4Room3>)
    (door-status <ds1> ^door <Room4Room3> ^status open)
    (door-status <ds2> ^door <Room7Room4> ^status closed)
    (door-status <ds3> ^door <Room2Room4> ^status open)
    (door-status <ds4> ^door <Room5Room4> ^status closed)
    (door-status <ds5> ^door <Room5Room7> ^status open)
    (door-status <ds6> ^door <Room2Room1> ^status open)
    (door-status <ds7> ^door <Room5Room2> ^status open)
    (door-status <ds8> ^door <Room6Room5> ^status open)
    (in-room <inr1> ^obj <box-e> ^room <Room1>)
    (in-room <inr2> ^obj <box-d> ^room <Room3>)
    (in-room <inr3> ^obj <box-c> ^room <Room6>)
    (in-room <inr4> ^obj <box-b> ^room <Room7>)
    (in-room <inr5> ^obj <box-a> ^room <Room3>)
    (in-room <inr6> ^obj <robot> ^room <Room6>)
    ;; goal statement
    (goal <g> ^desired-state <d>)
    (desired <d> ^goal-conjuncts <gc> ^better lower
     )
```

```
    (goal-conjuncts <gc> ^door-closed <gnt> + &,  <gnt2> + &, ^in-room <gnt3>
    )
    (door-closed <gnt> ^door <Room5Room7> ^args one ^goal-name <gname1>)
    (door-closed <gnt2> ^door <Room2Room1> ^args one ^goal-name <gname2>)
    (in-room <gnt3> ^box <box-b> ^room <Room4> ^args two ^goal-name <gname3>)

;;; Task 3
    ;; initial state augmentations
    (next-to <nt1> ^obj1 <box-c> ^obj2 <box-a>)
    (next-to <nt2> ^obj1 <box-a> ^obj2 <box-c>)
    (next-to <nt3> ^obj1 <box-a> ^obj2 <Room5Room2>)
    (next-to <nt4> ^obj1 <box-d> ^obj2 <Room5Room7>)
    (next-to <nt5> ^obj1 <box-b> ^obj2 <Room5Room7>)
    (door-status <ds1> ^door <Room4Room3> ^status open)
    (door-status <ds2> ^door <Room7Room4> ^status open)
    (door-status <ds3> ^door <Room2Room4> ^status open)
    (door-status <ds4> ^door <Room5Room4> ^status closed)
    (door-status <ds5> ^door <Room5Room7> ^status open)
    (door-status <ds6> ^door <Room2Room1> ^status closed)
    (door-status <ds7> ^door <Room5Room2> ^status open)
    (door-status <ds8> ^door <Room6Room5> ^status open)
    (in-room <inr1> ^obj <box-e> ^room <Room3>)
    (in-room <inr2> ^obj <box-d> ^room <Room7>)
    (in-room <inr3> ^obj <box-c> ^room <Room2>)
    (in-room <inr4> ^obj <box-b> ^room <Room5>)
    (in-room <inr5> ^obj <box-a> ^room <Room2>)
    (in-room <inr6> ^obj <robot> ^room <Room3>)
    ;; goal statement
    (goal <g> ^desired-state <d>)
    (desired <d> ^goal-conjuncts <gc> ^better lower
     )
    (goal-conjuncts <gc> ^door-closed <gnt> + &,  <gnt2> + &, ^in-room <gnt3>
    )
    (door-closed <gnt> ^door <Room4Room3> ^args one ^goal-name <gname1>)
    (door-closed <gnt2> ^door <Room2Room4> ^args one ^goal-name <gname2>)
    (in-room <gnt3> ^box <box-d> ^room <Room4> ^args two ^goal-name <gname3>)

;;; Task 4
    ;; initial state augmentations
    (next-to <nt1> ^obj1 <box-d> ^obj2 <box-b>)
    (next-to <nt2> ^obj1 <box-b> ^obj2 <box-d>)
    (next-to <nt3> ^obj1 <box-a> ^obj2 <Room5Room2>)
    (door-status <ds1> ^door <Room4Room3> ^status closed)
    (door-status <ds2> ^door <Room7Room4> ^status open)
    (door-status <ds3> ^door <Room2Room4> ^status open)
    (door-status <ds4> ^door <Room5Room4> ^status closed)
    (door-status <ds5> ^door <Room5Room7> ^status open)
    (door-status <ds6> ^door <Room2Room1> ^status open)
    (door-status <ds7> ^door <Room5Room2> ^status closed)
    (door-status <ds8> ^door <Room6Room5> ^status open)
    (in-room <inr1> ^obj <box-e> ^room <Room2>)
    (in-room <inr2> ^obj <box-d> ^room <Room7>)
    (in-room <inr3> ^obj <box-c> ^room <Room1>)
    (in-room <inr4> ^obj <box-b> ^room <Room7>)
    (in-room <inr5> ^obj <box-a> ^room <Room5>)
    (in-room <inr6> ^obj <robot> ^room <Room2>)
    ;; goal statement
    (goal <g> ^desired-state <d>)
    (desired <d> ^goal-conjuncts <gc> ^better lower
     )
    (goal-conjuncts <gc> ^door-closed <gnt> + &,
    ^in-room <gnt2> + &, <gnt3> + &
    )
    (door-closed <gnt> ^door <Room6Room5> ^args one ^goal-name <gname1>)
    (in-room <gnt2> ^box <box-d> ^room <Room1> ^args two ^goal-name <gname2>)
    (in-room <gnt3> ^box <box-a> ^room <Room1> ^args two ^goal-name <gname3>)

;;; Task 5
    ;; initial state augmentations
    (next-to <nt1> ^obj1 <box-b> ^obj2 <Room5Room7>)
```

```
    (door-status <ds1> ^door <Room4Room3> ^status open)
    (door-status <ds2> ^door <Room7Room4> ^status closed)
    (door-status <ds3> ^door <Room2Room4> ^status open)
    (door-status <ds4> ^door <Room5Room4> ^status closed)
    (door-status <ds5> ^door <Room5Room7> ^status closed)
    (door-status <ds6> ^door <Room2Room1> ^status open)
    (door-status <ds7> ^door <Room5Room2> ^status open)
    (door-status <ds8> ^door <Room6Room5> ^status open)
    (in-room <inr1> ^obj <box-e> ^room <Room3>)
    (in-room <inr2> ^obj <box-d> ^room <Room4>)
    (in-room <inr3> ^obj <box-c> ^room <Room1>)
    (in-room <inr4> ^obj <box-b> ^room <Room5>)
    (in-room <inr5> ^obj <box-a> ^room <Room2>)
    (in-room <inr6> ^obj <robot> ^room <Room4>)
    ;; goal statement
    (goal <g> ^desired-state <d>)
    (desired <d> ^goal-conjuncts <gc> ^better lower
     )
    (goal-conjuncts <gc> ^door-closed <gnt> + &,  ^next-to <gnt2> + &,
    ^in-room <gnt3>  ^in-same-room <gnt4>
    )
    (door-closed <gnt> ^door <Room4Room3> ^args one ^goal-name <gname1>)
    (next-to <gnt2> ^box <box-e> + &, <box-c> + &, ^args two
     ^goal-name <gname2>)
    (in-room <gnt3> ^robot <robot> ^room <Room5> ^args two ^goal-name <gname3>)
    ;; additional goal for problem-space invariant
    (in-same-room <gnt4> ^box <box-e> + &, <box-c> + &, ^args two
  ^goal-name <gname4>)
;;; Task 6
    ;; initial state augmentations
    (next-to <nt1> ^obj1 <box-b> ^obj2 <Room4Room3>)
    (next-to <nt2> ^obj1 <box-c> ^obj2 <Room5Room7>)
    (next-to <nt3> ^obj1 <box-d> ^obj2 <Room2Room1>)
    (door-status <ds1> ^door <Room4Room3> ^status open)
    (door-status <ds2> ^door <Room7Room4> ^status closed)
    (door-status <ds3> ^door <Room2Room4> ^status closed)
    (door-status <ds4> ^door <Room5Room4> ^status closed)
    (door-status <ds5> ^door <Room5Room7> ^status closed)
    (door-status <ds6> ^door <Room2Room1> ^status open)
    (door-status <ds7> ^door <Room5Room2> ^status open)
    (door-status <ds8> ^door <Room6Room5> ^status closed)
    (in-room <inr1> ^obj <box-e> ^room <Room7>)
    (in-room <inr2> ^obj <box-d> ^room <Room2>)
    (in-room <inr3> ^obj <box-c> ^room <Room5>)
    (in-room <inr4> ^obj <box-b> ^room <Room4>)
    (in-room <inr5> ^obj <box-a> ^room <Room1>)
    (in-room <inr6> ^obj <robot> ^room <Room4>)
    ;; goal statement
    (goal <g> ^desired-state <d>)
    (desired <d> ^goal-conjuncts <gc> ^better lower
     )
    (goal-conjuncts <gc> ^door-open <gnt> + &,  ^in-room <gnt2> + &,
    ^in-room <gnt3> + &,
    ^next-to <gnt4> + &,
    )
    (door-open <gnt> ^door <Room5Room7> ^args one ^goal-name <gname1>)
    (in-room <gnt2> ^box <box-a> ^room <Room5> ^args two ^goal-name <gname2>)
    (in-room <gnt3> ^box <box-b> ^room <Room5> ^args two ^goal-name <gname3>)
    ;; additional goal for problem-space invariant
    (next-to <gnt4> ^box <box-a> + &, <box-b> + &, ^args two
     ^goal-name <gname4>)
;;; Task 7
    ;; initial state augmentations
    (next-to <nt1> ^obj1 <box-b> ^obj2 <Room7Room4>)
    (next-to <nt2> ^obj1 <box-c> ^obj2 <Room2Room1>)
    (door-status <ds1> ^door <Room4Room3> ^status open)
    (door-status <ds2> ^door <Room7Room4> ^status closed)
    (door-status <ds3> ^door <Room2Room4> ^status open)
    (door-status <ds4> ^door <Room5Room4> ^status closed)
```

```
    (door-status <ds5> ^door <Room5Room7> ^status open)
    (door-status <ds6> ^door <Room2Room1> ^status closed)
    (door-status <ds7> ^door <Room5Room2> ^status open)
    (door-status <ds8> ^door <Room6Room5> ^status open)
    (in-room <inr1> ^obj <box-e> ^room <Room7>)
    (in-room <inr2> ^obj <box-d> ^room <Room4>)
    (in-room <inr3> ^obj <box-c> ^room <Room2>)
    (in-room <inr4> ^obj <box-b> ^room <Room7>)
    (in-room <inr5> ^obj <box-a> ^room <Room1>)
    (in-room <inr6> ^obj <robot> ^room <Room5>)
    ;; goal statement
    (goal <g> ^desired-state <d>)
    (desired <d> ^goal-conjuncts <gc> ^better lower
     )
    (goal-conjuncts <gc>
    ^in-room <gnt> + &, <gnt2> + &,
    ^in-room <gnt3> + &,
    ^next-to <gnt4> + &,
    )
    (in-room <gnt> ^box <box-a> ^room <Room2> ^args two ^goal-name <gname1>)
    (in-room <gnt2> ^box <box-b> ^room <Room6> ^args two ^goal-name <gname2>)
    ;; additional goal for problem-space invariant
    (in-room <gnt3> ^box <box-c> ^room <Room6> ^args two ^goal-name <gname3>)
    (next-to <gnt4> ^box <box-c> + &, <box-b> + &, ^args two
     ^goal-name <gname4>)
;;; Task 8
    ;; initial state augmentations
    (next-to <nt1> ^obj1 <box-d> ^obj2 <box-a>)
    (next-to <nt2> ^obj1 <box-a> ^obj2 <box-d>)
    (next-to <nt3> ^obj1 <box-d> ^obj2 <Room2Room1>)
    (next-to <nt4> ^obj1 <box-e> ^obj2 <Room5Room2>)
    (door-status <ds1> ^door <Room4Room3> ^status open)
    (door-status <ds2> ^door <Room7Room4> ^status closed)
    (door-status <ds3> ^door <Room2Room4> ^status closed)
    (door-status <ds4> ^door <Room5Room4> ^status closed)
    (door-status <ds5> ^door <Room5Room7> ^status closed)
    (door-status <ds6> ^door <Room2Room1> ^status closed)
    (door-status <ds7> ^door <Room5Room2> ^status closed)
    (door-status <ds8> ^door <Room6Room5> ^status open)
    (in-room <inr1> ^obj <box-e> ^room <Room5>)
    (in-room <inr2> ^obj <box-d> ^room <Room1>)
    (in-room <inr3> ^obj <box-c> ^room <Room7>)
    (in-room <inr4> ^obj <box-b> ^room <Room7>)
    (in-room <inr5> ^obj <box-a> ^room <Room1>)
    (in-room <inr6> ^obj <robot> ^room <Room2>)
    ;; goal statement
    (goal <g> ^desired-state <d>)
    (desired <d> ^goal-conjuncts <gc> ^better lower
     )
    (goal-conjuncts <gc> ^in-room <gnt> + &, <gnt2> + &, <gnt3> + &,
    )
    (in-room <gnt> ^box <box-c> ^room <Room6> ^args two ^goal-name <gname1>)
    (in-room <gnt2> ^box <box-a> ^room <Room7> ^args two ^goal-name <gname2>)
    (in-room <gnt3> ^box <box-d> ^room <Room7> ^args two ^goal-name <gname3>)
;;; Task  9
    ;; initial state augmentations
    (next-to <nt1> ^obj1 <box-b> ^obj2 <Room4Room3>)
    (next-to <nt2> ^obj1 <box-a> ^obj2 <Room5Room4>)
    (door-status <ds1> ^door <Room4Room3> ^status closed)
    (door-status <ds2> ^door <Room7Room4> ^status open)
    (door-status <ds3> ^door <Room2Room4> ^status closed)
    (door-status <ds4> ^door <Room5Room4> ^status open)
    (door-status <ds5> ^door <Room5Room7> ^status closed)
    (door-status <ds6> ^door <Room2Room1> ^status open)
    (door-status <ds7> ^door <Room5Room2> ^status open)
    (door-status <ds8> ^door <Room6Room5> ^status open)
    (in-room <inr1> ^obj <box-e> ^room <Room6>)
    (in-room <inr2> ^obj <box-d> ^room <Room7>)
    (in-room <inr3> ^obj <box-c> ^room <Room5>)
```

```
    (in-room <inr4> ^obj <box-b> ^room <Room3>)
    (in-room <inr5> ^obj <box-a> ^room <Room4>)
    (in-room <inr6> ^obj <robot> ^room <Room1>)
    ;; goal statement
    (goal <g> ^desired-state <d>)
    (desired <d> ^goal-conjuncts <gc> ^better lower
     )
    (goal-conjuncts <gc> ^door-open <gnt> + &, <gnt2> + &,
    ^next-to <gnt3> + &, ^in-same-room <gnt4>
    )
    (door-open <gnt> ^door <Room4Room3> ^args one ^goal-name <gname1>)
    (door-open <gnt2> ^door <Room5Room7> ^args one ^goal-name <gname2>)
    (next-to <gnt3> ^box <box-c> + &, <box-a> + &,
     ^args two ^goal-name <gname3>)
    ;; additional goal for problem-space invariant
    (in-same-room <gnt4> ^box <box-c> + &, <box-a> + &,
     ^args two ^goal-name <gname4>)

;;; Task 10
    ;; initial state augmentations
    (next-to <nt1> ^obj1 <box-b> ^obj2 <box-a>)
    (next-to <nt2> ^obj1 <box-a> ^obj2 <box-b>)
    (next-to <nt3> ^obj1 <box-b> ^obj2 <box-e>)
    (next-to <nt4> ^obj1 <box-e> ^obj2 <box-b>)
    (next-to <nt5> ^obj1 <box-e> ^obj2 <box-c>)
    (next-to <nt6> ^obj1 <box-c> ^obj2 <box-e>)
    (next-to <nt7> ^obj1 <box-a> ^obj2 <Room2Room4>)
    (next-to <nt8> ^obj1 <box-e> ^obj2 <Room2Room4>)
    (next-to <nt9> ^obj1 <box-c> ^obj2 <Room2Room1>)
    (next-to <nt10> ^obj1 <box-d> ^obj2 <Room4Room3>)
    (door-status <ds1> ^door <Room4Room3> ^status open)
    (door-status <ds2> ^door <Room7Room4> ^status closed)
    (door-status <ds3> ^door <Room2Room4> ^status open)
    (door-status <ds4> ^door <Room5Room4> ^status closed)
    (door-status <ds5> ^door <Room5Room7> ^status open)
    (door-status <ds6> ^door <Room2Room1> ^status closed)
    (door-status <ds7> ^door <Room5Room2> ^status closed)
    (door-status <ds8> ^door <Room6Room5> ^status open)
    (in-room <inr1> ^obj <box-e> ^room <Room2>)
    (in-room <inr2> ^obj <box-d> ^room <Room3>)
    (in-room <inr3> ^obj <box-c> ^room <Room2>)
    (in-room <inr4> ^obj <box-b> ^room <Room2>)
    (in-room <inr5> ^obj <box-a> ^room <Room2>)
    (in-room <inr6> ^obj <robot> ^room <Room5>)
    ;; goal statement
    (goal <g> ^desired-state <d>)
    (desired <d> ^goal-conjuncts <gc> ^better lower
     )
    (goal-conjuncts <gc> ^door-open <gnt> + &,
    ^in-room <gnt3> + &,
    <gnt2> + &,
    ^next-to <gnt4> + &,
    )
    (door-open <gnt> ^door <Room2Room1> ^args one ^goal-name <gname1>)
    (in-room <gnt3> ^box <box-a> ^room <Room3> ^args two ^goal-name <gname3>)
    (in-room <gnt2> ^box <box-c> ^room <Room3> ^args two ^goal-name <gname2>)
    ;; additional goal for problem-space invariant
    (next-to <gnt4> ^box <box-c> + &, <box-a> + &, ^args two
     ^goal-name <gname4>)
;;; Task 11
    ;; initial state augmentations
    (next-to <nt1> ^obj1 <box-e> ^obj2 <box-d>)
    (next-to <nt2> ^obj1 <box-d> ^obj2 <box-e>)
    (next-to <nt3> ^obj1 <box-e> ^obj2 <Room2Room1>)
    (next-to <nt4> ^obj1 <box-d> ^obj2 <Room2Room1>)
    (next-to <nt5> ^obj1 <box-b> ^obj2 <Room6Room5>)
    (door-status <ds1> ^door <Room4Room3> ^status open)
    (door-status <ds2> ^door <Room7Room4> ^status open)
    (door-status <ds3> ^door <Room2Room4> ^status closed)
    (door-status <ds4> ^door <Room5Room4> ^status open)
```

```
    (door-status <ds5> ^door <Room5Room7> ^status closed)
    (door-status <ds6> ^door <Room2Room1> ^status open)
    (door-status <ds7> ^door <Room5Room2> ^status closed)
    (door-status <ds8> ^door <Room6Room5> ^status open)
    (in-room <inr1> ^obj <box-e> ^room <Room1>)
    (in-room <inr2> ^obj <box-d> ^room <Room1>)
    (in-room <inr3> ^obj <box-c> ^room <Room7>)
    (in-room <inr4> ^obj <box-b> ^room <Room5>)
    (in-room <inr5> ^obj <box-a> ^room <Room6>)
    (in-room <inr6> ^obj <robot> ^room <Room6>)
    ;; goal statement
    (goal <g> ^desired-state <d>)
    (desired <d> ^goal-conjuncts <gc> ^better lower
     )
    (goal-conjuncts <gc> ^door-closed <gnt> + &,
    ^next-to <gnt3> + &, <gnt2> + &,
    )
    (door-closed <gnt> ^door <Room7Room4> ^args one ^goal-name <gname1>)
    (next-to <gnt2> ^box <box-e> + &, <box-a> + &, ^args two
     ^goal-name <gname2>)
    (next-to <gnt3> ^box <box-c> + &, <box-b> + &, ^args two
     ^goal-name <gname3>)
;; Task 12
    ;; initial state augmentations
    (next-to <nt1> ^obj1 <box-b> ^obj2 <box-c>)
    (next-to <nt2> ^obj1 <box-c> ^obj2 <box-b>)
    (next-to <nt3> ^obj1 <box-c> ^obj2 <Room5Room7>)
    (door-status <ds1> ^door <Room4Room3> ^status open)
    (door-status <ds2> ^door <Room7Room4> ^status open)
    (door-status <ds3> ^door <Room2Room4> ^status closed)
    (door-status <ds4> ^door <Room5Room4> ^status closed)
    (door-status <ds5> ^door <Room5Room7> ^status closed)
    (door-status <ds6> ^door <Room2Room1> ^status open)
    (door-status <ds7> ^door <Room5Room2> ^status closed)
    (door-status <ds8> ^door <Room6Room5> ^status open)
    (in-room <inr1> ^obj <box-e> ^room <Room6>)
    (in-room <inr2> ^obj <box-d> ^room <Room1>)
    (in-room <inr3> ^obj <box-c> ^room <Room5>)
    (in-room <inr4> ^obj <box-b> ^room <Room5>)
    (in-room <inr5> ^obj <box-a> ^room <Room2>)
    (in-room <inr6> ^obj <robot> ^room <Room2>)
    ;; goal statement
    (goal <g> ^desired-state <d>)
    (desired <d> ^goal-conjuncts <gc> ^better lower
     )
    (goal-conjuncts <gc> ^door-closed <gnt> + &,
    ^door-open <gnt2> + &,
    ^in-room <gnt3>
    )
    (door-closed <gnt> ^door <Room6Room5> ^args one ^goal-name <gname1>)
    (door-open <gnt2> ^door <Room5Room2> ^args one ^goal-name <gname2>)
    (in-room <gnt3> ^box <box-a> ^room <Room5> ^args two ^goal-name <gname3>)
;;; Task 13
    ;; initial state augmentations
    (next-to <nt1> ^obj1 <box-b> ^obj2 <box-d>)
    (next-to <nt2> ^obj1 <box-d> ^obj2 <box-b>)
    (next-to <nt3> ^obj1 <box-d> ^obj2 <box-c>)
    (next-to <nt4> ^obj1 <box-c> ^obj2 <box-d>)
    (next-to <nt5> ^obj1 <box-a> ^obj2 <Room6Room5>)
    (door-status <ds1> ^door <Room4Room3> ^status closed)
    (door-status <ds2> ^door <Room7Room4> ^status closed)
    (door-status <ds3> ^door <Room2Room4> ^status open)
    (door-status <ds4> ^door <Room5Room4> ^status open)
    (door-status <ds5> ^door <Room5Room7> ^status open)
    (door-status <ds6> ^door <Room2Room1> ^status open)
    (door-status <ds7> ^door <Room5Room2> ^status open)
    (door-status <ds8> ^door <Room6Room5> ^status closed)
    (in-room <inr1> ^obj <box-e> ^room <Room7>)
```

```
    (in-room <inr2> ^obj <box-d> ^room <Room1>)
    (in-room <inr3> ^obj <box-c> ^room <Room1>)
    (in-room <inr4> ^obj <box-b> ^room <Room1>)
    (in-room <inr5> ^obj <box-a> ^room <Room6>)
    (in-room <inr6> ^obj <robot> ^room <Room1>)
    ;; goal statement
    (goal <g> ^desired-state <d>)
    (desired <d> ^goal-conjuncts <gc> ^better lower
     )
    (goal-conjuncts <gc> ^door-closed <gnt> + &,
     ^in-room  <gnt2> + &,
    ^next-to <gnt3> + &, ^in-same-room <gnt4>
    )
    (door-closed <gnt> ^door <Room5Room2> ^args one ^goal-name <gname1>)
    (in-room <gnt2> ^robot <robot> ^room <Room6> ^args two ^goal-name <gname2>)
    (next-to <gnt3> ^box <box-e> + &, <box-b> + &,
     ^args two ^goal-name <gname3>)
    ;; additional goal for problem-space invariant
    (in-same-room <gnt4> ^box <box-e> + &, <box-b> + &,
  ^args two ^goal-name <gname4>)
;;; Task 14
    ;; initial state augmentations
    (next-to <nt1> ^obj1 <box-e> ^obj2 <box-d>)
    (next-to <nt2> ^obj1 <box-d> ^obj2 <box-e>)
    (next-to <nt3> ^obj1 <box-d> ^obj2 <Room2Room4>)
    (next-to <nt4> ^obj1 <box-c> ^obj2 <Room5Room2>)
    (door-status <ds1> ^door <Room4Room3> ^status closed)
    (door-status <ds2> ^door <Room7Room4> ^status closed)
    (door-status <ds3> ^door <Room2Room4> ^status open)
    (door-status <ds4> ^door <Room5Room4> ^status closed)
    (door-status <ds5> ^door <Room5Room7> ^status closed)
    (door-status <ds6> ^door <Room2Room1> ^status closed)
    (door-status <ds7> ^door <Room5Room2> ^status closed)
    (door-status <ds8> ^door <Room6Room5> ^status closed)
    (in-room <inr1> ^obj <box-e> ^room <Room4>)
    (in-room <inr2> ^obj <box-d> ^room <Room4>)
    (in-room <inr3> ^obj <box-c> ^room <Room5>)
    (in-room <inr4> ^obj <box-b> ^room <Room2>)
    (in-room <inr5> ^obj <box-a> ^room <Room3>)
    (in-room <inr6> ^obj <robot> ^room <Room7>)
    ;; goal statement
    (goal <g> ^desired-state <d>)
    (desired <d> ^goal-conjuncts <gc> ^better lower
     )
    (goal-conjuncts <gc> ^door-open <gnt> + &,
     ^in-room  <gnt2> + &, <gnt3> + &,
    ^next-to <gnt4> + &,
    )
    (door-open <gnt> ^door <Room2Room1> ^args one ^goal-name <gname1>)
    (in-room <gnt2> ^box <box-d> ^room <Room5> ^args two ^goal-name <gname2>)
    (next-to <gnt4> ^box <box-a> + &, <box-d> + &,
     ^args two ^goal-name <gname4>)
    ;; additional goal for problem-space invariant
    (in-room <gnt3> ^box <box-a> ^room <Room5> ^args two ^goal-name <gname3>)

;;; Task 15
    ;; initial state augmentations
    (next-to <nt1> ^obj1 <box-c> ^obj2 <box-a>)
    (next-to <nt2> ^obj1 <box-a> ^obj2 <box-c>)
    (next-to <nt3> ^obj1 <box-e> ^obj2 <Room4Room3>)
    (next-to <nt4> ^obj1 <box-c> ^obj2 <Room2Room4>)
    (door-status <ds1> ^door <Room4Room3> ^status closed)
    (door-status <ds2> ^door <Room7Room4> ^status closed)
    (door-status <ds3> ^door <Room2Room4> ^status closed)
    (door-status <ds4> ^door <Room5Room4> ^status closed)
    (door-status <ds5> ^door <Room5Room7> ^status closed)
    (door-status <ds6> ^door <Room2Room1> ^status open)
    (door-status <ds7> ^door <Room5Room2> ^status open)
    (door-status <ds8> ^door <Room6Room5> ^status open)
    (in-room <inr1> ^obj <box-e> ^room <Room4>)
```

```
    (in-room <inr2> ^obj <box-d> ^room <Room3>)
    (in-room <inr3> ^obj <box-c> ^room <Room2>)
    (in-room <inr4> ^obj <box-b> ^room <Room6>)
    (in-room <inr5> ^obj <box-a> ^room <Room2>)
    (in-room <inr6> ^obj <robot> ^room <Room7>)
    ;; goal statement
    (goal <g> ^desired-state <d>)
    (desired <d> ^goal-conjuncts <gc> ^better lower
     )
    (goal-conjuncts <gc> ^door-open <gnt> + &,
     ^in-room  <gnt2> + &,
    ^next-to <gnt3> + &,
    )
    (door-open <gnt> ^door <Room5Room4> ^args one ^goal-name <gname1>)
    (in-room <gnt2> ^box <box-d> ^room <Room7> ^args two ^goal-name <gname2>)
    (next-to <gnt3> ^box <box-b> ^door <Room5Room2>
     ^args two ^goal-name <gname3>)
```

D.5 4-goal-conjunct tasks

The following pieces of Soar code describe the initial and goal states for the 4-goal-conjunct Robot Domain tasks used in Chapter 6. For each task, the listing first describes the relevant initial state augmentations for the task, and then the task goal.

```
;;; Task 1
    ;; initial state augmentations
    (next-to <nt1> ^obj1 <box-c> ^obj2 <box-b>)
    (next-to <nt2> ^obj1 <box-b> ^obj2 <box-c>)
    (next-to <nt3> ^obj1 <box-e> ^obj2 <Room7Room4>)
    (next-to <nt4> ^obj1 <box-a> ^obj2 <Room5Room7>)
    (door-status <ds1> ^door <Room4Room3> ^status closed)
    (door-status <ds2> ^door <Room7Room4> ^status open)
    (door-status <ds3> ^door <Room2Room4> ^status open)
    (door-status <ds4> ^door <Room5Room4> ^status open)
    (door-status <ds5> ^door <Room5Room7> ^status closed)
    (door-status <ds6> ^door <Room2Room1> ^status closed)
    (door-status <ds7> ^door <Room5Room2> ^status open)
    (door-status <ds8> ^door <Room6Room5> ^status open)
    (in-room <inr1> ^obj <box-e> ^room <Room4>)
    (in-room <inr2> ^obj <box-d> ^room <Room4>)
    (in-room <inr3> ^obj <box-c> ^room <Room1>)
    (in-room <inr4> ^obj <box-b> ^room <Room1>)
    (in-room <inr5> ^obj <box-a> ^room <Room5>)
    (in-room <inr6> ^obj <robot> ^room <Room3>)
    ;; goal statement
    (goal <g> ^desired-state <d>)
    (desired <d> ^goal-conjuncts <gc> ^better lower
     )
    (goal-conjuncts <gc> ^door-open <gnt> + &, <gnt4> + &,
    ^in-room <gnt2> + &, <gnt3> + &,
    )
    (door-open <gnt> ^door <Room2Room1> ^args one ^goal-name <gname1>)
    (in-room <gnt2> ^box <box-b> ^room <Room5> ^args two ^goal-name <gname2>)
    (in-room <gnt3> ^box <box-e> ^room <Room2> ^args two ^goal-name <gname3>)
    (door-open <gnt4> ^door <Room4Room3> ^args one ^goal-name <gname4>)
;;; Task 2
    ;; initial state augmentations
    (next-to <nt1> ^obj1 <box-c> ^obj2 <Room5Room7>)
    (door-status <ds1> ^door <Room4Room3> ^status closed)
```

```
    (door-status <ds2> ^door <Room7Room4> ^status closed)
    (door-status <ds3> ^door <Room2Room4> ^status open)
    (door-status <ds4> ^door <Room5Room4> ^status closed)
    (door-status <ds5> ^door <Room5Room7> ^status open)
    (door-status <ds6> ^door <Room2Room1> ^status closed)
    (door-status <ds7> ^door <Room5Room2> ^status closed)
    (door-status <ds8> ^door <Room6Room5> ^status closed)
    (in-room <inr1> ^obj <box-e> ^room <Room3>)
    (in-room <inr2> ^obj <box-d> ^room <Room7>)
    (in-room <inr3> ^obj <box-c> ^room <Room5>)
    (in-room <inr4> ^obj <box-b> ^room <Room7>)
    (in-room <inr5> ^obj <box-a> ^room <Room1>)
    (in-room <inr6> ^obj <robot> ^room <Room4>)
    ;; goal statement
    (goal <g> ^desired-state <d>)
    (desired <d> ^goal-conjuncts <gc> ^better lower
     )
    (goal-conjuncts <gc> ^door-open <gnt> ^in-room <gnt2> + &, <gnt3> + &,
    <gnt4> + &,
    )
    (door-open <gnt> ^door <Room6Room5> ^args one ^goal-name <gname1>)
    (in-room <gnt2> ^box <box-e> ^room <Room7> ^args two ^goal-name <gname2>)
    (in-room <gnt3> ^robot <robot> ^room <Room5> ^args two ^goal-name <gname3>)
    (in-room <gnt4> ^box <box-a>  ^room <Room6> ^args two ^goal-name <gname4>)

;;; Task 3
    ;; initial state augmentations
    (next-to <nt1> ^obj1 <box-d> ^obj2 <box-b>)
    (next-to <nt2> ^obj1 <box-b> ^obj2 <box-d>)
    (next-to <nt3> ^obj1 <box-e> ^obj2 <Room5Room4>)
    (door-status <ds1> ^door <Room4Room3> ^status open)
    (door-status <ds2> ^door <Room7Room4> ^status closed)
    (door-status <ds3> ^door <Room2Room4> ^status open)
    (door-status <ds4> ^door <Room5Room4> ^status open)
    (door-status <ds5> ^door <Room5Room7> ^status closed)
    (door-status <ds6> ^door <Room2Room1> ^status closed)
    (door-status <ds7> ^door <Room5Room2> ^status closed)
    (door-status <ds8> ^door <Room6Room5> ^status closed)
    (in-room <inr1> ^obj <box-e> ^room <Room4>)
    (in-room <inr2> ^obj <box-d> ^room <Room7>)
    (in-room <inr3> ^obj <box-c> ^room <Room5>)
    (in-room <inr4> ^obj <box-b> ^room <Room7>)
    (in-room <inr5> ^obj <box-a> ^room <Room2>)
    (in-room <inr6> ^obj <robot> ^room <Room6>)
    ;; goal statement
    (goal <g> ^desired-state <d>)
    (desired <d> ^goal-conjuncts <gc> ^better lower
     )
    (goal-conjuncts <gc> ^in-room <gnt> + &, <gnt4> + &,
    ^next-to <gnt3> + &, <gnt2> + &,
    )
    (in-room <gnt> ^box <box-e> ^room <Room5> ^args two ^goal-name <gname1>)
    (next-to <gnt2> ^box <box-b> ^door <Room4Room3> ^args two
     ^goal-name <gname2>)
    (next-to <gnt3> ^box <box-a> + &, <box-d> + &, ^args two
     ^goal-name <gname3>)
    (in-room <gnt4> ^box <box-c>  ^room <Room7> ^args two ^goal-name <gname4>)

;;; Task 4
    ;; initial state augmentations
    (next-to <nt1> ^obj1 <box-d> ^obj2 <box-e>)
    (next-to <nt2> ^obj1 <box-e> ^obj2 <box-d>)
    (next-to <nt3> ^obj1 <box-b> ^obj2 <box-c>)
    (next-to <nt4> ^obj1 <box-c> ^obj2 <box-b>)
    (next-to <nt5> ^obj1 <box-a> ^obj2 <Room5Room2>)
    (door-status <ds1> ^door <Room4Room3> ^status open)
    (door-status <ds2> ^door <Room7Room4> ^status open)
    (door-status <ds3> ^door <Room2Room4> ^status open)
    (door-status <ds4> ^door <Room5Room4> ^status closed)
    (door-status <ds5> ^door <Room5Room7> ^status closed)
```

```
    (door-status <ds6> ^door <Room2Room1> ^status closed)
    (door-status <ds7> ^door <Room5Room2> ^status open)
    (door-status <ds8> ^door <Room6Room5> ^status open)
    (in-room <inr1> ^obj <box-e> ^room <Room4>)
    (in-room <inr2> ^obj <box-d> ^room <Room4>)
    (in-room <inr3> ^obj <box-c> ^room <Room1>)
    (in-room <inr4> ^obj <box-b> ^room <Room1>)
    (in-room <inr5> ^obj <box-a> ^room <Room2>)
    (in-room <inr6> ^obj <robot> ^room <Room6>)
    ;; goal statement
    (goal <g> ^desired-state <d>)
    (desired <d> ^goal-conjuncts <gc> ^better lower
     )
    (goal-conjuncts <gc> ^door-open <gnt> + &, <gnt2> + &, <gnt3> + &,
    ^next-to <gnt4>
    )
    (door-open <gnt> ^door <Room5Room7> ^args one ^goal-name <gname1>)
    (door-open <gnt2> ^door <Room2Room1> ^args one ^goal-name <gname2>)
    (door-open <gnt3> ^door <Room5Room4> ^args one ^goal-name <gname3>)
    (next-to <gnt4> ^box <box-e> + &, <box-c> + &, ^args two
     ^goal-name <gname4>)

;;; Task 5
    ;; initial state augmentations
    (next-to <nt1> ^obj1 <box-b> ^obj2 <box-e>)
    (next-to <nt2> ^obj1 <box-e> ^obj2 <box-b>)
    (next-to <nt3> ^obj1 <box-c> ^obj2 <Room2Room1>)
    (door-status <ds1> ^door <Room4Room3> ^status open)
    (door-status <ds2> ^door <Room7Room4> ^status open)
    (door-status <ds3> ^door <Room2Room4> ^status closed)
    (door-status <ds4> ^door <Room5Room4> ^status closed)
    (door-status <ds5> ^door <Room5Room7> ^status closed)
    (door-status <ds6> ^door <Room2Room1> ^status closed)
    (door-status <ds7> ^door <Room5Room2> ^status closed)
    (door-status <ds8> ^door <Room6Room5> ^status closed)
    (in-room <inr1> ^obj <box-e> ^room <Room7>)
    (in-room <inr2> ^obj <box-d> ^room <Room3>)
    (in-room <inr3> ^obj <box-c> ^room <Room2>)
    (in-room <inr4> ^obj <box-b> ^room <Room7>)
    (in-room <inr5> ^obj <box-a> ^room <Room4>)
    (in-room <inr6> ^obj <robot> ^room <Room6>)
    ;; goal statement
    (goal <g> ^desired-state <d>)
    (desired <d> ^goal-conjuncts <gc> ^better lower
     )
    (goal-conjuncts <gc> ^next-to <gnt> + &, <gnt3> + &,
    ^in-room <gnt2> + &, <gnt4> + &
    )
    (next-to <gnt> ^box <box-a> + &, <box-b> + &, ^args two
     ^goal-name <gname1>)
    (in-room <gnt2> ^box <box-a> ^room <Room1> ^args two ^goal-name <gname2>)
    (next-to  <gnt3> ^box <box-e> ^door <Room2Room4> ^args two
      ^goal-name <gname3>)
    (in-room <gnt4> ^box <box-b> ^room <Room1> ^args two ^goal-name <gname4>)

;;; Task 6
    ;; initial state augmentations
    (next-to <nt1> ^obj1 <box-d> ^obj2 <box-a>)
    (next-to <nt2> ^obj1 <box-a> ^obj2 <box-d>)
    (next-to <nt3> ^obj1 <box-c> ^obj2 <Room5Room2>)
    (door-status <ds1> ^door <Room4Room3> ^status open)
    (door-status <ds2> ^door <Room7Room4> ^status open)
    (door-status <ds3> ^door <Room2Room4> ^status closed)
    (door-status <ds4> ^door <Room5Room4> ^status closed)
    (door-status <ds5> ^door <Room5Room7> ^status open)
    (door-status <ds6> ^door <Room2Room1> ^status open)
    (door-status <ds7> ^door <Room5Room2> ^status open)
    (door-status <ds8> ^door <Room6Room5> ^status open)
    (in-room <inr1> ^obj <box-e> ^room <Room1>)
    (in-room <inr2> ^obj <box-d> ^room <Room5>)
```

```
    (in-room <inr3> ^obj <box-c> ^room <Room2>)
    (in-room <inr4> ^obj <box-b> ^room <Room5>)
    (in-room <inr5> ^obj <box-a> ^room <Room5>)
    (in-room <inr6> ^obj <robot> ^room <Room1>)
    ;; goal statement
    (goal <g> ^desired-state <d>)
    (desired <d> ^goal-conjuncts <gc> ^better lower
     )
    (goal-conjuncts <gc> ^door-closed <gnt> + &, ^door-closed <gnt2> + &,
    ^in-room <gnt3> + &,^in-room <gnt4> + &
    )
    (door-closed <gnt> ^door <Room6Room5> ^args one ^goal-name <gname1>)
    (door-closed <gnt2> ^door <Room5Room7> ^args one ^goal-name <gname2>)
    (in-room <gnt3> ^robot <robot> ^room <Room7> ^args two ^goal-name <gname3>)
    (in-room <gnt4> ^box <box-d> ^room <Room2> ^args two ^goal-name <gname4>)

;;; Task 7
    ;; initial state augmentations
    (next-to <nt1> ^obj1 <box-d> ^obj2 <box-b>)
    (next-to <nt2> ^obj1 <box-b> ^obj2 <box-d>)
    (next-to <nt3> ^obj1 <box-e> ^obj2 <Room7Room4>)
    (next-to <nt4> ^obj1 <box-c> ^obj2 <Room4Room3>)
    (door-status <ds1> ^door <Room4Room3> ^status closed)
    (door-status <ds2> ^door <Room7Room4> ^status closed)
    (door-status <ds3> ^door <Room2Room4> ^status open)
    (door-status <ds4> ^door <Room5Room4> ^status open)
    (door-status <ds5> ^door <Room5Room7> ^status closed)
    (door-status <ds6> ^door <Room2Room1> ^status closed)
    (door-status <ds7> ^door <Room5Room2> ^status closed)
    (door-status <ds8> ^door <Room6Room5> ^status open)
    (in-room <inr1> ^obj <box-e> ^room <Room7>)
    (in-room <inr2> ^obj <box-d> ^room <Room6>)
    (in-room <inr3> ^obj <box-c> ^room <Room5>)
    (in-room <inr4> ^obj <box-b> ^room <Room6>)
    (in-room <inr5> ^obj <box-a> ^room <Room4>)
    (in-room <inr6> ^obj <robot> ^room <Room2>)
    ;; goal statement
    (goal <g> ^desired-state <d>)
    (desired <d> ^goal-conjuncts <gc> ^better lower
     )
    (goal-conjuncts <gc> ^in-room <gnt> + &, <gnt3> + &, <gnt4> + &,
    ^door-closed <gnt2> ^next-to <gnt5>)
    (in-room <gnt> ^box <box-d> ^room <Room4> ^args two ^goal-name <gname1>)
    (door-closed <gnt2> ^door <Room5Room4> ^args one ^goal-name <gname2>)
    (in-room <gnt3> ^box <box-a> ^room <Room3> ^args two ^goal-name <gname3>)
    ;; additional goal for problem-space invariant
    (in-room <gnt4> ^box <box-c> ^room <Room3> ^args two ^goal-name <gname4>)
    (next-to <gnt5> ^box <box-a> + &, <box-c> + &, ^args two ^goal-name <gname5>)

;;; Task 8
    ;; initial state augmentations
    (next-to <nt1> ^obj1 <box-a> ^obj2 <box-b>)
    (next-to <nt2> ^obj1 <box-b> ^obj2 <box-a>)
    (next-to <nt3> ^obj1 <box-e> ^obj2 <Room2Room1>)
    (door-status <ds1> ^door <Room4Room3> ^status closed)
    (door-status <ds2> ^door <Room7Room4> ^status closed)
    (door-status <ds3> ^door <Room2Room4> ^status open)
    (door-status <ds4> ^door <Room5Room4> ^status closed)
    (door-status <ds5> ^door <Room5Room7> ^status closed)
    (door-status <ds6> ^door <Room2Room1> ^status closed)
    (door-status <ds7> ^door <Room5Room2> ^status closed)
    (door-status <ds8> ^door <Room6Room5> ^status closed)
    (in-room <inr1> ^obj <box-e> ^room <Room2>)
    (in-room <inr2> ^obj <box-d> ^room <Room1>)
    (in-room <inr3> ^obj <box-c> ^room <Room5>)
    (in-room <inr4> ^obj <box-b> ^room <Room3>)
    (in-room <inr5> ^obj <box-a> ^room <Room3>)
    (in-room <inr6> ^obj <robot> ^room <Room1>)
    ;; desired state
    (goal <g> ^desired-state <d>)
```

```
    (desired <d> ^goal-conjuncts <gc> ^better lower
     )
    (goal-conjuncts <gc> ^in-room <gnt> + &, <gnt3> + &,
    <gnt4> + &,
    ^door-closed <gnt2> ^next-to <gnt5>)
    (in-room <gnt> ^box <box-e> ^room <Room3> ^args two ^goal-name <gname1>)
    (door-closed <gnt2> ^door <Room2Room4> ^args one ^goal-name <gname2>)
    (in-room <gnt3> ^box <box-c> ^room <Room3> ^args two ^goal-name <gname3>)
    ;; additional goal for problem-space invariant
    (in-room <gnt4> ^box <box-d> ^room <Room6> ^args two ^goal-name <gname4>)
    (next-to <gnt5> ^box <box-e> + &, <box-c> + &, ^args two
     ^goal-name <gname5>)
;;; Task 10
    ;; initial state augmentations
    (next-to <nt1> ^obj1 <box-a> ^obj2 <box-e>)
    (next-to <nt2> ^obj1 <box-e> ^obj2 <box-a>)
    (next-to <nt3> ^obj1 <box-e> ^obj2 <Room2Room1>)
    (next-to <nt4> ^obj1 <box-a> ^obj2 <Room5Room2>)
    (next-to <nt5> ^obj1 <box-d> ^obj2 <Room2Room4>)
    (next-to <nt6> ^obj1 <box-d> ^obj2 <box-b>)
    (next-to <nt7> ^obj1 <box-b> ^obj2 <box-d>)
    (door-status <ds1> ^door <Room4Room3> ^status closed)
    (door-status <ds2> ^door <Room7Room4> ^status closed)
    (door-status <ds3> ^door <Room2Room4> ^status open)
    (door-status <ds4> ^door <Room5Room4> ^status open)
    (door-status <ds5> ^door <Room5Room7> ^status closed)
    (door-status <ds6> ^door <Room2Room1> ^status open)
    (door-status <ds7> ^door <Room5Room2> ^status closed)
    (door-status <ds8> ^door <Room6Room5> ^status open)
    (in-room <inr1> ^obj <box-e> ^room <Room2>)
    (in-room <inr2> ^obj <box-d> ^room <Room4>)
    (in-room <inr3> ^obj <box-c> ^room <Room1>)
    (in-room <inr4> ^obj <box-b> ^room <Room4>)
    (in-room <inr5> ^obj <box-a> ^room <Room2>)
    (in-room <inr6> ^obj <robot> ^room <Room1>)
    ;; goal statement
    (goal <g> ^desired-state <d>)
    (desired <d> ^goal-conjuncts <gc> ^better lower
     )
    (goal-conjuncts <gc> ^in-room <gnt> + &, ^next-to <gnt2> + &,
    ^door-open <gnt3> + &,
    ^door-closed <gnt4>
    )
    (in-room <gnt> ^box <box-b> ^room <Room5> ^args two ^goal-name <gname1>)
    (door-closed <gnt4> ^door <Room2Room1> ^args one ^goal-name <gname4>)
    (door-open <gnt3> ^door <Room5Room2> ^args one ^goal-name <gname3>)
    (next-to <gnt2> ^box <box-a> + &, <box-c> + &, ^args two
     ^goal-name <gname2>)

;;; Task 11
    ;; initial state augmentations
    (next-to <nt1> ^obj1 <box-a> ^obj2 <box-c>)
    (next-to <nt2> ^obj1 <box-c> ^obj2 <box-a>)
    (next-to <nt3> ^obj1 <box-c> ^obj2 <Room7Room4>)
    (next-to <nt4> ^obj1 <box-a> ^obj2 <Room7Room4>)
    (door-status <ds1> ^door <Room4Room3> ^status closed)
    (door-status <ds2> ^door <Room7Room4> ^status closed)
    (door-status <ds3> ^door <Room2Room4> ^status closed)
    (door-status <ds4> ^door <Room5Room4> ^status open)
    (door-status <ds5> ^door <Room5Room7> ^status closed)
    (door-status <ds6> ^door <Room2Room1> ^status closed)
    (door-status <ds7> ^door <Room5Room2> ^status open)
    (door-status <ds8> ^door <Room6Room5> ^status open)
    (in-room <inr1> ^obj <box-e> ^room <Room1>)
    (in-room <inr2> ^obj <box-d> ^room <Room3>)
    (in-room <inr3> ^obj <box-c> ^room <Room7>)
    (in-room <inr4> ^obj <box-b> ^room <Room5>)
    (in-room <inr5> ^obj <box-a> ^room <Room7>)
    (in-room <inr6> ^obj <robot> ^room <Room7>)
    ;; goal statement
```

```
    (goal <g> ^desired-state <d>)
    (desired <d> ^goal-conjuncts <gc> ^better lower
     )
    (goal-conjuncts <gc> ^in-room <gnt> + &, ^in-room <gnt2> + &,
    ^next-to <gnt3> + &,
    )
    (in-room <gnt> ^box <box-b> ^room <Room3> ^args two ^goal-name <gname1>)
    (in-room <gnt2> ^box <box-a> ^room <Room2> ^args two ^goal-name <gname2>)
    (next-to <gnt3> ^box <box-e> + &, <box-d> + &, ^args two
     ^goal-name <gname3>)
;;; Task 12
    ;; initial state augmentations
    (next-to <nt1> ^obj1 <box-a> ^obj2 <box-b>)
    (next-to <nt2> ^obj1 <box-b> ^obj2 <box-a>)
    (next-to <nt3> ^obj1 <box-a> ^obj2 <Room7Room4>)
    (door-status <ds1> ^door <Room4Room3> ^status closed)
    (door-status <ds2> ^door <Room7Room4> ^status open)
    (door-status <ds3> ^door <Room2Room4> ^status open)
    (door-status <ds4> ^door <Room5Room4> ^status open)
    (door-status <ds5> ^door <Room5Room7> ^status closed)
    (door-status <ds6> ^door <Room2Room1> ^status closed)
    (door-status <ds7> ^door <Room5Room2> ^status closed)
    (door-status <ds8> ^door <Room6Room5> ^status open)
    (in-room <inr1> ^obj <box-e> ^room <Room2>)
    (in-room <inr2> ^obj <box-d> ^room <Room3>)
    (in-room <inr3> ^obj <box-c> ^room <Room7>)
    (in-room <inr4> ^obj <box-b> ^room <Room4>)
    (in-room <inr5> ^obj <box-a> ^room <Room4>)
    (in-room <inr6> ^obj <robot> ^room <Room5>)
    ;; goal statement
    (goal <g> ^desired-state <d>)
    (desired <d> ^goal-conjuncts <gc> ^better lower
     )
    (goal-conjuncts <gc> ^door-closed <gnt> + &,
    ^next-to <gnt3> + &, <gnt2> + &,
    )
    (door-closed <gnt> ^door <Room7Room4> ^args one ^goal-name <gname1>)
    (next-to <gnt2> ^box <box-a> + &, <box-c> + &, ^args two
     ^goal-name <gname2>)
    (next-to <gnt3> ^box <box-c> + &, <box-d> + &, ^args two
     ^goal-name <gname3>)
;;; Task 13
    ;; initial state augmentations
    (next-to <nt1> ^obj1 <box-d> ^obj2 <box-c>)
    (next-to <nt2> ^obj1 <box-c> ^obj2 <box-d>)
    (next-to <nt3> ^obj1 <box-c> ^obj2 <Room6Room5>)
    (door-status <ds1> ^door <Room4Room3> ^status closed)
    (door-status <ds2> ^door <Room7Room4> ^status open)
    (door-status <ds3> ^door <Room2Room4> ^status closed)
    (door-status <ds4> ^door <Room5Room4> ^status open)
    (door-status <ds5> ^door <Room5Room7> ^status closed)
    (door-status <ds6> ^door <Room2Room1> ^status closed)
    (door-status <ds7> ^door <Room5Room2> ^status open)
    (door-status <ds8> ^door <Room6Room5> ^status open)
    (in-room <inr1> ^obj <box-e> ^room <Room1>)
    (in-room <inr2> ^obj <box-d> ^room <Room6>)
    (in-room <inr3> ^obj <box-c> ^room <Room6>)
    (in-room <inr4> ^obj <box-b> ^room <Room3>)
    (in-room <inr5> ^obj <box-a> ^room <Room4>)
    (in-room <inr6> ^obj <robot> ^room <Room7>)
    ;; goal statement
    (goal <g> ^desired-state <d>)
    (desired <d> ^goal-conjuncts <gc> ^better lower
     )
    (goal-conjuncts <gc> ^in-room <gnt> + &, ^in-room <gnt2> + &,
    ^next-to <gnt3>
    ^door-closed <gnt4>
    )
    (in-room <gnt> ^box <box-e> ^room <Room7> ^args two ^goal-name <gname1>)
    (in-room <gnt2> ^box <box-a> ^room <Room7> ^args two ^goal-name <gname2>)
```

```
(door-closed <gnt4> ^door <Room6Room5> ^args one ^goal-name <gname4>)
(next-to <gnt3> ^box <box-b> + &, <box-c> + &, ^args two
 ^goal-name <gname3>)
```

Appendix E

Experimental Results: Detailed Information

This appendix gives the detailed numeric information used to construct the figures of Chapter 6. In the figures below, the tasks corresponding to each task number are given in Appendices B, D, and C.

E.1 Solution Quality

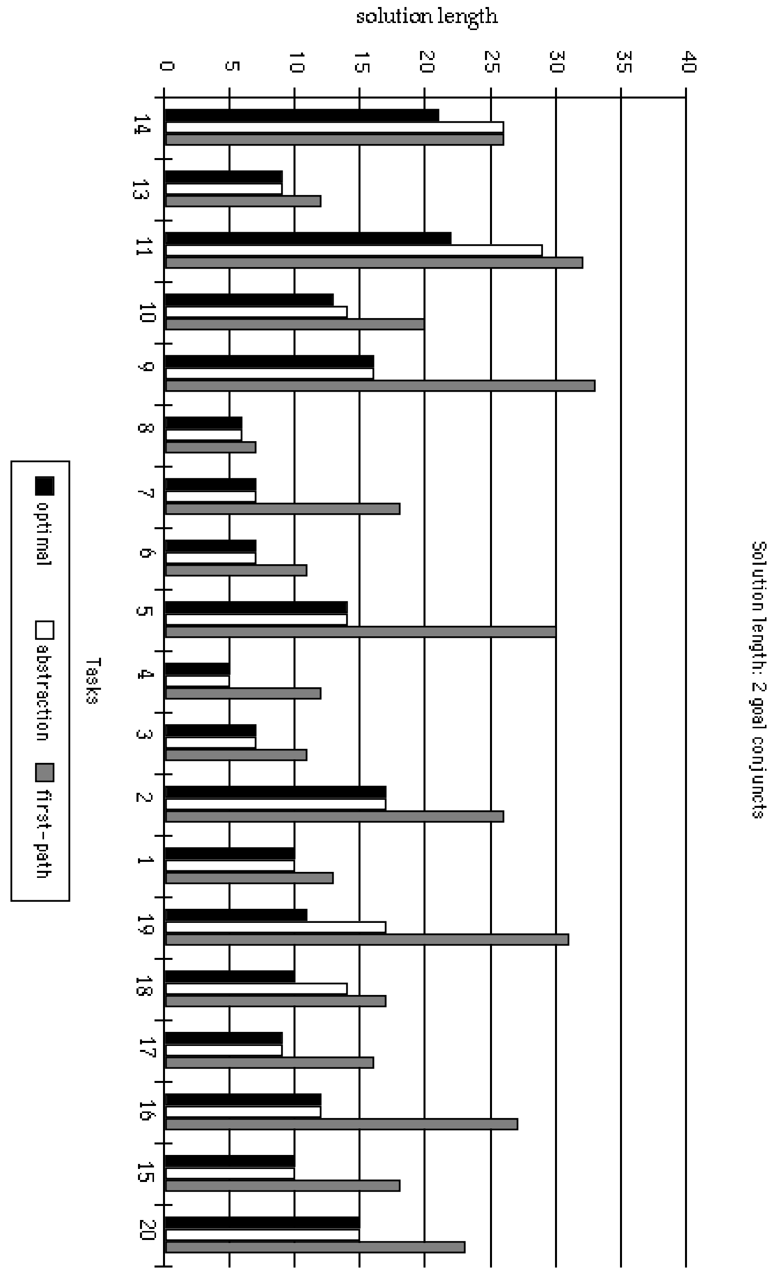


Figure E.1: Robot Domain: Solution length for 2-goal-conjunct tasks, original room layout.

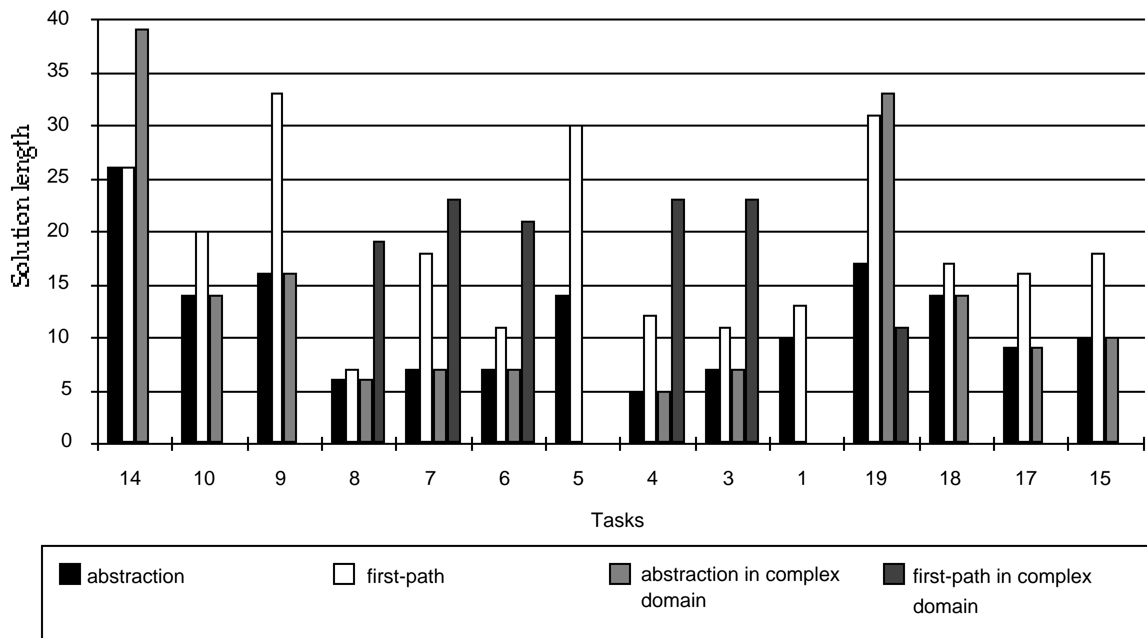


Figure E.2: Robot Domain: Solution length for 2-goal-conjunct tasks in complex room layout, as compared with the corresponding tasks in the original room layout. Optimal solutions remain the same. Missing entries indicate which tasks were not able to finish.

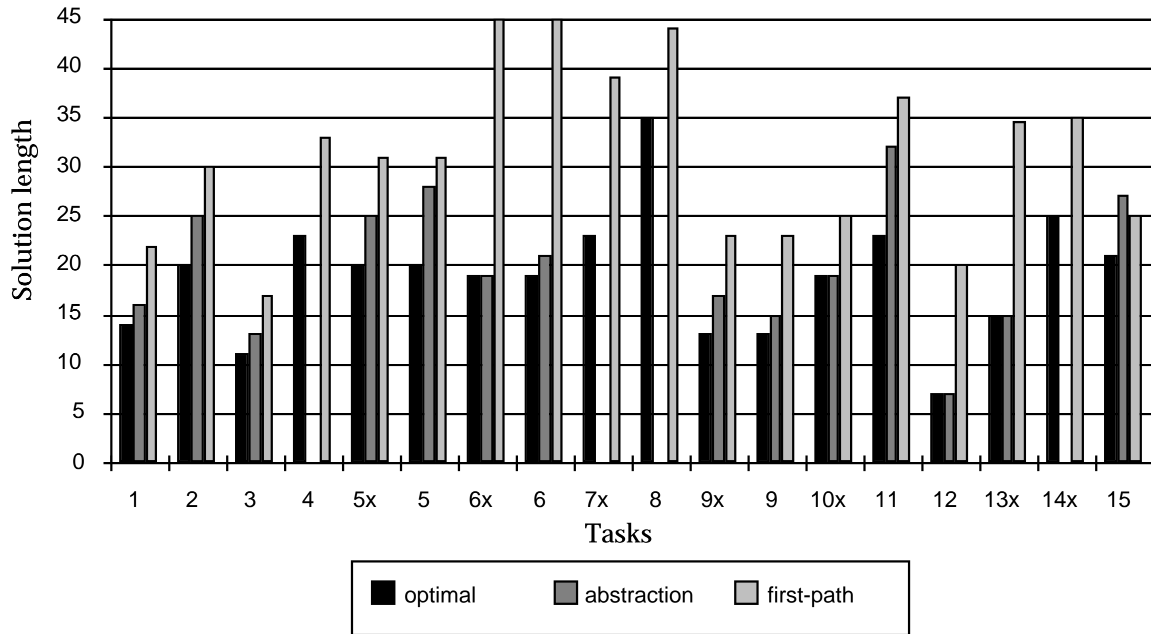


Figure E.3: Robot Domain: Solution length for 3-goal-conjunct tasks, original room layout. Missing entries indicate which tasks did not finish. Only 1 task with best-path search was able to finish.

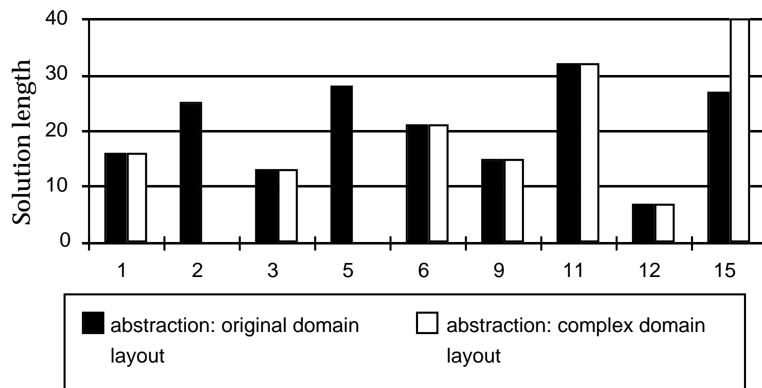


Figure E.4: Robot Domain: Solution length produced by using abstraction for 3-goal-conjunct tasks in complex room layout, as compared with the corresponding tasks in the original room layout. Missing entries indicate which tasks did not finish. Optimal solutions remain the same.

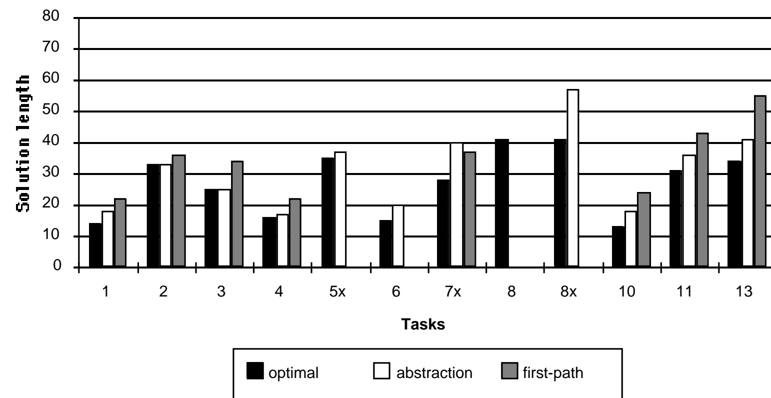


Figure E.5: Robot Domain: Solution length for 4-goal-conjunct tasks, original room layout. Missing entries indicate which tasks did not finish. Best-path search was intractable.

E.2 Problem-solving Efficiency

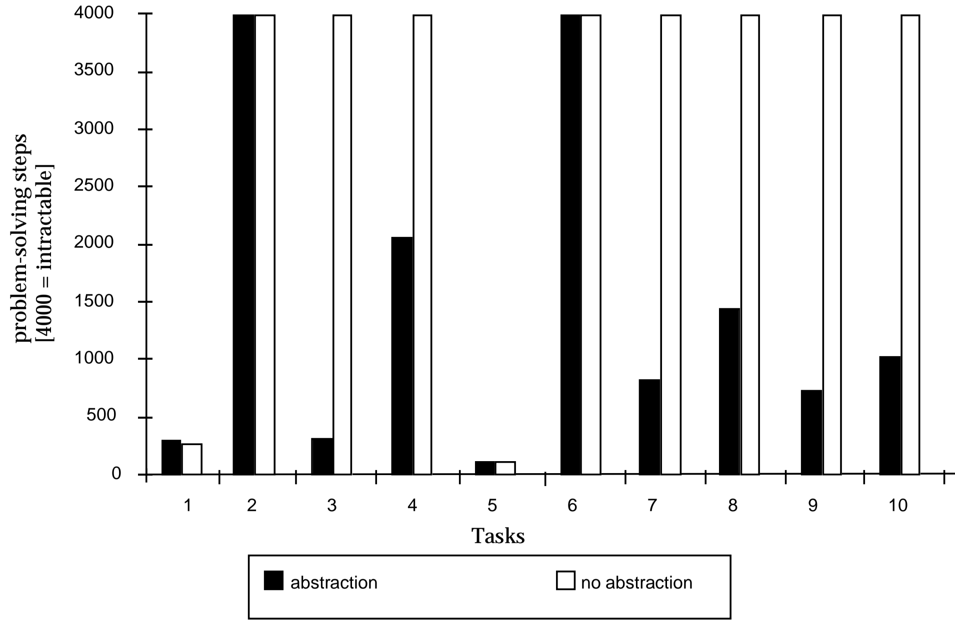


Figure E.6: Eight-puzzle: problem-solving steps.

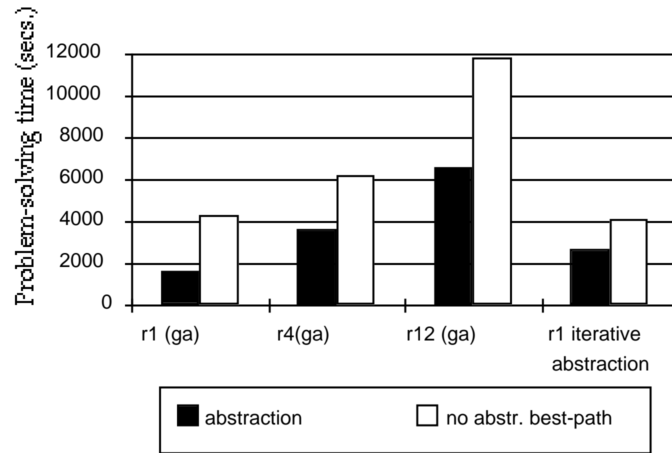


Figure E.7: Eight-puzzle: Total problem-solving time for a subset of the EP tasks. Tasks marked with '(ga)' used a modified version of goal achievement iteration.

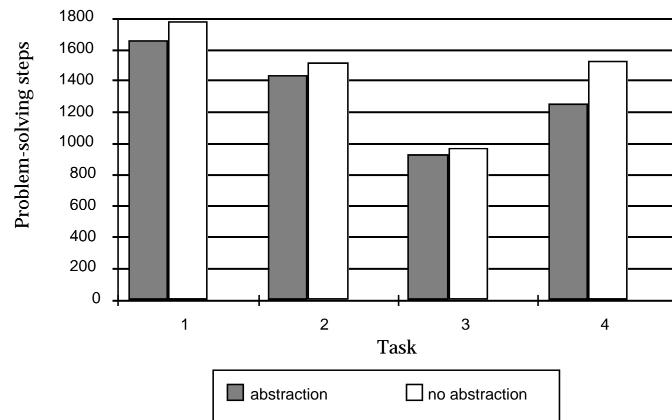


Figure E.8: Tower of Hanoi, 4-disk tasks: problem-solving steps.

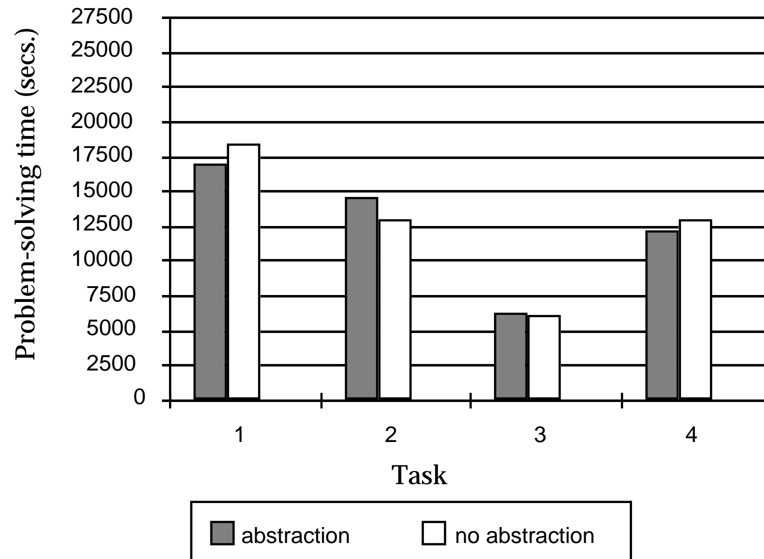


Figure E.9: Tower of Hanoi, 4-disk tasks: problem-solving time in seconds.

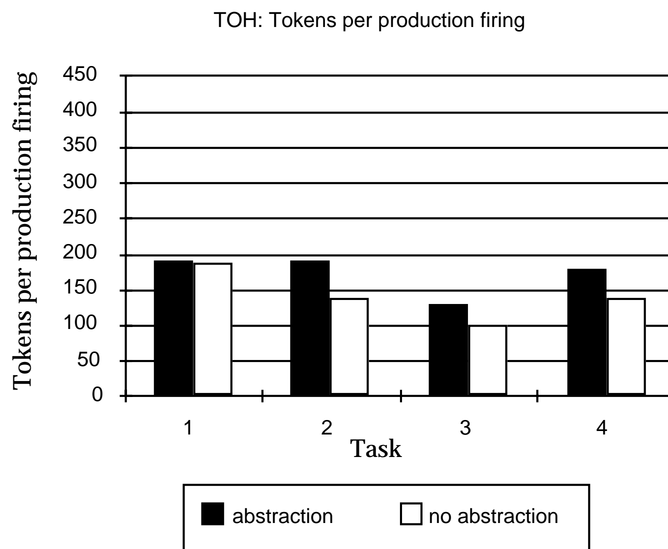


Figure E.10: Tower of Hanoi: number of tokens per production firing.

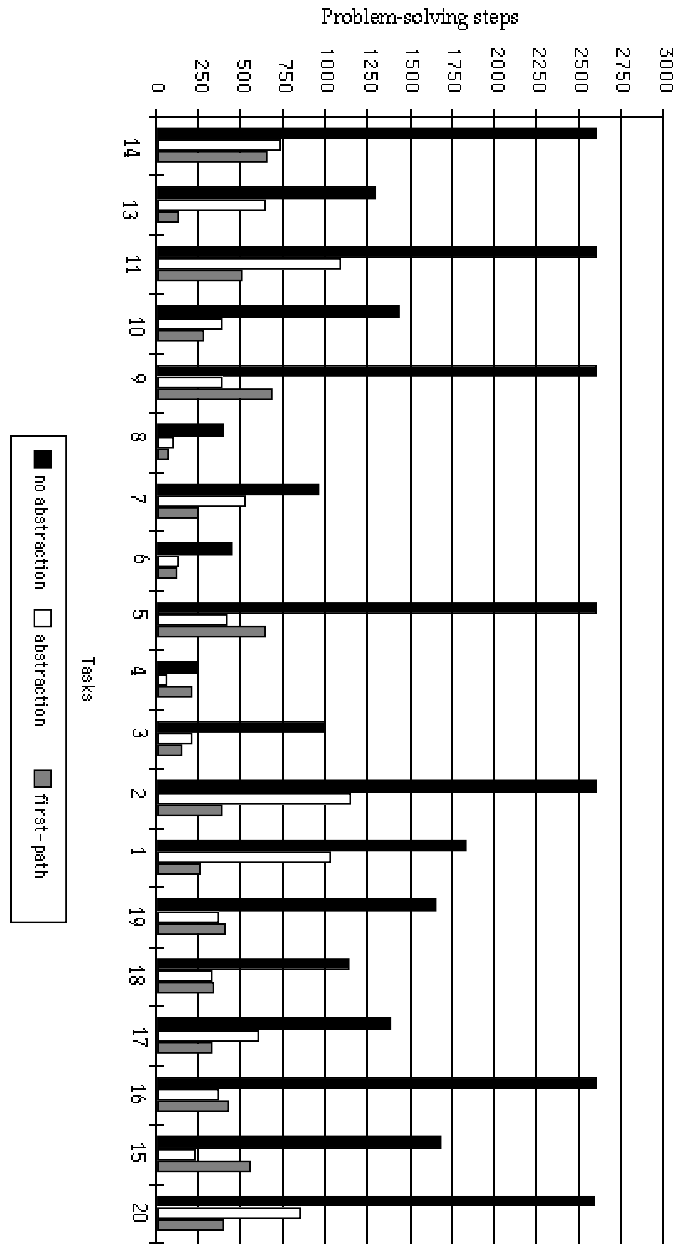


Figure E.11: Robot Domain: problem-solving steps for 2-goal-conjunct tasks. Tasks which took more than 2600 steps were cut off.

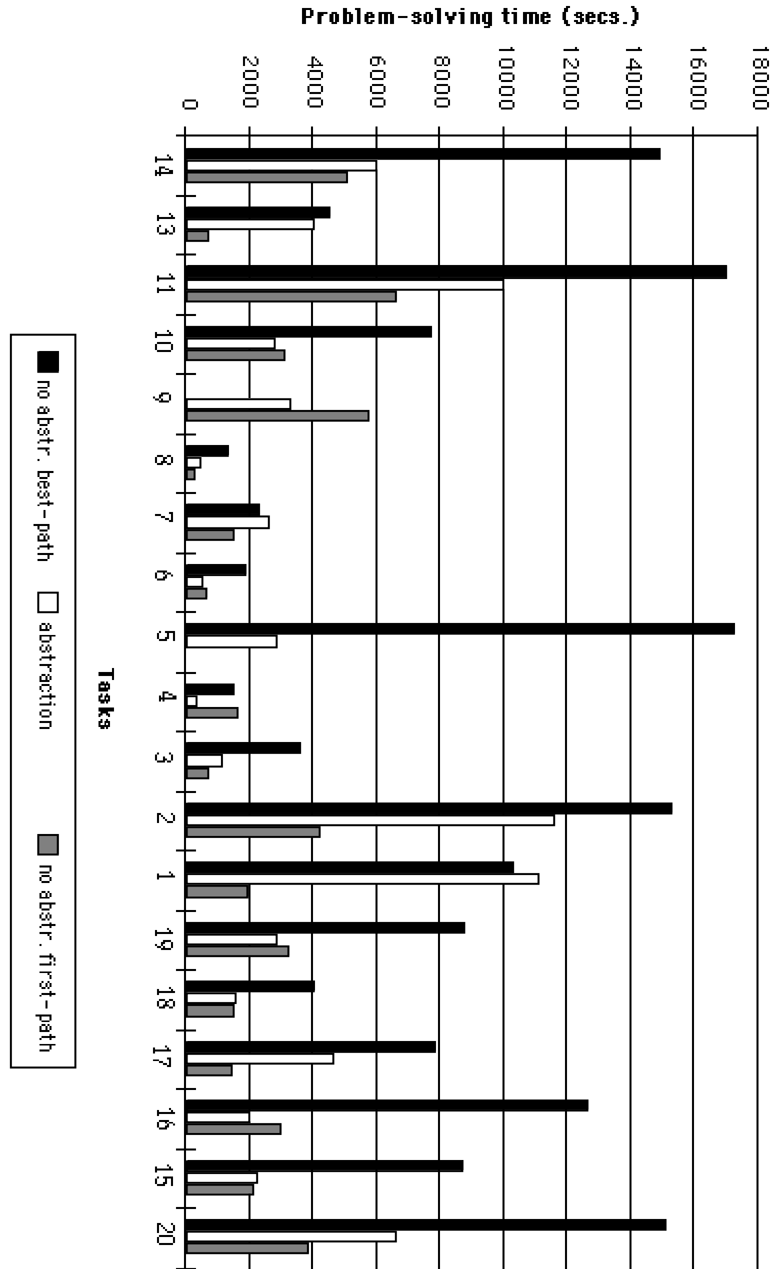


Figure E.12: Robot Domain, 2 goal conjuncts: problem-solving time (secs.) Entries are missing for one first-path and one non-abstract best-path search because of recording errors.

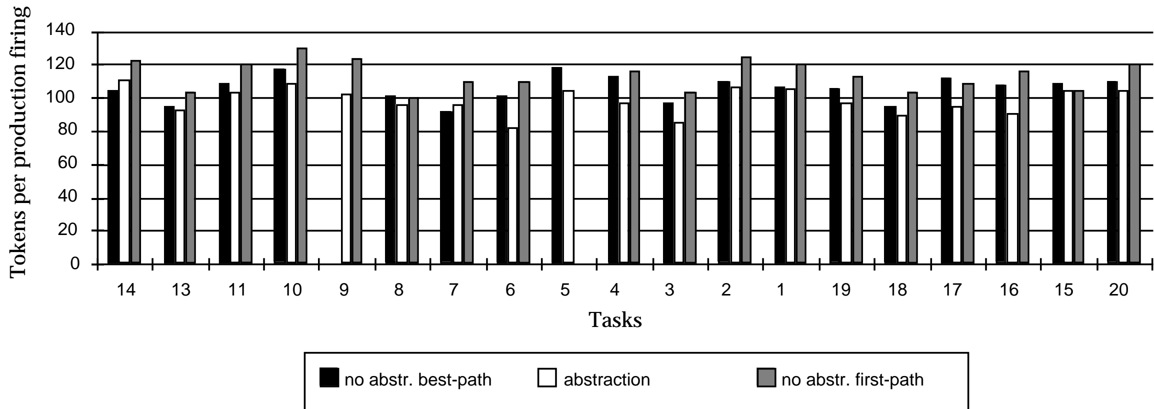


Figure E.13: Robot Domain, 2 goal conjuncts: tokens per production firing.

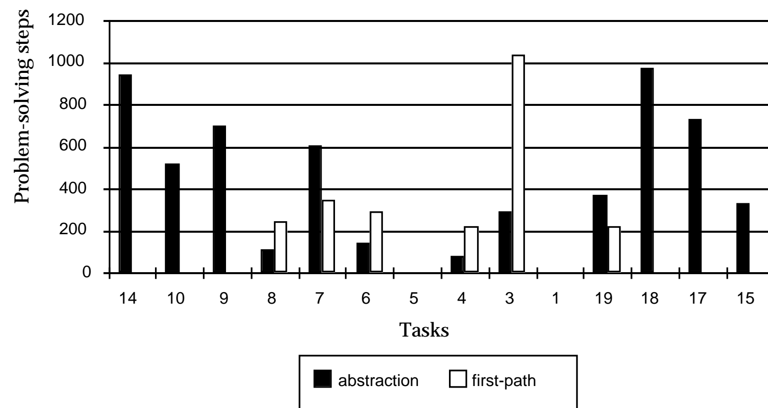


Figure E.14: Robot Domain: problem-solving steps for 2-goal-conjunct tasks in complex room layout. Blank entries indicate tasks which did not complete.

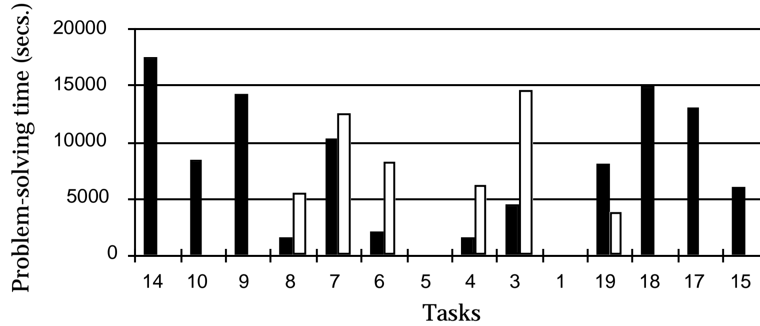


Figure E.15: Robot Domain, 2 goal conjuncts: problem-solving time in complex room layout. Blank entries indicate tasks which did not complete.

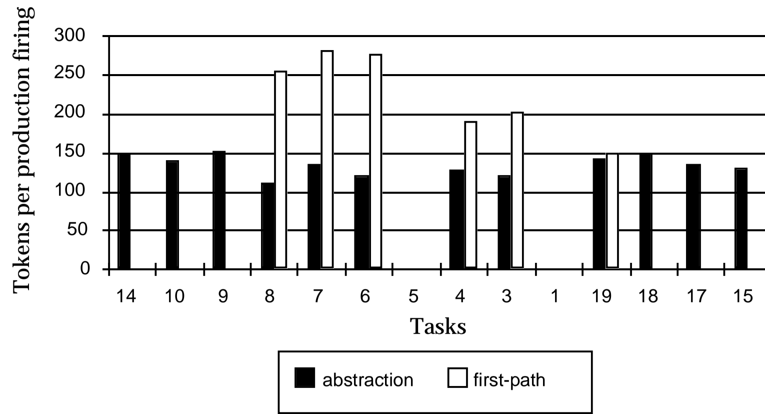


Figure E.16: Robot Domain, 2 goal conjuncts: tokens per production firing in complex room layout. Blank entries indicate tasks which did not complete.

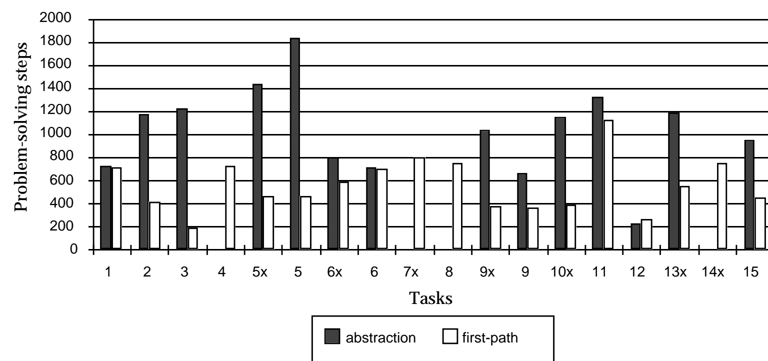


Figure E.17: Robot Domain, 3 goal conjuncts: problem-solving steps; original room layout. Non-abstract best-path search was not tractable. Missing entries indicate tasks which did not complete.

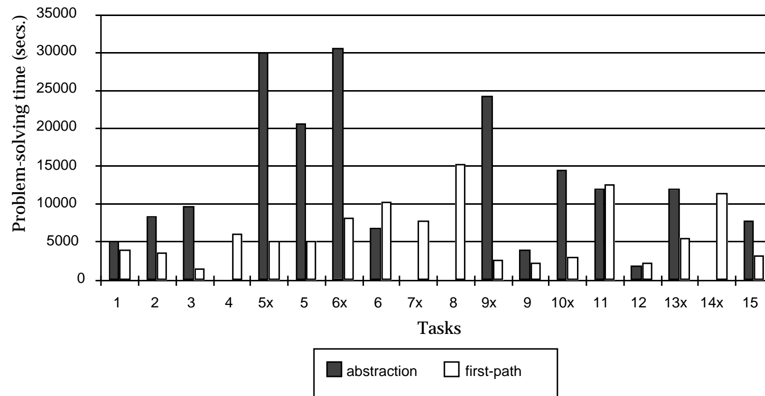


Figure E.18: Robot Domain, 3 goal conjuncts: problem-solving time (secs.); original room layout. Non-abstract best-path search was not tractable. Missing entries indicate tasks which did not complete.

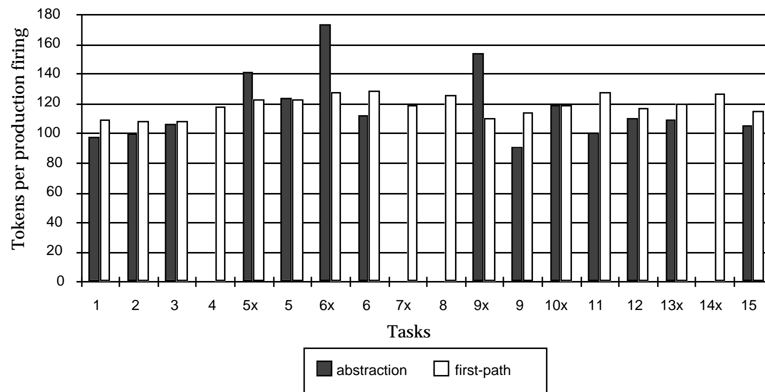


Figure E.19: Robot Domain, 3 goal conjuncts: tokens per production firing; original room layout.

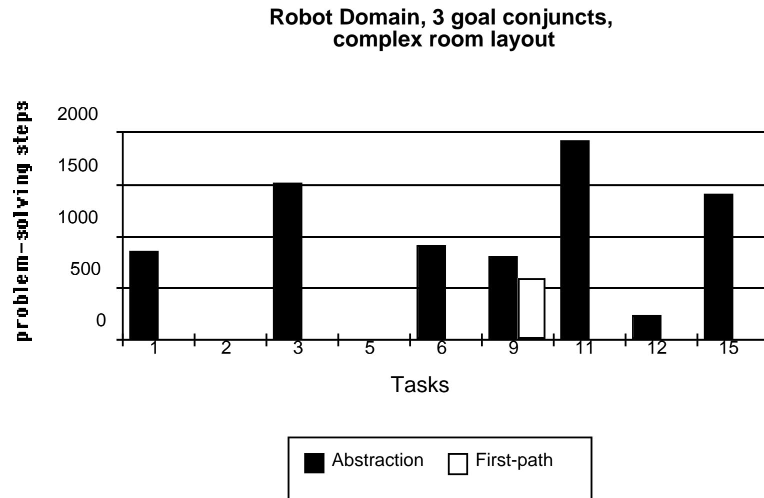


Figure E.20: Robot Domain, 3 goal conjuncts: problem-solving steps; complex room layout. Non-abstract best-path search was not tractable. Missing entries indicate tasks which did not complete.

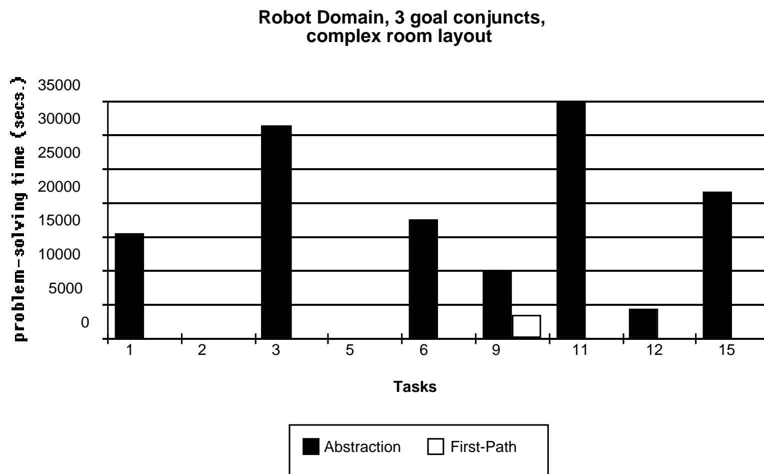


Figure E.21: Robot Domain, 3 goal conjuncts: problem-solving time; complex room layout. Non-abstract best-path search was not tractable. Missing entries indicate tasks which did not complete.

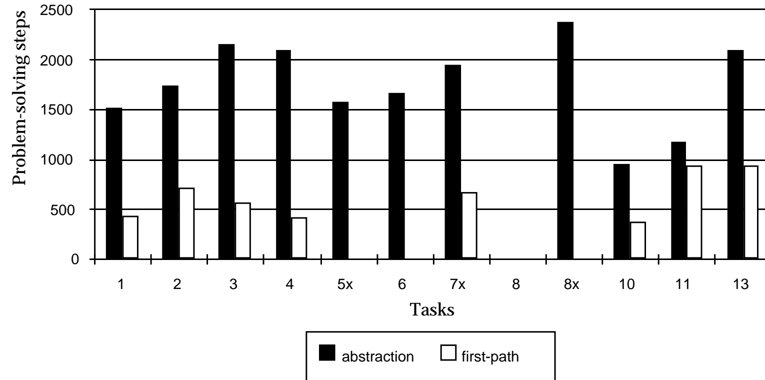


Figure E.22: Robot Domain: problem-solving steps for 4-goal-conjunct tasks; original room layout. Non-abstract best-path search was not tractable. Missing entries indicate tasks which did not complete.

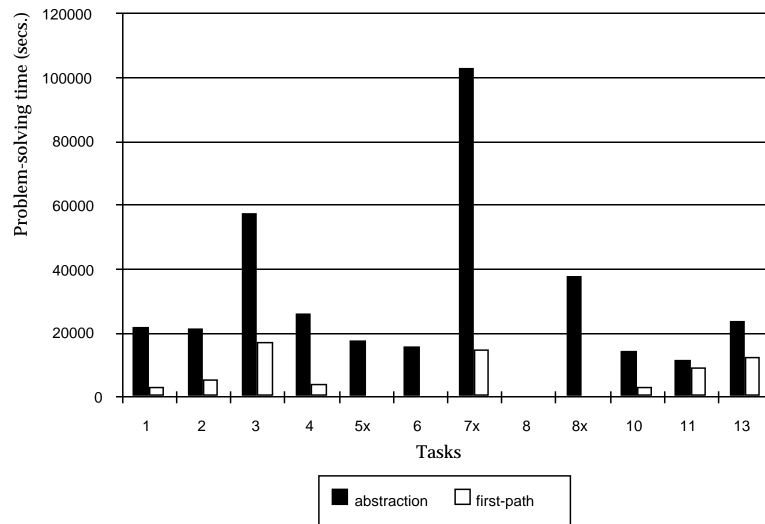


Figure E.23: Robot Domain, 4 goal conjuncts: problem-solving time (secs.); original room layout. Non-abstract best-path search was not tractable. Missing entries indicate tasks which did not complete. Time for “7x” task is probably unreliable, as suggested by tokens changes graph of Figure E.24.

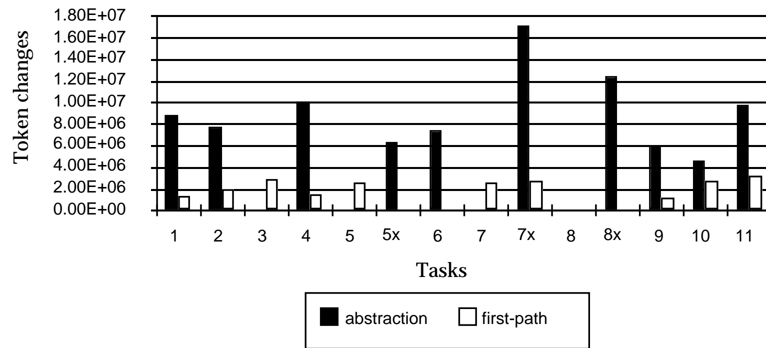


Figure E.24: Robot Domain, 4 goal conjuncts: total token changes, original room layout. Non-abstract best-path search was not tractable. Missing entries indicate tasks which did not complete.

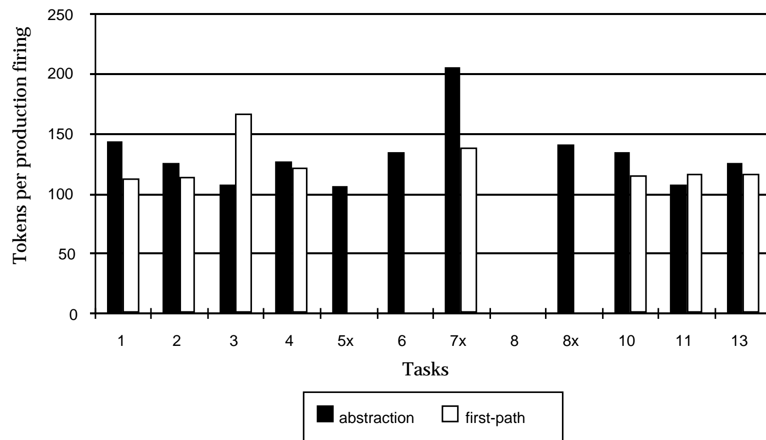


Figure E.25: Robot Domain, 4 goal conjuncts: tokens per production firing.

E.3 Learning

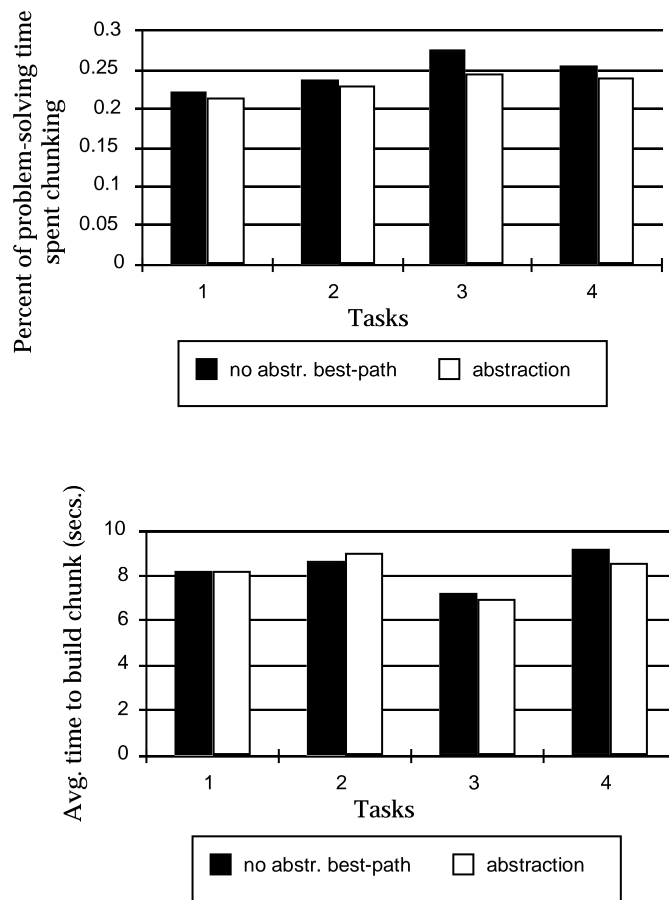


Figure E.26: Tower of Hanoi: chunking expense.

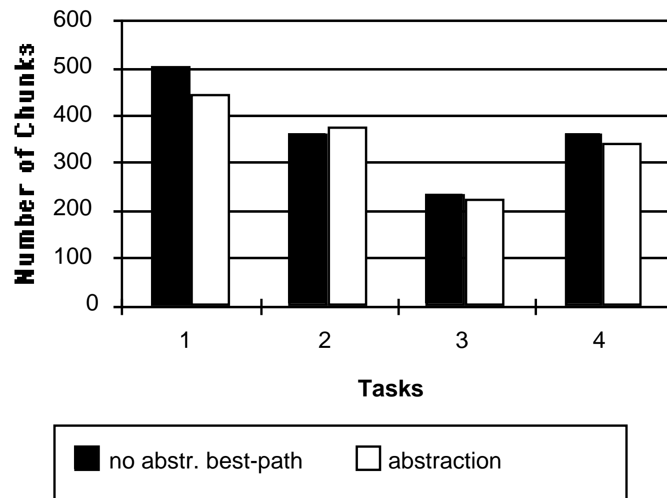


Figure E.27: Tower of Hanoi: Number of chunks.

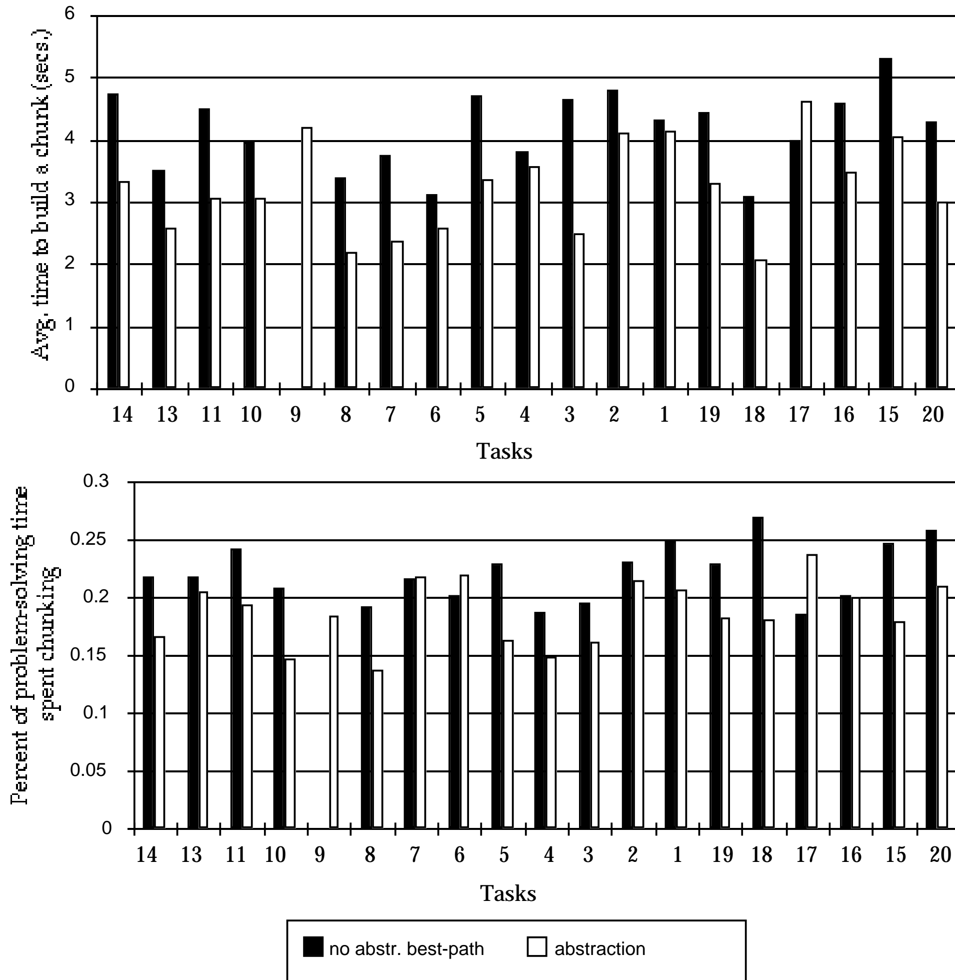


Figure E.28: Robot Domain, 2 goal conjuncts: chunking expense. Missing data is due to errors in recording data.

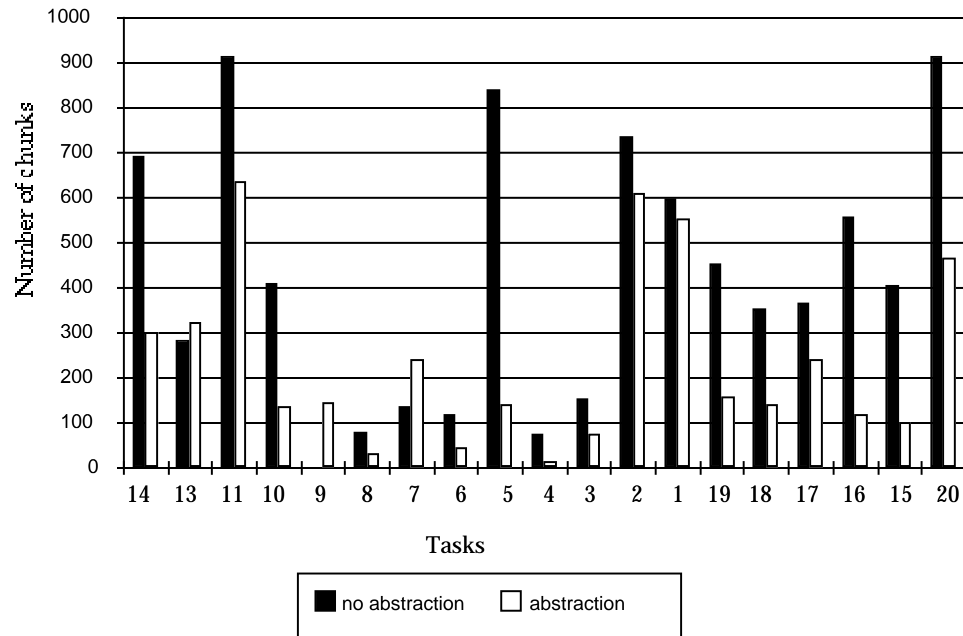


Figure E.29: Robot Domain, 2 goal conjuncts: Number of chunks. Missing data is due to errors in recording.

E.3.1 The Extended-Plan-Use Method Increment

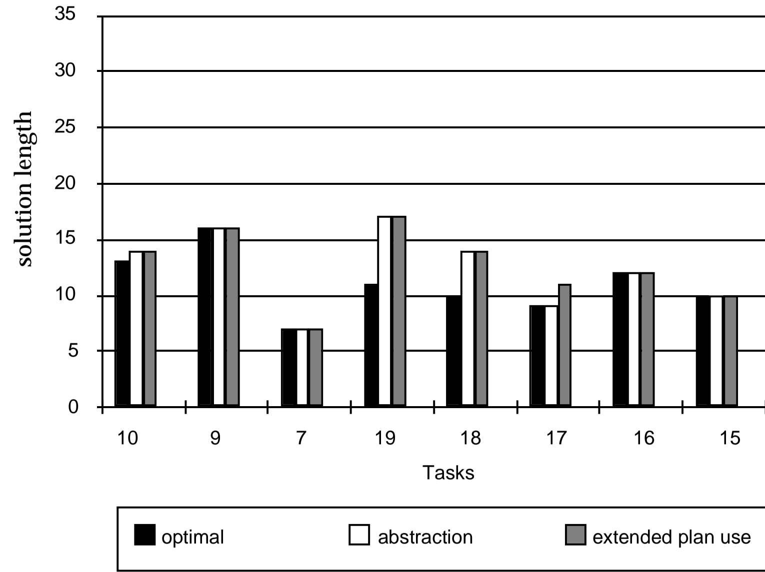


Figure E.30: Robot Domain, original layout, 2 goal conjuncts, extended-plan-use method increment: solution length.

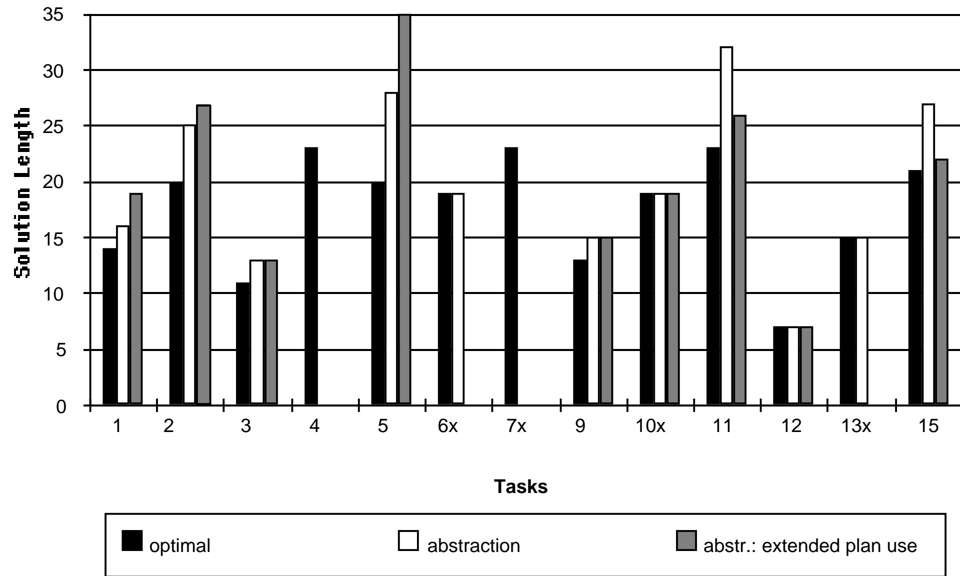


Figure E.31: Robot Domain, original layout, 3 goal conjuncts, extended-plan-use method increment: solution length. Blank entries indicate tasks which did not finish.

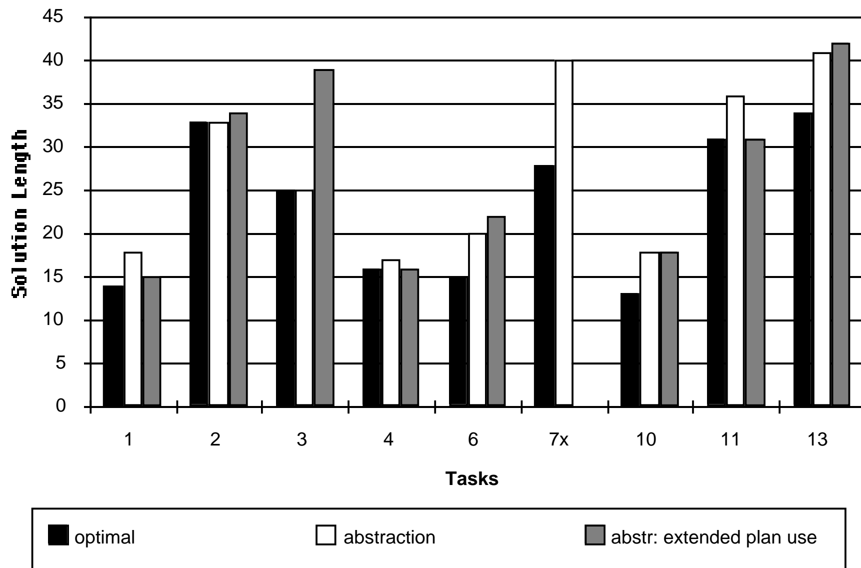


Figure E.32: Robot Domain, original layout, 4 goal conjuncts, extended-plan-use method increment: solution length. Blank entries indicate tasks which did not finish.

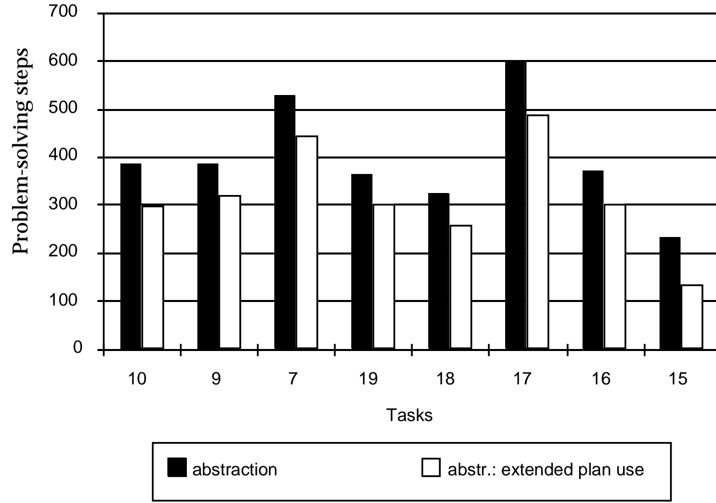


Figure E.33: Robot Domain, original layout, 2 goal conjuncts, extended-plan-use method increment: problem-solving steps.

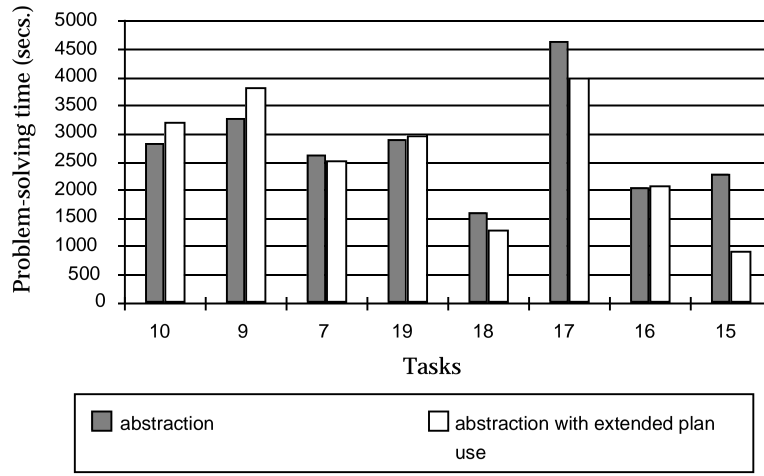


Figure E.34: Robot Domain, original layout, 2 goal conjuncts, extended-plan-use method increment: problem-solving time.

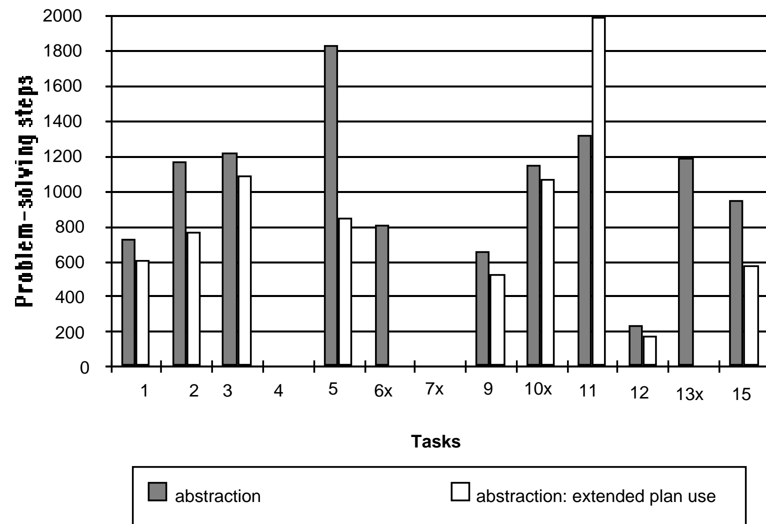


Figure E.35: Robot Domain, original layout, 3 goal conjuncts, extended-plan-use method increment: problem-solving steps. Blank entries indicate tasks which did not finish.

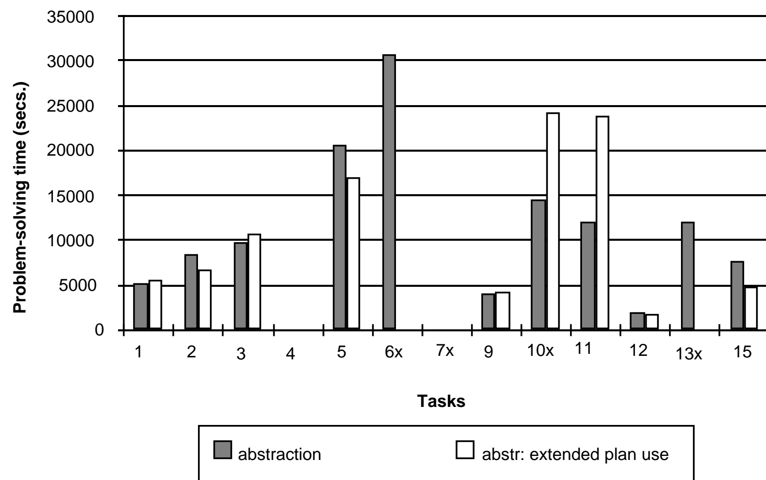


Figure E.36: Robot Domain, original layout, 3 goal conjuncts, extended-plan-use method increment: problem-solving time. Blank entries indicate tasks which did not finish.

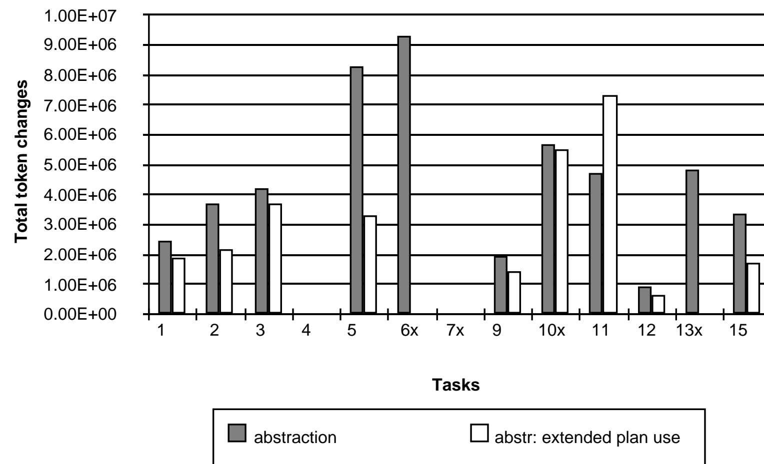


Figure E.37: Robot Domain, original layout, 3 goal conjuncts, extended-plan-use method increment: token changes.

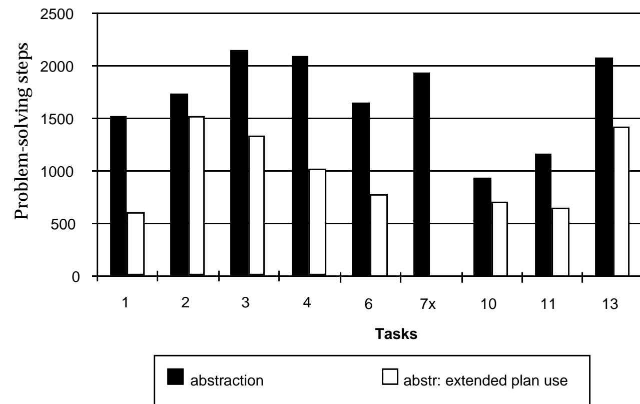


Figure E.38: Robot Domain, original layout, 4 goal conjuncts, extended-plan-use method increment: problem-solving steps. Blank entries indicate tasks which did not finish.

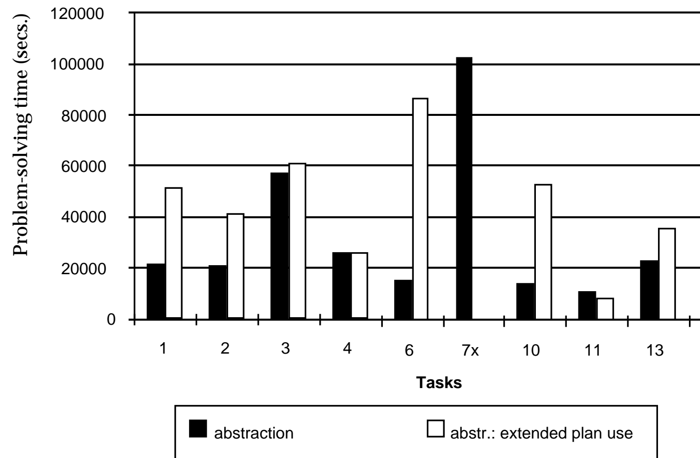


Figure E.39: Robot Domain, original layout, 4 goal conjuncts, extended-plan-use method increment: problem-solving time. Blank entries indicate tasks which did not finish.

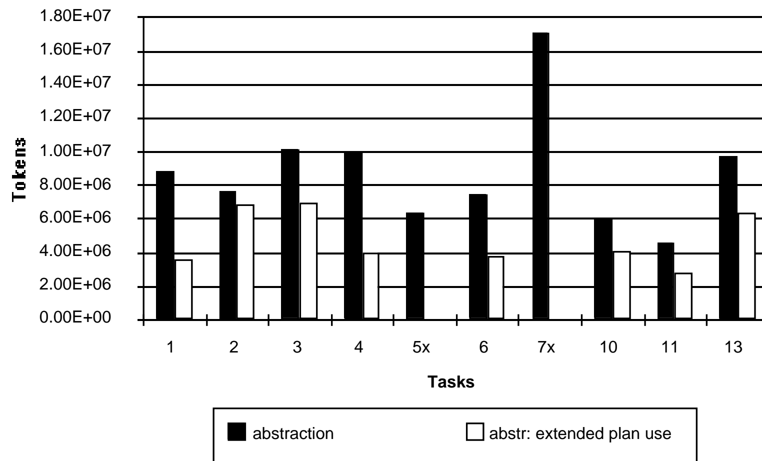


Figure E.40: Robot Domain, original layout, 4 goal conjuncts, extended-plan-use method increment: token changes.

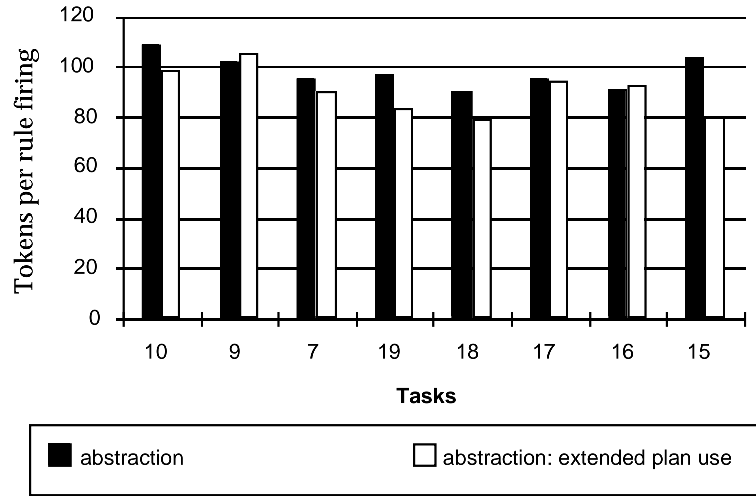


Figure E.41: Robot Domain, original layout, 2 goal conjuncts, extended-plan-use method increment: tokens per production firing.

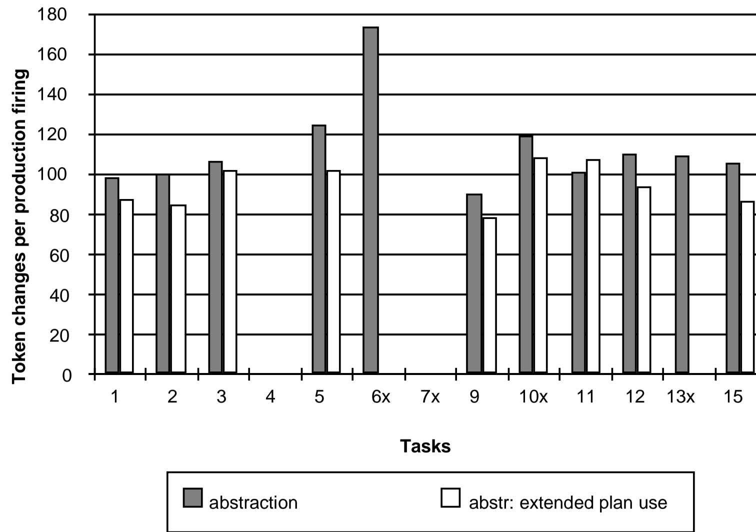


Figure E.42: Robot Domain, original layout, 3 goal conjuncts, extended-plan-use method increment: tokens per production firing.

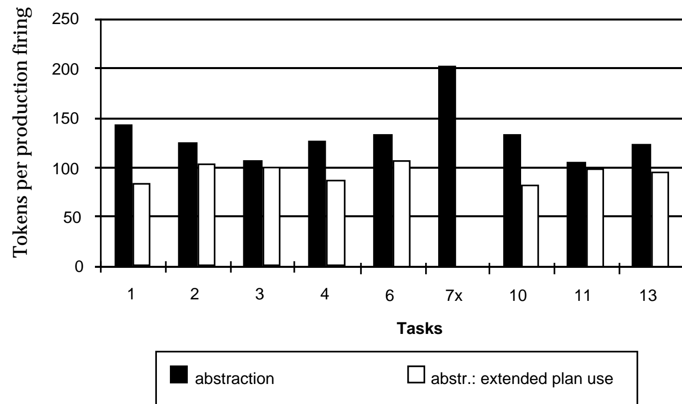


Figure E.43: Robot Domain, original layout, 4 goal conjuncts, extended-plan-use method increment: tokens per production firing.

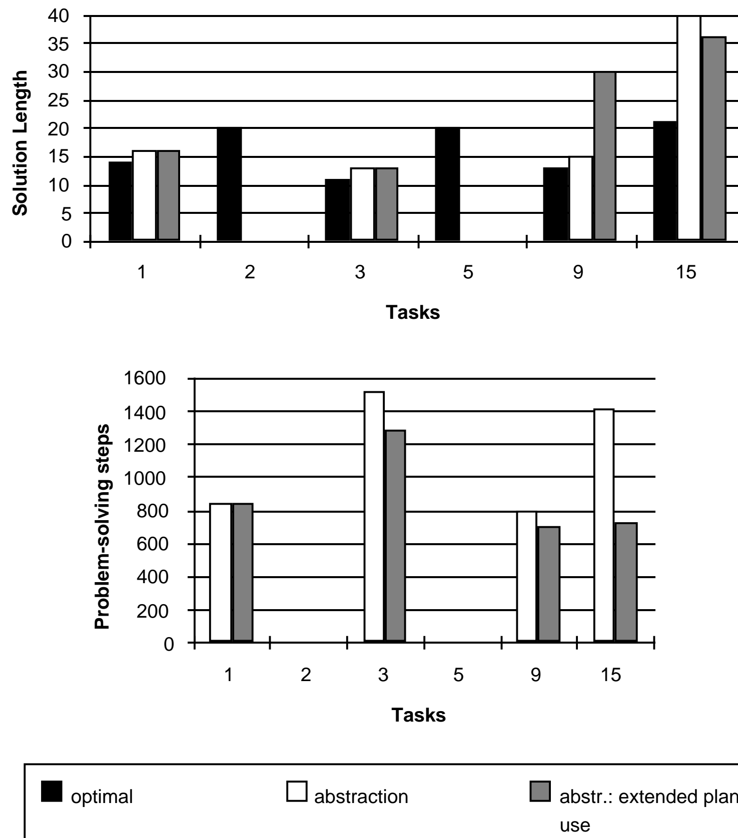


Figure E.44: Robot Domain, 3 goal conjuncts: problem-solving with the extended plan use method in the complex room layout.

E.3.2 Plan Transfer

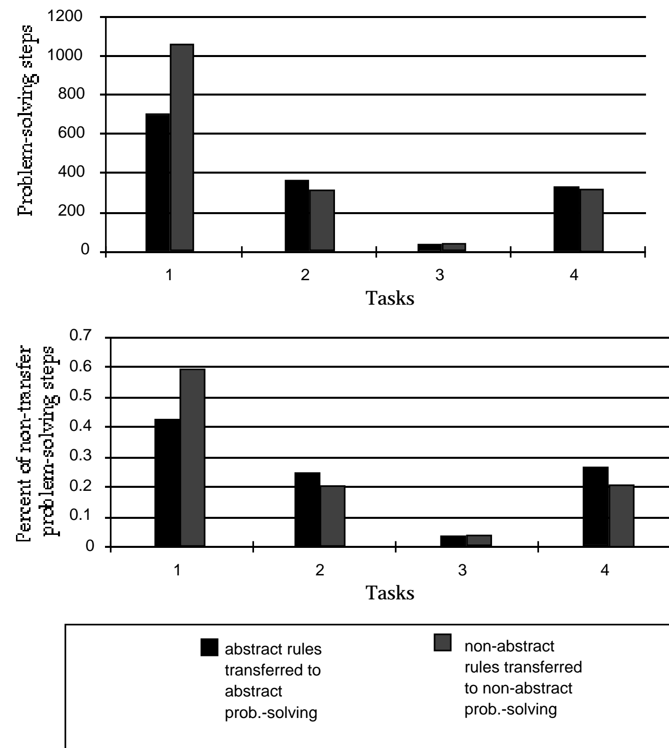


Figure E.45: Effect of transfer of rules on number of problem-solving steps in the Tower of Hanoi.

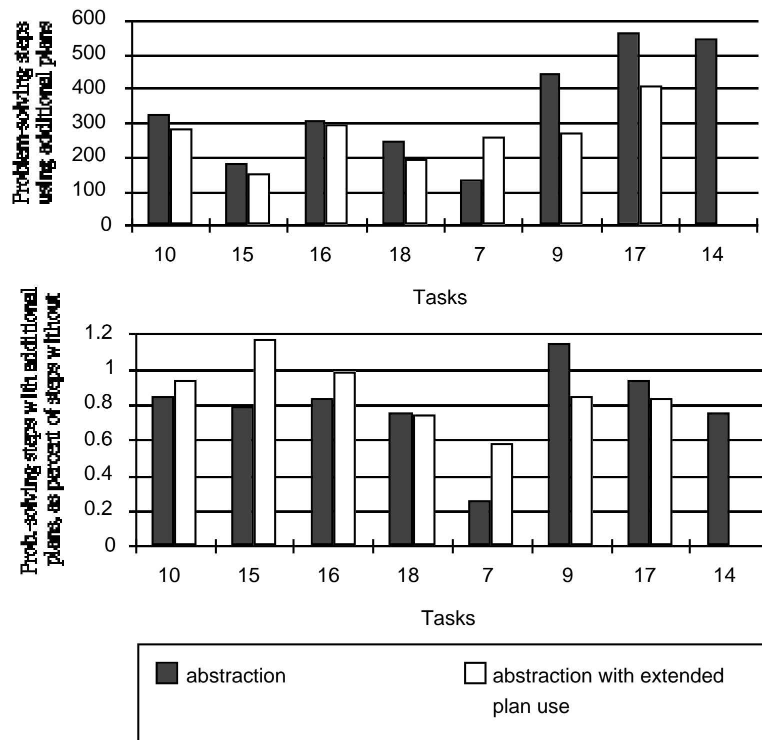


Figure E.46: Effect of transfer of rules on number of problem-solving steps in the Robot Domain.

Bibliography

[Amarel, 1967] S. Amarel. An approach to heuristic problem-solving and theorem-proving in the propositional calculus. In J. Hart and S. Takasu, editors, *Systems and Computer Science*. University of Toronto Press, Toronto, 1967.

[Anderson and Farley, 1988] J. Anderson and A. Farley. Plan abstraction based on operator generalization. In *Proceedings of the Seventh National Conference on Artifical Intelligence*, 1988.

[Anderson and Fickas, 1990] J. Anderson and S. Fickas. Approximations and abstractions in design and analysis of software spcifications. In *Proceedings of the AAAI Workshop on Automatic Generation of Approximations and Abstractions*, 1990.

[Bacchus and Yang, 1992] F. Bacchus and Q. Yang. The expected value of hierarchical problem-solving. In *Proceedings of the Tenth National Conference on Artifical Intelligence*, pages 369–374, 1992.

[Benjamin, 1989] D. P. Benjamin. Learning problem-solving abstractions via enablement. In *AAAI Spring Symposium Series: Planning and Search*, 1989.

[Bennett, 1990a] S. Bennett. An overview of approximation as employed by the grasper system. In *Proceedings of the AAAI Workshop on Automatic Generation of Approximations and Abstractions*, 1990.

[Bennett, 1990b] S. Bennett. Planning to address uncertainty: An incremental approach employing learning through experience. In *Proceedings of the DARPA workshop on Innovative Approaches to Planning, Scheduling, and Control*, 1990.

[Bresina, 1988] J. Bresina. REAPPR – an expert system shell for planning. Technical Report LCSR-TR-119, Laboratory for Computer Science Research, Rutgers University, 1988.

[Campbell, 1988] M Campbell. *Chunking as an Abstraction Mechanism*. PhD thesis, Carnegie-Mellon University, 1988.

[Chapman, 1987] D. Chapman. Planning for conjunctive goals. *Artificial Intelligence*, 32:333–378, 1987.

[Chase *et al.*, 1989] M. P. Chase, M. Zweben, R. Piazza, J. Burger, P. Maglio, and H. Hirsh. Approximating learned search control knowledge. In *Proceedings of the Sixth International Workshop on Machine Learning*, pages 218–220. Morgan Kaufmann, 1989.

[Chien, 1989] S. A. Chien. Failure-guided search in planning. In *Proceedings of the AAAI Symposium on Planning and Search*, 1989.

[Christensen, 1990] J. Christensen. A hierarchical planner that generates its own hierarchies. In *Proceedings of AAAI-90*, 1990. In Press.

[Christensen, 1991] J. Christensen. *Automatic Abstraction in Planning*. PhD thesis, Stanford University, 1991.

[Currie and Tate, 1988] K. Currie and A. Tate. O-plan: the open planning architecture, December 1988. SERC grant report.

[Danyluk, 1989] A. Danyluk. Finding new rules for incomplete theories: explicit biases for induction with contextual information. In *Proceedings of the Sixth International Workshop on Machine Learning*, pages 34–36. Morgan Kaufmann, 1989.

[Dietterich, 1980] T. Dietterich. Applying general induction methods to the card game eleusis. In *AAAI-1*, pages 218–220, 1980.

[Dietterich, 1986] T. Dietterich. Learning at the knowledge level. *Machine Learning*, 1:287–316, 1986.

[Drummond *et al.*, 1992] M. Drummond, R. Levinson, J. Bresina, and K. Swanson. Reaction-first search: Incremental planning with guaranteed performance improvement. Technical report, NASA Ames Research Center, 1992.

[Elkan, 1990a] C. Elkan. Incremental, approximate planning. In *Proceedings of the Eight National Conference on Artificial Intelligence*, pages 145–150, 1990.

[Elkan, 1990b] C. Elkan. Incremental, approximate planning: abductive default reasoning. In *Proceedings of the AAAI Spring Symposium Series on Planning in Uncertain, Unpredictable, or Changing Environments*, 1990.

[Ellman, 1988] T. Ellman. Approximate theory formation: An explanation-based approach. In *Proceedings of the Seventh National Conference on Artifical Intelligence*, 1988.

[Ellman, 1990] T. Ellman. Mechanical generation of heuristics through approximation of intractable theories. In *Proceedings of the AAAI Workshop on Automatic Generation of Approximations and Abstractions*, 1990.

[Etzioni and Minton, 1992] O. Etzioni and S. Minton. Why EBL produces overly-specific knowledge: A critique of the PRODIGY approaches. In *AAAI-92 Workshop on Approximation and Abstraction of Computational Theories*, pages 61–67, 1992.

[Etzioni, 1991] O. Etzioni. STATIC: A problem-space compiler for PRODIGY. In *Proceedings of the Ninth National Conference on Artifical Intelligence*, pages 533–540, 1991.

[Fawcett, 1989] T. Fawcett. Learning from plausible explanations. In *Proceedings of the Sixth International Workshop on Machine Learning*, pages 37–39. Morgan Kaufmann, 1989.

[Fikes and Nilsson, 1971] R. E. Fikes and N. J. Nilsson. Strips: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2:189–208, 1971.

[Gaschnig, 1979] J. Gaschnig. A problem simililarity approach to devising heuristics. In *Proceedings of IJCAI-79*, pages 301–307, 1979.

[Gervasio and DeJong, 1989] Gervasio and DeJong. Explanation-based learning of reactive operators. In *Proceedings of the Sixth International Workshop on Machine Learning*. Morgan Kaufmann, 1989.

[Ginsberg, 1991] M. Ginsberg. The computational value of nonmontonic reasoning. In *Proceedings of the Conference on Knowledge Representation and Reasoning*, 1991.

[Giunchiglia and Walsh, 1990a] F. Giunchiglia and T. Walsh. Abstract theorem proving: Mapping back. Technical Report IRST-TR #8911-16, Istituto per la Ricera Scientifica e tecnologica, 1990.

[Giunchiglia and Walsh, 1990b] F. Giunchiglia and T. Walsh. Using abstraction. Technical Report IRST-TR #9010-08, Istituto per la Ricera Scientifica e tecnologica, 1990.

[Guvenir and Ernst, 1990] H. A. Guvenir and G.W. Ernst. Learning problem solving strategies using refinement and macro generation. *Artificial Intelligence*, 44:209–243, 1990.

[Hansson and Mayer, 1989] O. Hansson and A. Mayer. Subgoal generation from problem relaxation. In *Proceedings of the AAAI Symposium on Planning and Search*, 1989.

[Hendler *et al.*, 1990] J. Hendler, A. Tate, and M. Drummond. AI planning: Systems and techniques. *AI Magazine*, summer:61–77, 1990.

[Iba, 1989] G. Iba. A heuristic approach to the discoery of macro-operators. *Machine Learning*, 3:287–317, 1989.

[Joslin and Roach, 1989] D. Joslin and J. Roach. A theoretical analysis of conjunctive-goal problems. *Artificial Intelligence*, 41:97–106, 1989.

[Keller, 1990] R. Keller. Learning approximate concept descriptions. In *Proceedings of the AAAI Workshop on Automatic Generation of Approximations and Abstractions*, 1990.

[Kibler, 1985] D. Kibler. Generation of heuristics by transforming the problem representation. Technical Report TR-85-20, ICS, 1985.

[Knoblock *et al.*, 1991] C. Knoblock, S. Minton, and O. Etzioni. Integrating abstraction and explanation-based learning in PRODIGY. In *Proceedings of the Ninth National Conference on Artifical Intelligence*, pages 541–546, 1991.

[Knoblock, 1989] Craig A. Knoblock. Learning hierarchies of abstraction spaces. In *Proceedings of the Sixth International Workshop on Machine Learning.* Morgan Kaufmann, 1989.

[Knoblock, 1991] C. Knoblock. *Automatically Generating Abstractions for Problem-Solving.* PhD thesis, Carnegie-Mellon University, 1991.

[Knoblock, 1992] C. Knoblock. An analysis of abstrips. Technical report, ISI, 1992.

[Korf, 1985a] R. Korf. Depth-first iterative-deepening: An optimal admissable tree search. *Artificial Intelligence*, 27:97–109, 1985.

[Korf, 1985b] R. Korf. Macro-operators: A weak method for learning. *Artificial Intelligence*, 26:35–77, 1985.

[Korf, 1987] R. E. Korf. Planning as search: A quantitative approach. *Artificial Intelligence*, 33:65–88, 1987.

[Korf, 1990] R. Korf. Real-time heuristic search. *Artificial Intelligence*, 42:189–212, 1990.

[Kuipers, 1986] B. Kuipers. Qualitative simulation. *Artificial Intelligence*, 29:289–338, 1986.

[Laird *et al.*, 1986a] J. E. Laird, P. S. Rosenbloom, and A. Newell. Chunking in Soar: The anatomy of a general learning mechanism. *Machine Learning*, 1:11–46, 1986.

[Laird *et al.*, 1986b] J.E. Laird, P.S. Rosenbloom, and A. Newell. Overgeneralization during knowledge compilation in soar. In *Proceedings of the Workshop on Knowledge Compilation*, pages 46–57, September 1986.

[Laird *et al.*, 1987a] J. E. Laird, A. Newell, , and P. S. Rosenbloom. Soar: An architecture for general intelligence. *Artificial Intelligence*, 33:1–64, 1987.

[Laird *et al.*, 1987b] J.E. Laird, A. Newell, and P.S. Rosenbloom. Soar: An architecture for general intelligence. *Artificial Intelligence @*, 33(1):1–64, 1987.

[Laird *et al.*, 1989] J.E. Laird, K.R. Swedlow, E. Altmann, and C.B. Congdon. Soar 5 user's manual. Technical report, School of Computer Science, Carnegie Mellon University and Department of Electrical Engineering and Computer Science, University of Michigan, June 1989. Unpublished.

[Laird, 1988] J. Laird. Recovery from incorrect knowledge in soar. In *Proceedings of the National Conference on Artificial Intelligence*, pages 618–623, August 1988.

[Lansky, 1992] A. Lansky. Localization vs. abstraction: A comparison of two search reduction techniques. In *AAAI-92 Workshop on Approximation and Abstraction of Computational Theories*, pages 138–145, 1992.

[Lee and Rosenbloom, 1992] S. Lee and P. Rosenbloom. Creating and coordinating multiple planning methods. In *Proceedings of the Second Pacific Rim International Conference on AI*, Seoul, Korea, 1992.

[Levy *et al.*, 1992] A. Levy, Y. Iwasaki, and H. Motoda. Relevance reasoning to guide compositional modeling. In *AAAI-92 Workshop on Approximation and Abstraction of Computational Theories*, pages 146–153, 1992.

[Lewis, 1992] R.L. Lewis. Recent developments in the nl-soar garden path theory. Technical Report CMU-CS-92-141, School of Computer Science, Carnegie Mellon University, May 1992.

[Martin and Allen, 1990] N. Martin and J. Allen. Combining reactive and strategic planning through decomposition abstraction. In *Proceedings of the DARPA workshop on Innovative Approaches to Planning, Scheduling, and Control*, 1990.

[McCarthy and Hayes, 1969] J. McCarthy and P. J. Hayes. Some philosophical problems from the standpoint of artificial intelligence. In B. Meltzer and D. Michie, editors, *Machine Intelligence 4*, pages 463–502. American Elsevier, 1969.

[McCarthy, 1977] J. McCarthy. Epistemological problems of artificial intelligence. In *Proceedings IJCAI-77*, pages 1038–1044, 1977.

[Mcllester, 1988] D. Mcllester. Conspiracy numbers for min-max search. *Artificial Intelligence*, 1988.

[Minton *et al.*, 1989] S. Minton, J. Carbonell, C. Knoblock, D. Kuokka, O. Etzioni, and Y. Gil. Explanation-based learning: a problem-solving perspective. *Artificial Intelligence*, 40:63–118, 1989.

[Minton, 1990] S. Minton. Quantitative results concerning the utility of explanation-based learning. *Artificial Intelligence*, 42:363–392, 1990.

[Mitchell *et al.*, 1986] T. Mitchell, R. Keller, and S. Kedar-Cabelli. Explanation-based generalization: A unifying view. *Machine Learning*, 1, 1986.

[Morgenstern, 1990] L. Morgenstern. A logic for a non-monotonic theory of planning. In *Proceedings of the AAAI Spring Symposium Series on Planning in Uncertain, Unpredictable, or Changing Environments*, pages 100–104, 1990.

[Mostow and Prieditis, 1989] J. Mostow and A. Prieditis. Discovering admissible heuristics by abstracting and optimizing: A transformational approach. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, pages 701–707, 1989.

[Newell and Simon, 1972] A. Newell and H. A. Simon. *Human Problem Solving.* Prentice Hall, New Jersey, 1972.

[Newell, 1990] A. Newell. *Unified Theories of Cognition.* Harvard University Press, Cambridge, Massachusetts, 1990.

[Nilsson, 1980] N. Nilsson. *Principles of Artificial Intelligence.* Tioga, Palo Alto, CA, 1980.

[O'Rorke, 1990] P. O'Rorke. Integrating abduction and learning. In *Proceedings of the AAAI Symposium on Automated Abduction*, 1990.

[Pearl, 1983] J. Pearl. On the discovery and generation of certain heuristics. *AI Magazine*, pages 23–33, 1983.

[Plaisted, 1981] D. Plaisted. Theorem proving with abstraction. *Artificial Intelligence*, 16:47–108, 1981.

[Rajamoney and DeJong, 1987] S. Rajamoney and G. DeJong. The classification, detection and handling of imperfect theory problems. In *Proceedings of IJCAI-87*, pages 205–207, 1987.

[Riddle, 1990] P. Riddle. Automating problem reformulation. In D. Paul Benjamin, editor, *Change of Representation and Inductive Bias*, pages 105–124. Kluwer, 1990.

[Rosenbloom and Laird, 1986] P. S. Rosenbloom and J. E. Laird. Mapping explanation-based generalization onto soar. In *Proceedings of the National Conference on Artificial Intelligence*, pages 561–567, August 1986.

[Rosenbloom *et al.*, 1987] P. S. Rosenbloom, J. E. Laird, and A. Newell. Knowledge level learning in soar. In *Proceedings of the National Conference on Artificial Intelligence*, pages 499–504, August 1987.

[Rosenbloom *et al.*, 1991a] P. S. Rosenbloom, J. E. Laird, A. Newell, and R. McCarl. A preliminary analysis of the Soar architecture as a basis for general intelligence. *Artificial Intelligence*, 47:289–325, 1991.

[Rosenbloom *et al.*, 1991b] P.S. Rosenbloom, A. Newell, and J.E. Laird. Towards the knowledge level in soar: The role of the architecture in the use of knowledge. In

K. VanLehn, editor, *Architectures for Intelligence.* Lawrence Erlbaum Associates, Hillsdale, NJ, 1991, 1991.

[Rosenbloom *et al.*, 1992] P. Rosenbloom, S. Lee, and A. Unruh. Bias in planning and explanation-based learning. In S. Chipman and A. Meyrowitz, editors, *Machine Learning: Induction, Analogy and Discovery.* Kluwer Academic Publishers, 1992. In Press. (Also available in S. Minton (Ed.) *Machine Learning Methods for Planning and Scheduling*: Morgan Kaufmann. In Press.).

[Ruby and Kibler, 1991] D. Ruby and D. Kibler. Steppingstone: An empirical and analytical evaluation. In *Proceedings of the Ninth National Conference on Artifical Intelligence*, pages 527–532, 1991.

[Sacerdoti, 1974] E. D. Sacerdoti. Planning in a hierarchy of abstraction spaces. *Artificial Intelligence*, 5:115–135, 1974.

[Sacerdoti, 1977] E. D. Sacerdoti. *A Structure for Plans and Behavior.* Elsevier, New York, 1977.

[Schoppers, 1990] M. Schoppers. Universal plans for reactive robots in unpredictable domains. In *Proceedings of the Tenth International Joint Conference on Artificial Intelligence*, 1990.

[Segre and Turney,] A.M. Segre and J. Turney. Planning, acting, and learning in a dynamic domain. In S. Minton, editor, *Machine Learning Methods for Planning.* Morgan Kaufmann. In Press. To appear March 1993.

[Stefik, 1981] M. J. Stefik. Planning with constraints. *Artificial Intelligence*, 16:111–140, 1981.

[Subramanian and Genesereth, 1987] D. Subramanian and M. Genesereth. The relevance of irrelevance. In *Proceedings of IJCAI-87*, pages 416–422, 1987.

[Tate, 1977] A. Tate. Generating project networks. In *Proceedings of IJCAI-77*, pages 888–893, 1977.

[Tenenberg, 1988] J. Tenenberg. *Abstraction in Planning.* PhD thesis, University of Rochester, 1988.

[Unruh *et al.*, 1987] A. Unruh, P. S. Rosenbloom, and J. E. Laird. Dynamic abstraction problem solving in soar. In *Proceedings of the Third Annual Conference on Aerospace Applications of Artificial Intelligence*, pages 245–256, October 1987.

[Unruh and Rosenbloom, 1989] A. Unruh and P. S. Rosenbloom. Abstraction in problem solving and learning. In *Proceedings of IJCAI-89, Eleventh International Joint Conference on Artificial Intelligence.* Morgan Kaufmann, 1989.

[Unruh and Rosenbloom, 1990] A. Unruh and P. S. Rosenbloom. Two new weak method increments for abstraction. In *Proceedings of the AAAI Workshop on Automatic Generation of Approximation and Abtractions*, July 1990.

[Valtorta, 1984] M. Valtorta. A result on the computational complexity of heuristic estimates for the A* algorithm. *Information Sciences*, 34:47–59, 1984.

[Veloso, 1989] M. Veloso. Nonlinear problem solving using intelligent causal-commitment. Technical Report CMU-CS-89-210, School of Computer Science, Carnegie-Mellon University, 1989.

[Weld, 1992] D. Weld. Automatic selection of bounding abstractions. In *AAAI-92 Workshop on Approximation and Abstraction of Computational Theories*, pages 227–234, 1992.

[Wilkins, 1984] D. E. Wilkins. Domain-independent planning: representation and plan generation. *Artificial Intelligence*, 22:269–301, 1984.

[Wilkins, 1988] D. E. Wilkins. *Practical Planning.* Morgan Kaufman, 1988.

[Williams, 1992] C. Williams. Imperfect abstraction. In *AAAI-92 Workshop on Approximation and Abstraction of Computational Theories*, pages 243–249, 1992.

[Yang and Tenenberg, 1990] Yang and Tenenberg. ABTWEAK: Abstracting a nonlinear, least commitment planner. In *Proceedings of the Eigth National Conference on Artificial Intelligence.* AAAI Press/The MIT Press, 1990.

[Yang, 1990] Q. Yang. Solving the problem of hierarchical inaccurarcy in planning. In *Proceedings of the Eight Biennial Conference of the Canadian Society for Computational Studies of Intelligence*, pages 140–145, 1990.