SIMULATION-BASED SEARCH FOR HYBRID SYSTEM CONTROL AND ANALYSIS

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
Todd William Neller
June 2000

© Copyright 2000 by Todd William Neller
All Rights Reserved

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

Richard E. Fikes
(Principal Adviser)

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

Claire J. Tomlin
Department of Aeronautics and Astronautics

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

Feng Zhao
Xerox Palo Alto Research Center

Approved for the University Committee on Graduate Studies:

Preface

This dissertation explores new algorithmic approaches to simulation-based optimization, game-tree search, and tree search for the control and analysis of hybrid systems. Hybrid Systems are systems that evolve with both discrete and continuous behaviors. Examples of hybrid systems include diverse mode-switching systems such as those we have used as focus problems: stepper motors, magnetic levitation units, and submarine detection avoidance scenarios. For hybrid systems with complex dynamics, the designer may have little other than simulation as a tool to detect design flaws or inform offline or real-time control. In approaching control and analysis of such systems, we thus limit ourselves to a black-box simulation of the system, assuming as little as possible about the underlying dynamics and extending various types of search algorithms to treat these difficult general cases.

Chapter 1 provides the reader with a more detailed overview, a summary of contributions, background reading, and chapter dependencies.

Chapter 2 presents a stepper motor control design problem where the designer wishes to use simulation to efficiently detect rare stall scenarios in the space of possible system parameters and initial states if such scenarios exist. A survey of global optimization techniques and extensions of such techniques are made, and we discover the importance of novel information-based and multi-level optimization methods.

Chapters 3–6 focus on game-tree search and tree search problems where a series of actions must be chosen under different assumptions about the existence of a given action or action timing discretization. If the search algorithm is given an action or action timing discretization, we say that the search algorithm has "static action discretization" or "static action timing discretization" respectively. If the search algorithm is

not given an action or action timing discretization, we say the search algorithm has "dynamic action discretization" or "dynamic action timing discretization" respectively. Thus various assumptions about whether or not either discretization is given define four quadrants:

		Action **T**iming Discretization: **S**tatic	**A**ction **T**iming Discretization: **D**ynamic
Action Discretization	**S**tatic	**SASAT**	**SADAT**
	Dynamic	**DASAT**	**DADAT**

The acronyms in each quadrant are used in this dissertation to keep track of these underlying assumptions about action and action timing discretization.

Chapter 3, SASAT game-tree search, presents a magnetic levitation control problem as an adversarial game for the purpose of robust control synthesis. We explore the use of a game-graph (augmented cell-map) approximation and alpha-beta pruning technique for fast adaptive online control.

Chapter 4, DASAT game-tree search, continues with the magnetic levitation control problem and instead focuses on the issue of action discretization for game-tree search. A novel application of information-based optimization to alpha-beta search is presented.

Chapter 5, SADAT tree search, presents a submarine detection avoidance problem as a solitaire game or search for the purpose of fast, real-time tactical planning assistance. Assuming discretized actions, we focus on the problem of action timing discretization. New iterative refinement techniques and a variant of best-first search are presented.

Chapter 6, DADAT tree search, continues with the submarine detection avoidance problem and removes the assumption of discretized actions. Augmenting the algorithms of the previous chapter, we explore random, information-based, and dispersed dynamic discretization of actions in search.

Acknowledgments

My years at Stanford have given me cause for much thankfulness. As my life beyond this work has been filled with encouraging people, a full, detailed set of acknowledgments could easily dwarf the rest of the dissertation. To be brief, this work owes its successful completion to many excellent people of whom I name only a few.

I am thankful for the generous funding which enabled me to do this work. This work was supported by the Defense Advanced Research Projects Agency and the National Institute of Standards and Technology under Cooperative Agreement 70NANB-6H0075, "Model-Based Support of Distributed Collaborative Design", the Stanford Gerald J. Lieberman Fellowship, and NASA Grant NAG2-1337. I am especially thankful for the efforts of my advisor Richard Fikes in obtaining this funding.

I am thankful for the people who took the time to share interesting research problems with me. For the stepper motor problem, I would like to thank Dana Clarke for describing the problem, and Bert Leenhouts for sharing his stepper motor modeling expertise. For the magnetic levitation problem, I would like to thank Feng Zhao and Jeff May for sharing their work. For the submarine tactical planning problem, I would like to thank David Watson, Chip McVey, and Adam Peterson for taking the time to describe their problem and prior work in detail. These people provided the seeds from which this work grew.

I am thankful for the education I have received which has enabled me to approach these problems. I would like to thank the faculty of Stanford University, Cornell University, and Phillips Exeter Academy for an enjoyable and valuable education. I would like to offer special thanks to my parents and grandparents for their financial support of my education.

I am thankful for the mentorship I have received as a Ph.D. student. Thank you, Richard, for being my advisor, for making yourself very available for helpful discussion, and for giving me such freedom to pursue these research interests. Thank you, Sheila McIlraith, for sharing my enthusiasm for this work and offering generous encouragement and advice far beyond the call of duty.

I am thankful for the readers who have helped me improve this dissertation amidst great time pressures. Thank you, Richard, for your skilled, careful reading of many drafts of this work over many years. Thank you, Feng Zhao, for your willingness to become involved in my reading committee and as a consulting professor here at Stanford. Thank you, Claire Tomlin, for your willingness to reach out and investigate unfamiliar work in AI, and for helping me understand the vocabulary and concepts of control theory. Only with researchers such a those on my committee can such interdisciplinary work be successful.

I am thankful for the Computer Science Department Staff, who have worked hard behind the scenes to support and bring cheer to my work environment. I have been very impressed by the friendliness and care of our staff, and am thankful to have found precious friends among them. I am especially thankful for Knowledge Systems Laboratory's long-time administrator Grace Smith, Responsible Person who Sees All and Knows All.

I am thankful for the loving support of my wife Johanna, my family, my friends, and my God. Johanna, thank you for your love, encouragement, and optimism. I want to offer thanks to my family for helping me discover my gifts and interests and encouraging me to pursue them with joy. I offer many thanks to many friends for blessing me with warmth, humor, and special memories through these years. Most of all, I praise God for creating such a marvelous world, for being the author of love, and for giving me the gift of experiencing the wonder of both.

For all these things, I am very thankful.

Contents

Preface **iv**

Acknowledgments **vi**

1 Introduction **1**

1.1 Motivation 1

1.2 Problem Characterizations 2

1.2.1 Simulation-based Global Optimization for Initial Safety Refutation of Hybrid Systems 4

1.2.2 Simulation-based Game-Tree Search for Robust Control Synthesis of Hybrid Systems 6

1.2.3 Simulation-Based Tree Search for Real-Time Control Assistance of Hybrid Systems 9

1.3 Contributions 10

1.4 Vision 12

1.5 Reading Guide 14

2 Heuristic Optimization for Initial Safety Refutation **16**

2.1 Introduction 16

2.2 Stepper Motor Stall Problem 17

2.3 Algorithms and test problems 20

2.4 Results 23

2.5 Conclusions of Comparative Study 29

2.6 Information-based global optimization 29

2.6.1 Strongin's Information Approach 30
2.6.2 Information-Based Optimization for Refutation 33
2.7 Multi-Dimensional, Multi-Level Information-Based Optimization . . . 34
2.7.1 Decision procedure 34
2.7.2 Multi-Level Local Optimization 37
2.7.3 MLLO-IQ and MLLO-RIQ 39
2.8 Experimental results 40
2.9 Conclusions 44

3 SASAT Game-Tree Search **46**
3.1 SASAT Hybrid System Game and Search Problem 47
3.2 Magnetic Levitation Problem 49
3.3 SASAT Dynamic Programming Game-Graph Method 51
3.4 SASAT Generalized Hybrid Alpha-Beta Method 55
3.5 Experimental Results 57
3.6 SASAT Alpha-Beta on a Game Graph 59
3.7 Relation to Memory-Based Techniques 70
3.8 Summary and Discussion 73

4 DASAT Game-Tree Search **76**
4.1 DASAT Hybrid System Game and Search Problem 77
4.2 DASAT Magnetic Levitation Problem 79
4.3 DASAT Alpha-Beta Search with Random Discretization 79
4.4 DASAT Alpha-Beta Search with Uniform Discretization 82
4.5 DASAT Information-Based Alpha-Beta Search 85
4.6 Comparison of Methods 90
4.7 Conclusions 92

5 SADAT Search **94**
5.1 SADAT Hybrid System Game and Search Problem 95
5.2 Submarine Channel Problem 97
5.2.1 The Submarine Tactical Planning Assistant 97

5.2.2 The SADAT Submarine Channel Problem 98
5.3 SADAT Iterative Refinement Search 100
5.4 SADAT Best-First Search 104
5.4.1 Simple SADAT Best-First Search 105
5.4.2 SADAT Best-First Search with Refinement Limits 110
5.5 SADAT Iterative Refinement with Strong Pruning, Node Ordering, and Upper Bound 115
5.6 SADAT Iterative Refinement with Recursive Best-First Search 119
5.6.1 SADAT ϵ - Recursive Best-First Search with Fixed Delay . . . 120
5.6.2 SADAT Iterative Refinement with ϵ - Recursive Best-First Search 121
5.7 Summary and Conclusions 124

6 DADAT Search **127**
6.1 DADAT Hybrid System Game and Search Problem 128
6.2 DADAT Submarine Channel Problem 130
6.3 DADAT Iterative Refinement with Random Action Discretization . . 130
6.4 DADAT Iterative Refinement with Information-Based Action Discretization 134
6.5 DADAT Iterative Refinement with Dispersed Action Discretization . 138
6.6 DADAT Iterative Refinement with Dispersed ϵ-RBFS 143
6.7 Conclusions 147

Bibliography **150**

List of Tables

2.1 Stepper Motor Acceleration Table 18
2.2 Algorithm Quick Reference 22
2.3 Objective Function Quick Reference 23
2.4 Successful global optimization trials and average function evaluations 25
2.5 Results for `STEP1` and `STEP2` 28
2.6 Successful global optimization trials and average function evaluations 42
2.7 Results for `STEP1` and `STEP2` 44

4.1 Results for DASAT Alpha-Beta Search with Random Discretization . 81
4.2 Results for DASAT Alpha-Beta Search with Uniform Discretization . 84
4.3 Results for DASAT Information-Based Alpha-Beta Search 89
4.4 Comparison of Effective Branching Factor Reduction 90
4.5 Results for Random versus Uniform Discretization 91
4.6 Results for Random versus Information-Based Discretization 92
4.7 Results for Uniform versus Information-Based Discretization 92

5.1 Results for SADAT Simple Iterative Refinement DFS 104
5.2 Results for SADAT Simple Iterative Refinement DFS with Goal Node Termination 105
5.3 Results for SADAT Best-First Search, $\Delta t = 1.05$ 111
5.4 Results for SADAT Best-First Search, $\Delta t = 1.40$ 115
5.5 Results for SADAT Best-First Search, $\Delta t = 1.51$ 115
5.6 Results for SADAT Iterative Refinement with Strong Pruning, Node Ordering, and Upper Bound 118

5.7 Results for SADAT ϵ - Recursive Best-First Search, $\epsilon = 0.25$ 122
5.8 Results for SADAT Iterative Refinement with ϵ - Recursive Best-First Search, $\epsilon = 0.25$ 123

6.1 Results for DADAT Iterative Refinement with Random Action Discretization 131
6.2 Results for SADAT Iterative Refinement with Strong Pruning, Node Ordering, Upper Bound, and Randomly Rotated Action Discretization 134
6.3 Results for DADAT Iterative Refinement with Dispersed Action Discretization 143
6.4 Results for DADAT Iterative Refinement with Dispersed ϵ-RBFS . . 145
6.5 Results for SADAT Iterative Refinement with ϵ-RBFS and Randomly Rotated Action Discretization 145

List of Figures

1.1 Chapter Dependencies 15

2.1 Simple Stepper Motor Stepping 18
2.2 Stepper Motor Test Problem `STEP1` 24
2.3 Stepper Motor Test Problem `STEP2` 24
2.4 Refinement of Peano curve in two dimensions 32
2.5 Shadowing example 36
2.6 Information-based global optimization of 2-D circular paraboloid . . . 39

3.1 Schematic of magnetic levitation system. 49
3.2 Block diagram of magnetic levitation system operation 49
3.3 Maglev output currents from the SASAT dynamic programming game-graph method, depth 2 60
3.4 Maglev output currents from the SASAT dynamic programming game-graph method, depth 4 60
3.5 Maglev output currents from the SASAT dynamic programming game-graph method, depth 6 61
3.6 Maglev output currents from the SASAT dynamic programming game-graph method, depth 8 61
3.7 Maglev trajectories from the SASAT dynamic programming game-graph method depth 2 62
3.8 Maglev trajectories from the SASAT dynamic programming game-graph method, depth 4 62

3.9 Maglev trajectories from the SASAT dynamic programming game-graph method, depth 6 63
3.10 Maglev trajectories from the SASAT dynamic programming game-graph method, depth 8 63
3.11 Maglev trajectory scores from the SASAT dynamic programming game-graph method, depth 2 64
3.12 Maglev trajectory scores from the SASAT dynamic programming game-graph method, depth 8 64
3.13 Maglev trajectories from the SASAT alpha-beta method, depth 2 (with current changes) 65
3.14 Maglev trajectories from the SASAT alpha-beta method, depth 4 (with current changes) 65
3.15 Maglev trajectories from the SASAT alpha-beta method, depth 2 (without current changes) 66
3.16 Maglev trajectories from the SASAT alpha-beta method, depth 4 (without current changes) 66

4.1 Maglev output currents from DASAT Alpha-Beta with Random Discretization, depth 2 80
4.2 Maglev output currents from DASAT Alpha-Beta with Random Discretization, depth 4 80
4.3 Maglev output currents from DASAT Alpha-Beta with Random Discretization, depth 6 81
4.4 Maglev output currents from DASAT Alpha-Beta with Uniform Discretization, depth 2 83
4.5 Maglev output currents from DASAT Alpha-Beta with Uniform Discretization, depth 4 83
4.6 Maglev output currents from DASAT Alpha-Beta with Uniform Discretization, depth 6 84
4.7 Maglev output currents from DASAT Information-Based Alpha-Beta, depth 2 88

4.8 Maglev output currents from DASAT Information-Based Alpha-Beta, depth 4 88
4.9 Maglev output currents from DASAT Information-Based Alpha-Beta, depth 6 89

5.1 Tactical Planning Associate Man-Machine Interface illustrating Generative Layer, from [46, Figure 6] 98
5.2 Submarine Channel Problem 99
5.3 Submarine Channel Problem Demo, 4 Ships 101
5.4 Submarine Channel Problem Demo, 10 Ships 102
5.5 Iterative Refinement 102
5.6 SADAT Best-First Search 108

6.1 Information-Based Optimization point choices for a finite values and an infinite target, confined to a circular region 139

Chapter 1

Introduction

1.1 Motivation

This dissertation explores new algorithmic approaches to simulation-based optimization, game-tree search, and tree search for the control and analysis of hybrid systems. Hybrid Systems are systems that evolve with both discrete and continuous behaviors. Examples of hybrid systems include diverse mode-switching systems such as those we have used as focus problems: stepper motors, magnetic levitation units, and submarine detection avoidance scenarios. For hybrid systems with complex dynamics, the designer may have little other than simulation as a tool to detect design flaws or inform offline or real-time control. In approaching control and analysis of such systems, we thus limit ourselves to a black-box simulation of the system, assuming as little as possible about the underlying dynamics and extending various types of search algorithms to treat these difficult general cases.

In fact, the system dynamics need not include both continuous and discrete changes. For optimization, we are interested in systems that tend to have similar behavior for similar initial conditions. For game-tree search and search, we are interested in systems for which simulation and control actions can be used to explore branching possibilities of system evolution in order to inform intelligent control action.

For each problem area, a representative problem was chosen to focus our work. For global optimization, Chapter 2 presents a stepper motor control design problem

where the designer wishes to use simulation to efficiently detect rare stall scenarios in the space of possible system parameters and initial states if such scenarios exist. The stepper motor system is hybrid in the sense that the system evolves with piecewise continuous intervals separated by scheduled coil voltage changes modeled as discrete events.

For game-tree search, Chapter 3 presents a magnetic levitation control problem as an adversarial game for the purpose of robust control synthesis. The magnetic levitation system is hybrid in the sense that the system evolves with piecewise continuous intervals separated by controlled input changes modeled as discrete events. Both the stepper motor and magnetic levitation systems are essentially continuous systems with fast controlled changes approximated as occurring instantaneously.

For tree search, Chapter 5 presents a submarine detection avoidance problem as a solitaire game or search for the purpose of fast, real-time tactical planning assistance. This problem is hybrid in the sense that the system evolves with piecewise continuous intervals separated by controlled and autonomous discrete events[1]. For a thorough review and unification of hybrid system models, see Branicky's dissertation[5].

In each case, we have sought to avoid use of complex problem-domain-specific knowledge. One can often trade off generality for performance through the use of domain-specific knowledge. As we formalize new problems and take first steps to address them in this dissertation, we take care to minimize the assumed knowledge of our problem domains so that the algorithms developed may serve as generally applicable kernels from which future advances can grow.

Each of the following chapters begins with a formal definition of the problem of interest. We now place these problems in perspective with one another.

1.2 Problem Characterizations

In Russell & Norvig's "Artificial Intelligence: a modern approach"[41], an agent-based approach to problem definition is used, where an agent maps percepts to actions

[1]For this problem, controlled and autonomous discrete events are changes in submarine and ship headings, speeds, and modes.

within a dynamical system. A PAGE description of an agent includes four basic components:

- **Percepts** - what the agent is able to sense about its environment,
- **Actions** - what the agent is able to affect in its environment,
- **Goals** - what the agent wishes to achieve in its environment, and
- **Environment** - a description of the environment itself.

From an optimal control viewpoint, this would be like taking a controller-centric approach to problem definition with each of these components respectively corresponding to controller inputs, controllers outputs, performance index[2], and plant.

Additionally, environment descriptions make the following distinctions:

- **Accessible vs. Inaccessible** - If the agent senses the entire state of the environment relevant to achieving its goal, the environment is *accessible*. Otherwise it is *inaccessible*. For example, chess as a game of perfect information is accessible, whereas poker as a game of imperfect information is inaccessible.
- **Deterministic vs. Nondeterministic** - If the next state of the environment is completely determined by the current state and the actions of the agents, the environment is *deterministic*. Otherwise, it is *nondeterministic*. For example, chess as a game without chance is deterministic, whereas poker as a game of chance is nondeterministic. Such (non)determinism is usually defined with respect to the agent's perspective. From the perspective of poker playing agents, cards drawn are not determined by the agents themselves and are a source of nondeterminism in game play.
- **Episodic vs. Nonepisodic** - If the agent's experience in the environment can be divided into separate "episodes" (i.e. a single mapping from percepts to

[2] "Performance index" may also be called "objective function" or "utility function" in other control contexts.

actions) which have no influence on the utility of actions in all other episodes, the environment is *episodic*. Otherwise, it is *nonepisodic*. The single-shot chance game of rock-scissors-paper is episodic, whereas the complex sequential nature of chess is nonepisodic.

- **Static vs. Dynamic** - If the environment cannot change while the agent is deliberating, the environment is *static*. Otherwise, the environment is *dynamic*. Chess is static[3], whereas baseball is dynamic.

- **Discrete vs. Continuous** - If there are a limited number of distinct percepts and actions, then the environment is *discrete*. Otherwise, it is *continuous*. With enumerable board positions, chess is discrete, whereas baseball is continuous.

We now characterize each of our problems in turn and discuss further particulars of each.

1.2.1 Simulation-based Global Optimization for Initial Safety Refutation of Hybrid Systems

For this problem, we are interested in detecting design flaws within an initial time period of simulation. Given a set of possible initial conditions (possible system parameters and initial states), we wish to know if a predefined controller remains within a desired set of "safe" states for an initial time period. We call this property "initial safety". Since the system is entirely predefined with no degrees of freedom for decision making, the controller is in this case a degenerate case of an agent, with neither percepts nor actions which can be used to deliberate about or affect achievement of the goal within the environment.

However, the task of *refuting* initial safety presents a more interesting study. A PAGE description of the initial safety refutation agent is as follows:

- **Percepts** - The agent perceives the current possible initial condition under consideration, and the evaluated heuristic measure of relative safety of a trajectory

[3] Or else the opponent's hand gets slapped for playing out of turn.

simulated from that initial condition.

- **Actions** - The agent chooses the next possible initial condition to consider and evaluates the heuristic measure of relative safety of the trajectory simulated from this initial condition.
- **Goals** - The agent wishes to, with minimal actions, find an initial condition for which simulation yields a trajectory with an unsafe state, thus refuting initial safety of the system.
- **Environment** - An offline simulation testing environment which is:
  - **Accessible** - The agent may obtain a heuristic evaluation of the relative safety of any possible initial condition.
  - **Deterministic** - Simulation and evaluation of the simulation is deterministic with respect to initial conditions.
  - **Nonepisodic** - Since the agent seeks to minimize the number of actions needed for refutation (if a refutation exists), each action affects the performance overall.
  - **Static** - The testing environment never changes.
  - **Continuous** - Perceived evaluations can include all non-negative real numbers. The range of possible actions is over a continuous space of possible system parameters and initial states.

One might wonder why we have the agent seek to minimize the number of actions rather than time. The reason is that we make the assumption that the computational cost of simulation and evaluation dominates the cost of the agent deliberation. In doing so, we approximate the goal of minimizing overall computational time by minimizing the number of calls to the most computationally expensive procedure.

We did not work with initial condition spaces with more than 6 dimensions. Such problems are often addressed by performing successive searches in lower-dimensional subspaces.

1.2.2 Simulation-based Game-Tree Search for Robust Control Synthesis of Hybrid Systems

Control theorists have long posed control problems as games in order to treat multi-agent control problems (e.g. pursuit-evasion games) or robust control problems (e.g. where the adversary represents worst case external perturbation, error, etc.)[1, 7]. We characterize these problems from the perspective of the first-player control agent as follows:

- **Percepts** - The agent perceives the current state of the hybrid system.
- **Actions** - In Chapter 3, the agent chooses from a discrete set of possible actions. In Chapter 4, the agent chooses from a set of possible closed, continuous action parameter regions.
- **Goals** - The agent wishes to maximize its score (utility) with respect to a given time horizon.
- **Environment** - A multi-agent hybrid dynamical system which is:
  - **Accessible** - The agent can perceive all hybrid state variables relevant to achieving its goal.
  - **Deterministic** - The actions of the players completely determine the dynamics of the system.
  - **Nonepisodic** - Each action can affect the system dynamics thus affecting the score/utility of future actions.
  - **Dynamic** - While a player is deliberating, another player can act and change the environment.[4]
  - **Continuous** - Both percepts (all state variables) and actions (chosen from action parameter regions) can be continuous.

[4] In Chapters 3 and 4, we simplify the problem by approximating the dynamic game as one in which the players take turns at fixed times. We approach the dynamic problem with a static approximation of the problem.

Now that we have characterized the general features of the problem, it is important to characterize problems for which game-tree search is a suitable approach. Beyond the commonalities we have discussed, good game-tree search applications also share informational/topological characteristics. In searching possible lines of play from the current state, the game-tree formed must contain *sufficient information* within a *limited depth*, given a *low branching factor*, to indicate intelligent action under *player modeling assumptions*:

1. **Information** - Like any process which works with information to form conclusions, one can expect the adage "garbage in, garbage out" to hold. Whether in the form of a utility function or a heuristic function estimating utility, one must have a means of evaluating the desirability of one sequence of moves over another. While such a function need not be perfect, poor information will lead to poor decisions. At the other extreme, a perfect utility function obviates the need for search. If the expected utility of performing a single move is perfectly computable, one need only look ahead one move. Game-tree search is better suited for games which benefit from a combination of lookahead and imperfect evaluation. Typically the expected utility of a move sequence is composed of one or both of the following: (a) the utility of performing the sequence of actions in the current state, and (b) an estimate of the utility of actions which will be chosen thereafter. A search technique which makes use of (a) only is said to exhibit "greedy" behavior.

2. **Search Depth** - Game-tree search can be thought of as an optimization in the space of move sequences under player modeling assumptions (see (4)). Given that such spaces can be vast for small, simple games, methods often assume that search will cover a small subset of move sequences, generally biased towards the shortest sequences. Often, such subsets of action sequences will have no path, or no optimal path which leads to a goal state (victory). The time required to perform search grows exponentially as $O(b^d)$, where b is the effective branching factor of the tree, and d is the search depth. Obviously, even for small branching factors, game-tree searches will only be successful in domains where limited

lookahead is sufficient to inform intelligent action.

3. **Search Breadth** - For the same reason, high branching factors can also render search ineffective. With players alternately placing pieces on a 19 $\times$ 19 grid, the game of Go provides a good example of how a high branching factor can make lookahead too computationally expensive for effective use. Game-tree search is best applied to games where branching factor is not so high as to prevent sufficient lookahead to inform intelligent action. For a continuous or hybrid game with infinite possible moves defined by continuous action parameter spaces, we can only sample a finite number of moves. Feasibility of search for approaching such problems depends on how well sampling can provide global information about the quality of decisions.

4. **Player Modeling Assumptions** - Rational game-play is based on player modeling assumptions. Although most game-theoretic research is focused on optimal rational play, understanding of one's opponent allows better game play. For instance, one can play chess well assuming that one's opponent approximates perfect rational play. However, if one knows that the opponent strongly favors material advantage, then one will do better to favor the strategy of sacrifice. Game-tree search techniques usually have very simple player models which are computationally efficient. The minimax assumption is an example.

So information characteristics concerning (expected) utility of moves and player modeling is intertwined with topological characteristic of search-tree depth and breadth. Put simply, there must be sufficient information in the possibilities we can consider during search to make intelligent choices. Beyond environmental characteristics, these form the core considerations for game-tree search applications.

One final important note is the distinction between the effect of the dimensionality of the state space versus the effect of the dimensionality of the action parameter regions. As the dimensionality of the state space increases, the computational complexity of *simulation* is affected. As the dimensionality of action parameter regions

increases, the effective branching factor of *search* increases exponentially to maintain the same granularity of discretization[5]. So long as the system can be simulated quickly, dimensionality of the state space is not a concern for the complexity of the search. Biologists have observed that complex behaviors in organisms with many degrees of freedom in movement arise from superposition of very simple signals of varying intensity[2]. If one can choose an appropriate low-dimensional parameterization of action, search has the potential to inform intelligent action of complex systems.

1.2.3 Simulation-Based Tree Search for Real-Time Control Assistance of Hybrid Systems

Tree search (or simply "search") can be viewed as a special solitaire case of game-tree search where there is only one player. The general challenge is to find a sequence of actions which either maximizes a score/utility/payoff, minimizes a cost, or achieves a desired state or set of states. We characterize these problems from the perspective of the first-player control agent as follows:

- **Percepts** - The agent perceives the current state of the hybrid system.
- **Actions** - In Chapter 5, the agent chooses from a discrete set of possible actions. In Chapter 6, the agent chooses from a set of possible closed, continuous action parameter regions.
- **Goals** - We treat multiple different goals in this context which take on some combination of (1) minimizing cost with respect to a given time horizon, and (2) achieving a desired goal state or set of states. Methods are presented which pursue (1) only, pursue (1) and stop if (2) is achieved, and pursue (2) making sure the cost is approximately optimal.
- **Environment** - A multi-agent hybrid dynamical system which is:

[5]Granularity is defined with respect to Euclidean distance of sampled action parameter points.

- **Accessible** - The agent can perceive all hybrid state variables relevant to achieving its goal.
- **Deterministic** - The actions of the agent completely determine the dynamics of the system.
- **Nonepisodic** - Each action can affect the system dynamics thus affecting the score/utility of future actions.
- **Static** - The agent is the sole affector of the environment.
- **Continuous** - Both percepts (all state variables) and actions (chosen from action parameter regions) can be continuous.

In Chapters 5 and 6, we no longer assume a given action timing discretization. In Chapter 5, we assume a given action discretization. In Chapter 6, we do not.

As a degenerate case of game-tree search, all preceding discussion of applicability beyond environmental concerns is relevant except for discussion concerning player modeling assumptions. To reiterate, in searching possible sequences of actions from the current state, the tree searched must contain *sufficient information* within a *limited depth*, given a *low branching factor*, to indicate intelligent action.

1.3 Contributions

In this section, we summarize the algorithmic contributions of this research. Beyond algorithmic contributions, Chapter 2 presents the definition of an initial safety problem and a novel reformulation of the problem to a specialization of global optimization. Chapters 3–6 each formally define hybrid system games and search problems under differing assumptions of action and action timing discretizations.

In Chapter 2, we present the first multidimensional approach to information-based optimization and the first local optimization application of the information-based optimization approach. We generalized the multi-level local optimization architecture of [10], and created two information-based multi-level optimization methods which were the only algorithms we found able to reliably find design faults with our difficult

stepper motor test problem. In addition, we created multi-level single-linkage[39] variants which assumed local optimization determinism, used ordering heuristics, and performed lazy objective function evaluation. Finally, we made constrained, epsilon-descent variants of quasi-Newton and Yuret's local optimization[54].

In Chapters 3–6, we develop game-tree search and search techniques for control of hybrid systems. In contrast to classical control techniques such as feedback linearization, we do not constrain our system to a specific analytical form. For most of our algorithms, we assume that a system simulator is given. However, the augmented cell-map techniques of Chapter 3 require only sufficient time-series data to approximate system dynamics. Furthermore, simulation can be approximated through the interpolation of time-series data (e.g. linear weighted regression from observed behavior[32]). From this perspective, our techniques not only enable model-based control, but also can be applied without explicit models given an appropriate means of interpolating unseen system behavior.

In Chapter 3, we present a new synthesis of cell-map and minimax methods for fast approximate control synthesis. We augmented a cell-map for multi-player evaluation, calling it a game-graph. We present two algorithms which are respectively suited for offline and online derivation of optimal control: Dynamic Programming on a Game-Graph and Alpha-Beta Pruning on a Game-Graph.

In Chapter 4, we show that alpha-beta search naturally provides bounds for the application of information-based optimization to the discretization of continuous action parameter spaces. We call the resulting algorithm Information-Based Alpha-Beta Search, and show empirically that it exceeds the good speed and pruning performance of random discretization while matching the control policy quality of uniform discretization.

In Chapter 5, we provide several new search approaches that do not rely on a given fixed action timing discretization. Simple Iterative Refinement successively searches for a solution from the initial time to a fixed time horizon with increasingly finer granularity until a solution is found. SADAT Best-First Search, the first systematic search that dynamically generates new internal nodes, was shown to exhibit a tradeoff

of speed versus solution quality. Iterative Refinement with Strong Pruning, Node Ordering, and Upper Bound yielded impressive performance given an appropriate time horizon and a monotonic heuristic evaluation function. We next created an epsilon variant of Korf's Recursive Best-First Search[25] and showed its extreme sensitivity to the input delay parameters. We conclude the chapter with a successful synthesis of ϵ - Recursive Best-First Search with iterative refinement ideas. Iterative Refinement with ϵ - Recursive Best First Search gave excellent results while behaving most consistently with respect to a wide range of initial delay parameters.

In Chapter 6, we describe the augmentation of the best new searches from the previous chapter with three forms of dynamic discretization: random, information-based, and dispersed. The previous chapter relied on a human-designed discretization which was aligned with topological features and object motion of the test problem domain. We repeated experiments from Chapter 5 with the given heading discretizations randomly rotated. Dynamic random discretization performed similarly to the randomly rotated static discretization. The computational complexity of information-based optimization made it unsuitable for the real-time requirements of the test problem. We developed a compromise between the speed of random discretization and the principled approach of information-based discretization. The compromise, called dispersed discretization, yielded performance far exceeding that of the randomly rotated static discretization.

1.4 Vision

While one might argue that control and AI researchers intersect in the study of neural networks, it appears that there is no significant intersection between AI and control game research. Constructing a program to make a computer play chess well primarily affects a philosophical change in the world, necessitating new conclusions about the nature of intelligence. However, constructing programs that think and act intelligently in continuous physical domains affects a material change in the world, creating new opportunities for practical application of computers.

We believe that the extension of discrete AI search techniques to hybrid control

domains can be of great benefit to both AI and control. By increasing the common ground of common goals, we hope to facilitate the promising merge of AI discrete system expertise and control continuous system expertise. Applications we envision are described below.

- Design fault detection: While a discrete search of a hybrid space is not complete, it can be an efficient means of detecting faulty behaviors without needing to over-abstract or over-approximate the model. We imagine a control engineer taking sources of error or uncertainty and modeling them as a player or players that seek to work against the controller. The game-tree search would then be an efficient means of searching for the most significant possible deviations from intended behavior.

- Robust control: In treating possible disturbances or errors as possible actions of an adversarial player in a control game, the objective of optimal game play is equivalent to the objective of robust control. We will see two different search approaches to robust control in Chapters 3 and 4. In one approach, we appromixate the continuous system as a graph and apply various forms of dynamic programming to compute optimal robust control for the approximated system. In the other approach, we perform a tree search of a sample of possible system trajectories. Since the discrete game-tree search of the continuous system is incomplete, it can only be considered an approximation of robust control to the extent that we can prove properties about the most that our sampling will miss in the course of search.

- Online control: For applications where safety is not critical, the online use of game-tree search or tree-search for control decisions may provide an immediate, approximate model-in-controller-out methodology for control. Using simulation to project the system state Δt time units into the future, we search from the projected point for Δt time units and use the results of search to inform control action. Such controllers would be especially useful in applications requiring exception versatility in adaptive control. Even if one cannot parameterize changes

in the model, one would need only change the simulation and/or cost model in order to adapt controller behavior to a new environment and/or goal.

- Rapid prototyping: In the design stage, we also believe that tree search (a solitaire game without adversaries) can be used to provide a rough initial control policy which can provide valuable information to the designer. If the design requirements are especially demanding, a fast approximate solution can be of benefit as an indication of what proven control techniques would be best applied. For instance, a designer might be able to use a straightforward simulation of a complex system (without need for difficult abstraction) to derive an approximately optimal control policy. From analysis of the approximate control policy, the designer might gain quick insight into the dynamics of the system, such as state space regions that exhibit significant nonlinearities.

Simulation is already a valuable tool in controller design validation. By providing intelligent means to perform directed simulation, we hope these techniques will find their place as powerful tools for control engineers.

1.5 Reading Guide

This dissertation assumes that the reader has an undergraduate-level background in Computer Science, and has introductory-level knowledge of the following areas:

- **Global Optimization** - A good, brief introduction to the area can be found in [38, Chapter 10]. [19, 39, 40] provide a more thorough survey of modern methods.

- **Game-Tree and Tree Search** - A good introduction to this area can be found in [41, Chapters 3–5]. In addition, the reader may want to read the relevant article on recursive best-first search[25].

- **Cell-Mapping Methods** - The most basic ideas of [20] are sufficient to understand Chapter 3.

If the reader is interested in a particular chapter, dependencies between chapters are shown in Figure 1.1.

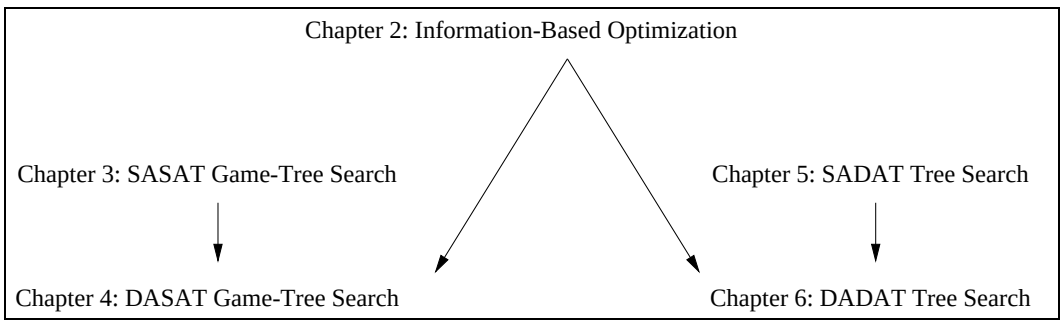

Figure 1.1: Chapter Dependencies

Chapter 2

Heuristic Optimization for Initial Safety Refutation

2.1 Introduction

Given a simulated hybrid dynamical system S, a set of possible initial states I, and a set of "unsafe" states U, we wish to verify nonexistence of an S-trajectory from I to U within t_{max} time units. We call this the *initial safety problem*. Suppose we are given an approximate measure of the relative safety of a trajectory. More specifically, let f be a function taking an initial state i as input, and evaluating the S trajectory from i such that $f(i) = 0$ if and only if the S-trajectory from i enters U within t_{max} time units, and $f(i) > 0$ otherwise. Then verification of the initial safety problem can be transformed into the global optimization (GO) problem:

$$\min_{i \in I}(f(i)) \stackrel{?}{>} 0$$

GO methods may therefore terminate when i is found such that $f(i) = 0$. Given that f does not generally have an analytic form, we do not assume the availability of derivatives. Since each evaluation of f may require a computationally expensive simulation, we are particularly interested in GO methods which perform relatively few evaluations of f. In this context, we compare several original variants of Simulated

Annealing (SA) and Multi Level Single Linkage (MLSL) methods and assess their suitability for our purposes. We discuss the use of knowledge of f gained in the course of GO, and consider the extent to which some GO methods assume special properties of the local optimization (LO) procedures they use.

Finally, we introduce the first multidimensional extension of information-based optimization and show global and local applications of information-based optimization in our multi-level local optimization architecture. These latter contributions are shown to be both (1) competitive with evaluation counts of prominent global optimization techniques, and (2) the most reliable means of finding rare failure scenarios for the motivating problem described in the next section.

2.2 Stepper Motor Stall Problem

Our research was largely motivated by the following safety verification task: Given bounds on the system parameters of a stepper motor (e.g. viscous friction, inertial load), bounds on initial conditions (e.g. angular displacement and velocity), and an open-loop motor acceleration control, verify that no scenario exists in which the motor stalls. We model the motor's continuous dynamics using ODEs given in [26]:

$$\begin{aligned}
\dot{\theta} &= \omega \\
\dot{\omega} &= \frac{-i_a N_b \sin(N\theta) + i_b N_b \cos(N\theta) - D\sin(4N\theta) - F_v\omega - F_c\text{sign}(\omega) - F_g}{J_l + J_m} \\
\dot{i_a} &= \frac{V_a - i_a R + \omega N_b \sin(N\theta)}{L} \\
\dot{i_b} &= \frac{V_b - i_b R - \omega N_b \cos(N\theta)}{L}
\end{aligned}$$

where θ and ω are motor shaft angular displacement and velocity, i_a and i_b are coil A and B currents, V_a and V_b are coil A and B voltages, R and L are coil resistance and inductance, N is the number of rotor teeth, N_b is the maximum motor torque per amp, D is the maximum detent torque, F_v is the viscous friction, F_c is the Coulomb friction, F_g is the gravitational torque load, and J_l and J_m are load and motor shaft inertia. For this system we classify a stall as deviation of $\frac{\pi}{N}$ or more radians from the

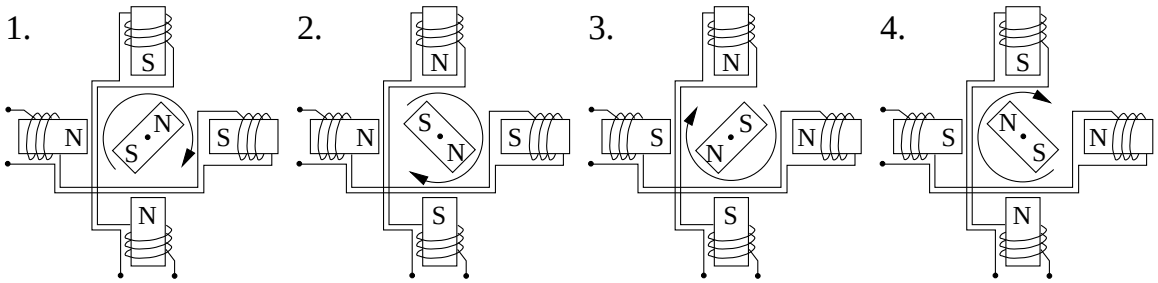

Figure 2.1: Simple Stepper Motor Stepping

255	250	245	240	235	231	226	222	217	213	209	...
205	201	197	193	189	185	182	178	175	171	168	...
164	161	158	155	152	149	146	143	140	137	135	...
132	129	127	124	122	120	117	115	113	110	108	...
106	104	102	100	98	96	94	92	90	89	87	85

Table 2.1: Stepper Motor Acceleration Table

current desired θ equilibrium.

The motor is stepped by reversing polarity of the coil voltages in alternation as shown in Figure 2.1.

Changes to coil voltages occur on such a small time scale that their continuous simulation is judged unnecessary for modeling dynamics relevant to the verification task. Voltage changes were therefore approximated as discrete events. Our acceleration control is open-loop: At fixed intervals the motor is stepped according to an acceleration table. The acceleration table is represented as a sequence of delays between each motor step. Each delay is measured in controller "ticks" where 1 tick = 2.9834e-5 sec. The acceleration table is shown in Table 2.1.

HyTech[15, 16] is a model checker for linear hybrid systems. To be more precise, it proves safety of "geometrically linear" hybrid systems as opposed to "algebraically linear" hybrid systems. Geometrically linear hybrid systems have constant continuous variable derivatives. Thus, the set of reachable states can be computed as a set of convex polyhedra using techniques from computational geometry. Algebraically linear hybrid systems have ODEs which can be expressed in a linear algebraic form.

In [17], Henzinger, Ho, and Wong-Toi suggest two approaches for creating linear approximations of nonlinear hybrid systems: a clock translation and a rate translation. HyTech makes use of an automaton representation of a linear hybrid system. As one increases the accuracy of the linear approximation, both clock and rate translations explode the size of the automaton representation exponentially. An approximation of our stepper motor system either (1) has too large a representation for the computational complexity of the underlying computational geometric algorithms of HyTech, or (2) is too inaccurate such that a conservative approximation that bounds actual system behavior will always yield an "unsafe" verdict over the course of a long stepper motor simulation.

So we first note that there is no apparent approximation of our system for the tools that are currently available. Next, we note that our verification is concerned with a fixed initial time interval (i.e. during acceleration) and is therefore an initial safety problem. Finally, we note that we can compute minimum angular displacement from a stall state over all simulation states as a simple heuristic to numerically rate the relative safety of safe trajectories. We can now ask, "For all possible system parameters and initial states, are all simulation trajectories rated safe?" Put another way, "Is the minimum heuristic evaluation of all possible simulations greater than zero?" If we can answer this optimization question positively, we have verified safety of our hybrid system.

One could argue that such optimization is *not* verification, that one cannot exhaustively simulate all possibilities and can therefore have no guarantees. One can only use such optimization for refutation. To this, we offer two responses: First, if one has additional knowledge of characteristics of one's heuristic evaluation function (e.g. Lipschitz conditions), then an intelligent optimization approach can utilize such characteristics to guarantee a strictly positive minimum with sufficient evaluation (e.g. of a global solution set for a Lipschitzian global optimization problem[36]). The key is to provide a heuristic evaluation that induces a helpful search landscape without itself become overly burdensome computationally. Second, if one has no such knowledge about the heuristic, the absence of verification techniques well-suited to non-trivial dynamics leaves good global optimization as the best assurance. Our desire is to

develop an information-based GO method which, when halted without finding an unsafe trajectory, provides some measure of the thoroughness of its search.

This said, we have endeavored to study a number of representative global optimization techniques in order to assess their suitability to our purpose and point the way towards future innovation.

2.3 Algorithms and test problems

In this section, we describe the global optimization (GO) algorithms used in this study, the local optimization procedures used by them, and the test functions to be minimized. Author-supplied default settings were used for GO algorithms when possible. Otherwise, reasonable parameters were held constant throughout testing. Since our goal is to perform a computationally expensive optimization, we would desire an algorithm which reliably and efficiently gives the desired result without tuning. Experienced users of such algorithms applying problem- and domain-specific knowledge to the choice of options and parameters could expect to yield better results.

The first set of algorithms we consider are variants of simulated annealing (SA) [29, 22]. SA algorithms are theoretically guaranteed to find the global minimum of a function provided that the annealing schedule starts with sufficiently high temperature and cools sufficiently slowly. However, this guarantee comes at great expense in terms of function evaluations. Finding a suitable annealing schedule which balances the tradeoff of reliability versus efficiency is key to the practicality of SA for our purposes.

`AMEBSA` [38, pp. 451-455] performs SA by modifying a downhill simplex method [38, pp. 408-412] such that actual function values of simplex points and possible replacement points are perturbed according to the temperature parameter when making move decisions. Since `AMEBSA` has no default annealing schedule, we have chosen to use the one supplied in the authors' example [37, pp. 182-184]. `ASA`[1] [21], "adaptive simulated annealing", is a SA variant that relies on randomly importance-sampling

[1] `ASA` software developed by Lester Ingber and other contributors is available at URL `http://www.ingber.com/` or `ftp://ftp.ingber.com`.

the search space and adapts separate annealing schedules for each parameter. The automatic adaptation of the annealing schedule trades off reliability for efficiency. `SALO` [10] seeks to combine the theoretical guarantees of SA with the efficiency of local optimization (LO). `SALO` on f is SA on f', where f' is f transformed by LO. At each point that SA evaluates, LO takes place and the value of the local minimum is returned. This is intended to "flatten" f and speed convergence to the global minimum. In both implementations described here and in [10], `ASA` is used as the SA method. In doing so, we again tradeoff reliability for efficiency. When each of these SA methods halts unsuccessfully, it is restarted from the lowest point found thus far.

The second set of algorithms we consider are variants of Multi Level Single Linkage (MLSL) [40]. MLSL uniformly, iteratively samples the search space and performs LO selectively. For each iteration, a new batch of points is evaluated. For each point sampled, LO takes place if there exists no lower sampled point within a critical distance.[2]. `MLSL1` is the original algorithm[40]. `MLSLD` is our variant of `MLSL1` which assumes that the LO procedure is deterministic and should therefore never be repeated from the same sampled point. `MLSLO` is another variant of ours that orders optimizations for each iteration by ascending function value of sampled points. `MLSLOD` has both variations. Our fourth variant, `MLSLSA`, alternates iterations of `MLSLOD` with runs of `ASA`, using the current minimum as the initial point for `ASA`. `LMLSL` is our variant of `MLSL1` which performs "lazy" function evaluation. That is, the function value of a point is only evaluated when it becomes necessary. This avoids the relatively large initial cost when optimizing simple functions. $\texttt{LMLSL}_\epsilon$ is `LMLSL` using an ϵ-descent LO procedure. An epsilon-descent procedure guarantees that, for a step greater than ϵ, the function values at epsilon intervals are sequentially descending.

`RANDLO` simply performs random local optimizations and is intended to provide a baseline for understanding how well LO knowledge is used by `SALO` and MLSL methods. `MONTE` is a Monte Carlo method, the weakest method of those we consider.

We next describe the local optimization procedures used by some of these global optimization algorithms. `FMINU` and `CONSTR` are MATLAB™ optimization functions [13].

[2] We used the critical distance parameter $\zeta = 2$ with 100 points generated per iteration.

AMEBSA	SA simplex method
ASA	Adaptive Simulated Annealing
CONSTR	Sequential quadratic programming method
FMINU	Quasi-Newton LO
FMINU$_\epsilon$	FMINU with ϵ-descent LO
LMLSL	MLSL with lazy f evaluation
LMLSL$_\epsilon$	LMLSL with ϵ-descent LO
MLSL	Multi-Level Single Linkage
MLSL1	basic MLSL method
MLSLD	MLSL assuming deterministic f
MLSLO	MLSL with ordering heuristic
MLSLOD	MLSLO + MLSLD
MLSLSA	MLSLOD and SA in succession
MONTE	Monte Carlo method
RANDLO	Random LO
SA	Simulated Annealing
SALO	SA with LO
YURETMIN	Yuret's LO

Table 2.2: Algorithm Quick Reference

`FMINU` performs unconstrained optimization using a quasi-Newton method with a BFGS formula for updating the Hessian matrix approximation. `FMINU`$_\epsilon$ is our ϵ-descent modification of `FMINU`. `CONSTR` performs constrained optimization using a sequential quadratic programming method. We supply search space bounds and no additional constraints. `YURETMIN` is our variant of Yuret's Masters thesis Procedure 4-1 [54, p.33] which allows specification of search space bounds.

A quick reference table for algorithms is given in Table 2.2.

Finally, we reference the objective functions used for comparing the global optimization algorithms. The first part of our study uses functions selected from GO literature and algorithm demonstrations in order to reveal their relative merits. `RAST` is a scaled Rastrigin function [10]. `HUMP` is the six-hump camelback function [6]. `G-P` is the Goldstein-Price function [6]. `GW1` and `GW100` are 6-dimensional Griewank functions with bounds of each dimension $[-1, 1]$ and $[-100, 100]$ respectively [10]. `SWISS`

CMMR	4-D paraboloid with troughs
G-P	Goldstein-Price function
GW1	Griewank function with $[-1, 1]$ bounds
GW100	Griewank function with $[-100, 100]$ bounds
HUMP	6-hump camelback function
RAST	Rastrigin function
STEP1	Stepper motor stall problem function
STEP2	STEP1 logarithmically scaled
SWISS	4-D paraboloid with pits

Table 2.3: Objective Function Quick Reference

is a 4-D paraboloid with a lattice of many circular pits [37]. `CMMR` is a 4-D paraboloid with a grid of deep troughs [8]. `GW100`, `SWISS`, and `CMMR` have many local minima. `RAST` has a moderate number. `HUMP`, `G-P`, and `GW1` have few. `RAST`, `GW100`, `SWISS`, and `CMMR` are generally paraboloid in shape with different local minima "traps". All slope up to the bounds of the search space.

The second part of our study concerns the motivating example for this research. Test function `STEP1` takes as input two parameters (viscous friction and load inertia) of the stepper motor model, simulates acceleration of the motor, and performs a simple heuristic evaluation of the trajectory by computing the minimum distance to a stall state (0 if stalled). Such a heuristic function is often simple to construct. `STEP2` is `STEP1` logarithmically scaled so as to focus on the unsafe region of the parameter space. These functions are shown in Figures 2.2 and 2.3.

A quick reference table for objective functions is given in Table 2.3.

2.4 Results

Our first tests made use of LO procedure `FMINU` where applicable. 100 optimization trials were performed for each objective function with a maximum of 10000 function evaluations permitted per trial. Each objective function was offset (if necessary) to

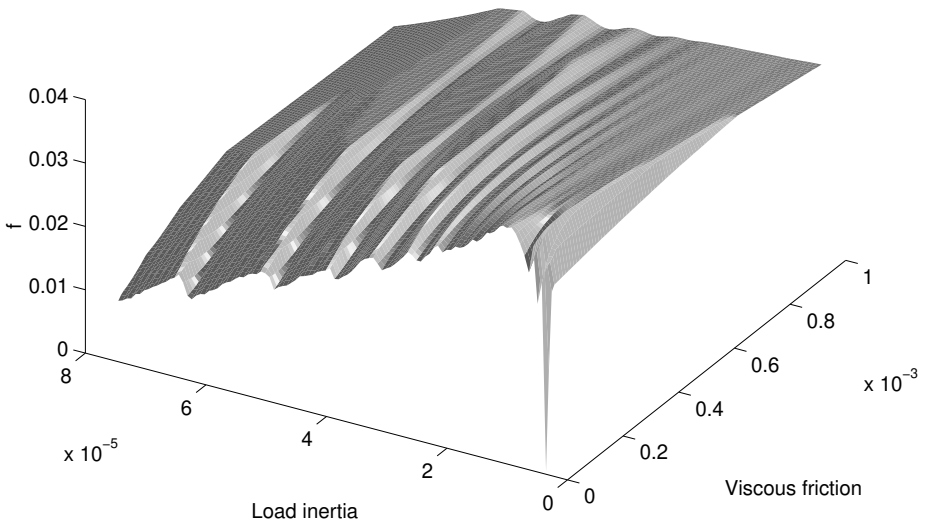

Figure 2.2: Stepper Motor Test Problem `STEP1`

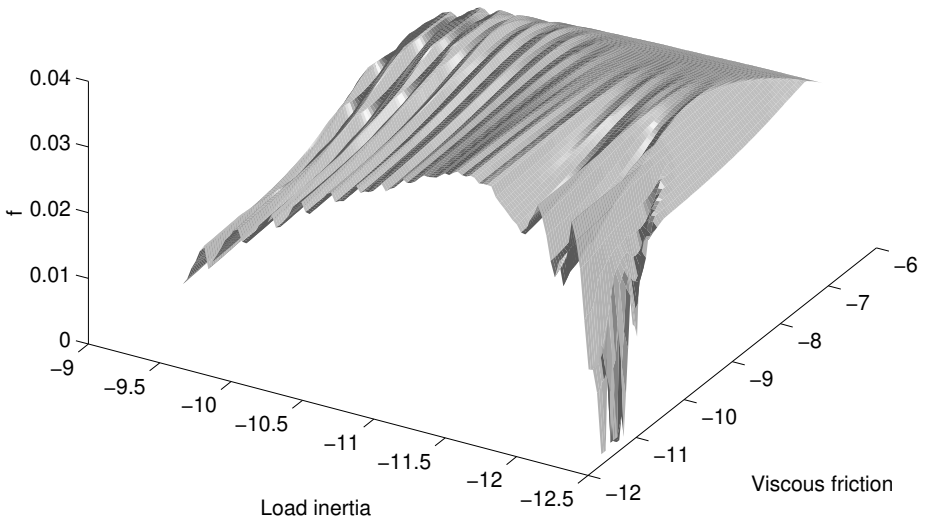

Figure 2.3: Stepper Motor Test Problem `STEP2`

have a global minimum value of 0. A successful trial was one in which the optimization routine found a point with function value less than .001 within 10000 function evaluations. This simulates situations where one is seeking a rare failure case in f.

Each entry in the table of results (Table 2.4) shows the number of successful trials (upper left) and the average number of function evaluations for such trials (lower right).

	RAST	HUMP	G-P	GW1	GW100	SWISS	CMMR
AMEBSA	16 39	**100** **40**	90 222	100 86	0 N/A	100 1340	2 5674
ASA	**100** **404**	100 225	100 1042	100 197	2 6003	100 903	**100** **3756**
SALO	100 585	100 65	100 97	**100** **85**	**95** **4501**	100 163	0 N/A
MLSL1	100 872	100 154	100 170	100 185	47 4315	100 239	0 N/A
MLSLD	100 636	100 154	100 170	100 185	60 4492	100 238	0 N/A
MLSLOD	100 556	100 130	100 132	100 173	52 4370	100 253	0 N/A
MLSLSA	100 544	100 131	100 130	100 174	22 2609	100 254	99 5019
LMLSL	100 847	100 105	100 118	100 96	50 4508	100 187	0 N/A
LMLSL$_\epsilon$	100 638	100 96	100 109	100 93	53 3864	100 192	0 N/A
RANDLO	100 706	100 70	**100** **96**	**100** **85**	58 4008	**100** **146**	0 N/A

Table 2.4: Successful global optimization trials and average function evaluations

Giving the best performance in nearly half of the tests, RANDLO performed surprisingly well, especially for SWISS which has a 4-D lattice of numerous "traps". As RANDLO's LO procedure, FMINU is clearly rarely caught in such traps. Since both trap and non-trap regions are paraboloid surfaces, they effectively "point" to the global minimum for LO procedures such as FMINU. The simple but important observation here is that local optimization does not necessarily find the *nearest* local optimum. We next observe that both SALO and MLSL each rely somewhat on *nearness* of LO. We will later turn our attention to the relationship between the global and local layers of each. FMINU, which assumes f is continuous, behaved understandably poorly for

highly discontinuous `CMMR`. Thus all methods dependent entirely on LO failed all `CMMR` trials. Given that the characteristics of f may not be well understood, this means that a less efficient LO procedure making fewer assumptions would likely be better suited to our purposes.

`SALO` yielded performance similar to that of `RANDLO` where few LOs sufficed and significantly better where more local optima trapped LO (e.g. in `RAST` and `GW100`). At the heart of `SALO`'s design is the following intention: "SA helps in locating good regions of the search space, while the local optimizer is used to rapidly hit the optimum."[10] It is clear from this comparison that `SALO` does indeed successfully apply SA on f' to find good regions of f. When comparing `ASA` with `SALO`, it also appears that the cost for transforming $f \overset{\texttt{FMINU}}{\rightarrow} f'$ is usually more than compensated for by the efficiency gained.

`SALO` was designed with hope that f' would be a "simpler" surface than f, reflecting the function value of the nearest optimum. Interestingly, the designers' experiments utilized Yuret's LO procedure which has short term memory and takes increasingly greater steps downhill as success allows. Such a LO procedure can possibly pass over nearest local minima as step size becomes large. Also Yuret's procedure, being stochastic, does not simply transform one surface to another. Nevertheless, their experiments and ours indicate that `ASA` is able to handle such LO output gracefully in the long run. The fact that `SALO` outperforms `RANDLO` for harder optimization problems is specifically a property of SA and more generally a form of learning. One can view the changing state probability distribution of SA as a gradual accumulation of knowledge about the location of the global minimum. While such learning is effective given a suitable annealing schedule, it is also weak. Heavily traversed local minima may be heavily traversed again. All but one of the function evaluations made in LO are ignored. Much information is wasted. Nonetheless, `SALO`'s performance was impressive.

Performance of MLSL methods, though similar to that of `RANDLO`, yields little to commend them over `RANDLO`. That selective uniform random LO should perform worse than unselective uniform random LO suggests an assumption in MLSL which is not met in our study. Following the analysis more closely in both [39] and [40], we see that

MLSL's LO procedure is assumed to be an ϵ-*descent procedure* such that the current critical distance effectively bounds the step size of LO.[3] We therefore modified `FMINU` to be an ϵ-descent procedure and tested $\texttt{LMLSL}_\epsilon$ for comparison. Although $\texttt{LMLSL}_\epsilon$ is somewhat of an improvement over `LMLSL`, it is still generally worse than `RANDLO`. ϵ-descent does not therefore appear to help us much. We conjecture that MLSL methods dominate `RANDLO` for objective functions where LO is trapped in many minima, and that `SALO` dominates MLSL methods for such objective functions in our study because our f'-surfaces are easily globally optimized with LO. To elucidate the latter point, consider `RAST`, `GW100`, and `SWISS`. LO roughly transforms each into a paraboloid of plateaus. LO of such LO-transformed functions can then efficiently lead to the global optimum. We can view the task of global optimization as *multi-level local optimization*. The base-level LO_0 takes advantage of whatever information about f is available (continuity, gradients, etc.), the next level LO_1 is suited to the class of one's LO_0-transformed function f', and so on. We may stop after arbitrarily many (probably 2-3) LO levels and perform global optimization at the top level. The role of each LO level is to enlarge the regions leading to global optima. Multi-Level local optimization methods we have developed are presented in Section 2.7.3.

Regarding MLSL methods, let us also note that, like `SALO`, they all but ignore information gained through LO. Uniformly sampled points are locally optimized based only on the values of sampled points within a critical distance. Again we find great waste of information gained at great expense.

`AMEBSA` gave mixed results which can likely be attributed to the lack of annealing schedule tuning. Perhaps an adaptive annealing schedule would make `AMEBSA` more suitable for such problems. `ASA`'s efficiency was unpredictable, although it was perhaps the most reliable method for this set of objective functions.

While these functions may give a general indication of the relative strengths of these methods without tuning, the functions share a common property undesirable for our purposes: The *unconstrained* global minimum is never located at or beyond the bounds of the search space. Therefore, our optimization methods need not perform

[3] This is nowhere mentioned in survey [3] and is not emphasized elsewhere in the literature.

	STEP1	STEP2
ASA	0 N/A	2 497
SALO	10 80	5 202
MLSLOD	10 127	10 191
LMLSL	10 163	**10** **137**
RANDLO	**10** **78**	10 359
MONTE	0 N/A	6 469

(a) CONSTR

	STEP1	STEP2
ASA	0 N/A	2 497
SALO	7 387	9 198
MLSLO	4 790	10 231
LMLSL	3 389	**10** **169**
RANDLO	**9** **501**	10 172
MONTE	0 N/A	6 469

(b) YURETMIN

Table 2.5: Results for STEP1 and STEP2

well along the bounds of our search space. It is for this reason that unconstrained FMINU was suitable for use with such global optimizations. We used this as an opportunity to try two constrained LO procedures CONSTR and YURETMIN for the stepper motor test problems STEP1 and STEP2. For this testing, we performed 10 trials to find a function value of 0 with a maximum of 1000 function evaluations per trial. The results appear in the tables of Table 2.5.

Since both STEP1 and STEP2 have a small number of local minima along the bounds of the search space, behavior of LO again figured most significantly in our results. Despite the fact that much of the search space slopes downward away from the corner where failures occur, CONSTR had a bias towards looking in that particular corner. It was thought that STEP2 (log-log scaled STEP1) would be an easier function to optimize, but this was not the case. Not only was the global minimum basin expanded, but nearby local minima also expanded, trapping LO more often.

ASA's function evaluation expenses were such that it was outperformed by MONTE. The remaining LO-based methods performed similarly overall. The cost of computing simple heuristic information about relative safety of trajectories is usually more than compensated for by efficiency in discovering unsafe trajectories through optimization. For both LO procedures, RANDLO gave best performance for STEP1 and LMLSL gave

best performance for `STEP2`. Although there was no universal "winner" among global optimization procedures, it is encouraging to note that procedures such as `SALO` and `LMLSL` could be run in parallel to achieve respectable, more reliable results. The choice of LO procedure proved very significant for performance, which again underscores the importance of developing robust, efficient LO procedures suited to large classes of functions.

2.5 Conclusions of Comparative Study

While no global optimization procedure was generally dominant in our comparative study, random local optimization seemed best suited for objective functions with few minima, and `SALO` with `ASA` seemed best suited for objective functions with many minima. By making use of `ASA` for SA, one both avoids the need to specify an annealing schedule and benefits from its relative efficiency among SA algorithms. Although one is encouraged to make use of `ASA`'s options to improve performance, we have not done so and have been pleased with most results nevertheless.

`SALO` and MLSL methods perform global optimization with global and local search phases, and rely on local optimization for efficiency. However, both methods make little or no use of information gained in the course of local optimization. We believe that great progress will be made in global optimization when global optimization and local optimization are seamlessly integrated to share knowledge gained of f. Where evaluation of f is computationally expensive, it is worth computational expense to utilize such knowledge for the efficiency of global optimization. To this end, we have developed a set of information-based optimization techniques where each optimization step is chosen with respect to the information gained thus far.

2.6 Information-based global optimization

In this section, we look at a particular class of global optimization techniques which are suited to specific characteristics of our problem. We describe previous information approaches to optimization, and present our own specialization of such techniques for

initial safety refutation.

From the previous comparative study, we noted that most global optimization methods throw away most of the information gained in the course of optimization. For our purposes, each evaluation of f requires a simulation and an evaluation of that simulation which may be computationally expensive, so we are particularly motivated to make good use of such information in order to reduce the function evaluations needed.

One approach is to characterize properties of the set of functions one wishes to optimize and to use such information to construct an optimal decision procedure for optimization. In the course of optimization, we use our current set of function evaluations to decide on the next best point to evaluate with respect to our function set. Such is the strategy of *Bayesian* or *information* approaches to global optimization[30, 31, 44, 49], which have optimal average-case behavior over the set of functions for which each is designed.

2.6.1 Strongin's Information Approach

The *information approach* to optimization was proposed by Roman Strongin in [47, 48, (in Russian)]. The first English publication of this work can be found in [49]. Most optimization techniques rely on some form of assumptions of objective function properties. Some techniques assume a function is Lipschitzian in order to bound solutions. Others assume the function is nearly parabolic near minima in order to claim quadratic convergence. Rather than rely on a restrictive constraint language to define properties of the functions of interest, Strongin sought to instead use a probability measure on the class of functions under consideration. Each step of his information approach to global optimization consists of a *maximum likelihood estimation* based on the results of previous iterations.

In [49], Strongin derives an implementation of the information approach for a

one-dimensional root-finding problem. Strongin's derivation is based on a probabilistic preference for functions which satisfy a Hölder condition[4] at the root. He also derives an implementation of the information approach for a one-dimensional global optimization problem. The derivation, described as similar to that of the root-finding algorithm, is not given in [49], but rather appears in [47].

In dealing with multidimensional objective functions, Strongin applies his one-dimensional approach through use of volume-filling Peano curves. Simply put, a uniform grid of points in the volume is connected by a single line such that the line comes within a certain distance ϵ of every point in the volume. The successive refinement of a Peano curve in two dimensions is shown in Figure 2.4. One-dimensional optimization is performed on this line as an approximation of the multidimensional global optimization problem. The problem with this approach is that a simple, multidimensional, global optimization with one optimum looks like a complex optimization along the Peano curve with local optima increasing with each Peano curve refinement. For a small ϵ, the curve must have such complexity that the corresponding one-dimensional optimization problem becomes needlessly complex. This is the price paid for applying one-dimensional optimization to multidimensional problems. In the next section, we will introduce the first truly multi-dimensional information approach to optimization.

Yaroslav Sergeyev augmented Strongin's information approach to global optimization with local tuning based on change in the local Lipschitz constant[5] of the objective function over different segments of the search region. Sergeyev also recommended application of the method using Peano curves. We implemented Sergeyev's information approach with local tuning and used Peano curves to apply the approaches to multidimensional objective functions of our comparative study. The results were disappointing. Not only was the success of results very sensitive to a reliability parameter r, but sampling irregularities introduced by the Peano curve were clearly visible as sharp sampling density contrasts were observed across quadrant and subquadrant boundaries.

[4] A Hölder condition is a Lipschitz condition $|f(x) - f(y)| \leq A(y)\,|x - y|^{\alpha}$ of order α with Lipschitz constant A.

[5] A local Lipschitz constant is a real number c such that $|f(x) - f(y)| \leq c\,|x - y|$ for all y local to x.

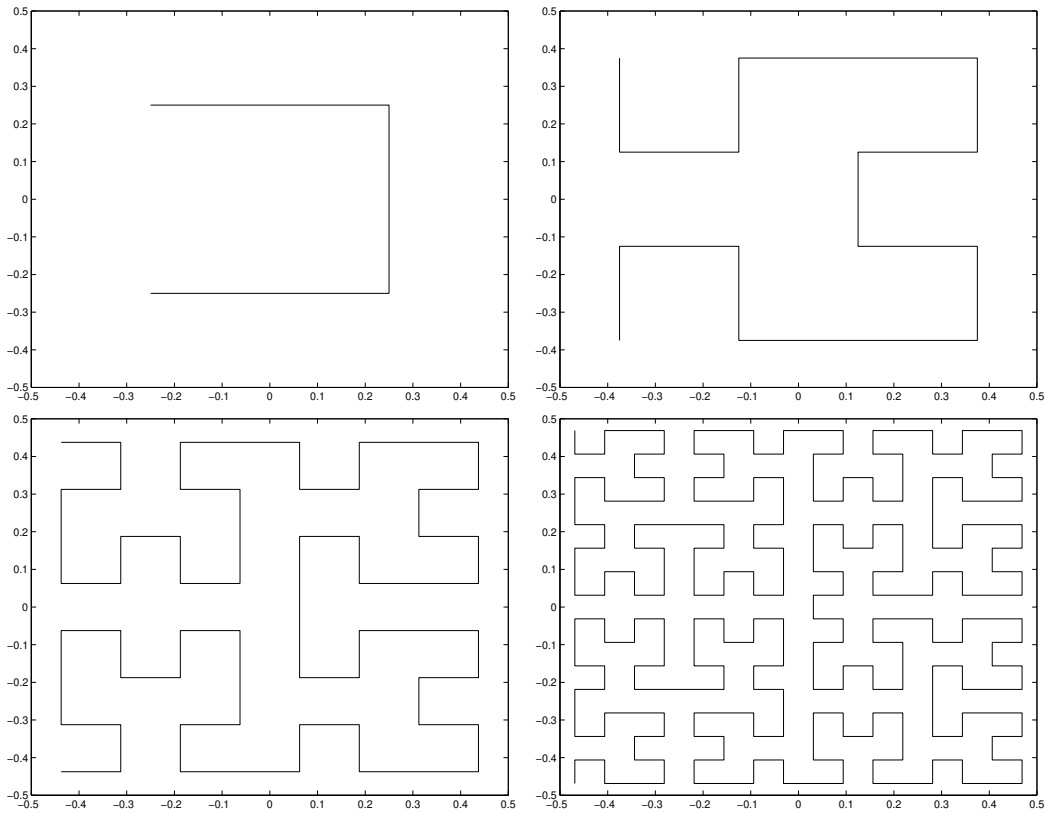

Figure 2.4: Refinement of Peano curve in two dimensions

2.6.2 Information-Based Optimization for Refutation

Our information-based optimization approach to refutation is most strongly influenced by the reasoning behind Strongin's approach. We desired a simple means of characterizing objective functions which (1) gave rise to a computationally simple optimal decision process, and (2) allowed discontinuities in objective functions. Furthermore, the purpose of our optimization is not simply to find the global minimum. Rather, we know we are seeking a zero of a non-negative, real-valued function. Instead of seeking the most likely minimum value, we specifically seek a zero in order to refute initial safety of a hybrid system.

Our approach relies on two main assumptions about the probability measure on the class of functions we consider. The first assumption is that the functions are more often locally continuous than not. This does not preclude discontinuities in functions. A zero is just as likely to occur anywhere in the class of discontinuous functions, so we rely on there being some local continuity for the maximum likelihood approach to be beneficial. As we will see, this approach can be surprisingly robust to discontinuities in the context of multi-level optimization techniques.

Our second assumption is that lower local Lipschitz constants are more likely than higher local Lipschitz constants. The ramification of this likelihood assumption is that zeros are most likely to occur where they require a minimal Lipschitz constant given the sample points evaluated thus far. On a one-dimensional curve, the optimization process is simple. First, both endpoints are evaluated. The next point *most likely* to be a zero will be that which minimizes slope between itself and neighboring (i.e. adjacent) evaluated points along the line. This most likely candidate is evaluated, and the process is repeated until a zero is found or the optimization is terminated. In the next section, we see that there are significant difficulties to overcome in applying such an approach in more than one dimension.

2.7 Multi-Dimensional, Multi-Level Information-Based Optimization

Previous information-based methods have been limited to global optimization in one dimension. In this section, we introduce two new information-based optimization methods for multidimensional problems. We first introduce the decision procedure used by these methods, thus explicating the class of functions for which the decision procedure is biased. Next we discuss the use of *multi-level local optimization* for speeding convergence. Finally, we introduce the information-based optimization algorithms themselves.

2.7.1 Decision procedure

At each iteration i of our algorithm, we wish to evaluate our heuristic function f at the location x_i for which $f(x_i) = 0$ is most likely to occur. We base our notion of likelihood on characteristics of a class of functions to which f belongs. Our decision procedure is then based on some decision ranking function g_i which computes a ranking corresponding to the relative likelihood of a zero occurring at an unevaluated point x_i given previous f-evaluations at $x_1, x_2, \ldots, x_{i-1}$:

$$g_i(x_i) \stackrel{\text{def}}{=} g(x_1, x_2, \ldots, x_{i-1}, x_i)$$

So for each iteration i, we could globally optimize g_i to choose the next x for which f is evaluated. However, a reliable global optimization of g for each iteration of a global optimization of f is not only computationally prohibitive, but increasingly very difficult as well. We instead desire to *approximate* an optimal decision with respect to our assumptions about f, and we do so by uniformly, randomly sampling g, returning the optimum of the samples. We call this `DECISION1` (Algorithm 1). The computational complexity of this decision procedure grows as the computational complexity of evaluating g_i (which we will see is $O(i^2)$).

Algorithm 1 Sampling information-based optimization decision function

DECISION1(L, $lBound$, $uBound$)
▷ **Input:** a list of {x,f(x)} pairs,
the lower bounding corner of the search space, and
the upper bounding corner of the search space
▷ **Output:** minimum point
$min_gx \leftarrow \infty$
for $i \leftarrow 1$ **to** $maxPts$ **do**
$x \leftarrow$ uniformly random vector in space bounded by $lBound$ and $uBound$
$gx \leftarrow \mathrm{g}(L, x)$
if $(gx < min_gx)$ **then**
$min_gx \leftarrow gx$
$min_x \leftarrow x$
return min_x

In order to construct g, we must make some assumptions over f's class of functions with regard to where we would most expect to find zeros. One assumption we make is that f is continuous[6]. Another assumption concerns flatness and smoothness preferences: Given a set of points and their f-evaluations, a zero is more likely to occur where it demands less slope between itself and previous points.

A first attempt at constructing g_i might be to create a function which returns

$$g_i(x) = \max_{j=1}^{i-1} \frac{f(x_j)}{||x_j - x||}.$$

That is, we could rank the likelihood of $f(x) = 0$ by computing the maximum slope between the hypothetical zero at x and other points we have already evaluated. The lesser the g-value, the more likely a zero f-value. The global minimum of g would then be the optimal point at which to next evaluate f given previous f evaluations.

Consider Figure 2.5.

Suppose we have evaluated the curve at points a, b, and c and are using such a g as our decision ranking function. Intuitively, we would want g to return point d as the next best point to evaluate. However, the slope between a and d will make d a less preferable decision point than one to the right of d for which a zero would have

[6]This is not a trivial assumption for our general application, of course. Our stepper motor system trajectories are continuous in the initial condition. Such continuity is preserved in our choice of f.

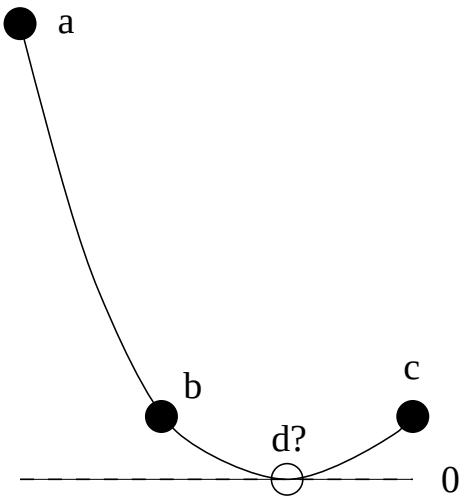

Figure 2.5: Shadowing example

equal slopes to a and c for this simple function. We would like instead for point b to "shadow" point d from point a. Our simple attempt to do so is shown as Algorithm 2. A point a is "shadowed" from point d by point b for function g if $||d-b|| < ||d-a||$ and $|g(a)-g(b)|/||a-b|| > |g(a)-g(d)|/||a-d||$. That is, a is shadowed by b if b is closer to d than a and the slope between a and b on g is greater than the slope between a and d on g.

The average-case optimality of the information-based approach relies on maximum likelihood assumptions over a class of objective functions. One of these assumptions is a greater likelihood for lesser local Lipschitz constants. In one dimension, local Lipschitz constants are computed with respect to the adjacent previously evaluated points along the curve. In more than one dimension, we must define "local". If we include all previously evaluated points in the computation of local Lipschitz constants, then "local" really means "global" over the entire search space. In evaluating candidate points with the shadowing approach, we restrict our attention to non-shadowed evaluated points as we compute local Lipschitz constants. If, for any candidate point, lower Lipschitz constants are more likely between a zero at that point and non-shadowed evaluated points, then our approach retains average-case optimality. Shadowing is a heuristic approach to relevance, and is helpful to the extent that it more accurately reflects maximum likelihood of zeros for problems of interest.

Algorithm 2 g, the decision procedure function to be optimized

```
G(L, x)
▷ Input: a list of {x, f(x)} pairs,
         the current decision point being evaluated
▷ Output: ranking of likelihood that x is a zero
for i ← 1 to length(L) do
    dx[i] ← ||x − first(L[i])||
sort dx in ascending order and permute L accordingly
maxSlope ← 0
for i ← 1 to length(L) do
    slope ← second(L[i])/dx[i]
    if (slope > maxSlope) then
        newMaxSlope ← true
        for j ← 1 to i − 1 do
            otherSlope ← |second(L[i]) − second(L[j])|/||first(L[i]) − first(L[j])||
            ▷ Note: This otherSlope information may be cached.
            if (otherSlope > slope) then
                newMaxSlope ← false
                break from for loop (j)
        if (newMaxSlope) then
            maxSlope ← slope
return maxSlope
```

2.7.2 Multi-Level Local Optimization

One might then construct the simple information-based global optimization procedure given in Algorithm 3.

However, we note that one ramification of random sampling in our decision procedure is that we do not achieve efficient convergence. This is illustrated in Figure 2.6, which shows an information-based global optimization of a two-dimensional circular paraboloid with a zero at the origin. From the initial random point in the lower left corner, the procedure then checks points in the upper right, lower right, upper left, and just left of the global minimum at the center. The cluster of 25 points that follows gradually expands towards the center from the fifth point. In practice, where failures do not occur in miniscule regions, this slow convergence is not a problem. However, we also note that our decision procedure will have to deal with the computational burden of small dense clusters of points which are not very informative globally. We

Algorithm 3 Simple information-based global optimization

```
INFO-BASED-OPT(lBound, uBound)
▷ Input: the lower bounding corner of the search space, and
         the upper bounding corner of the search space
H ← {}
newx ← random point in search space
fx ← f(newx)
if (fx = 0) then
    terminate with success
H ← append(H, {newx, fx})
while (true) do
    newx1 ← DECISION1(H, lBound, uBound)
    fx ← f(newx)
    if (fx = 0) then
        terminate with success
    H ← append(H, {newx, fx})
```

may wish instead to apply a rapidly convergent local optimization procedure and pay attention only to the first and last points of such an optimization.

In our previous comparative study, we note that this is a common approach among the most successful methods of the study. A global search phase makes use of a local optimization subroutine so that the global phase is, in effect, searching $f'(x_1) \stackrel{\text{def}}{=} f(x_2)$ where $\{x_2, f_{min}\} = \texttt{LO}(f, x_1)$, where `LO` is a local optimization procedure. In SALO [10] (simulated annealing atop local optimization), for each point evaluation in the global phase, a local optimization takes place and the function value of the local minimum is associated with the original point. The effect can be roughly described as a "flattening" of a search space into many plateaux (with plateaux corresponding to local minimum values). This search paradigm may be generalized to arbitrary levels where each level performs some optimizing transformation of its search landscape to create a "simpler" one for the level above. Obviously, the work done to simplify should be more than compensated for by the reduced search effort for the level above. The top level performs a global optimization, and all lower levels perform local optimization. We call this paradigm *Multi-Level Local Optimization* (MLLO). We assert that information-based optimization is particularly well-suited to optimizing coarsely plateaued search landscapes. Now let us consider two information-based

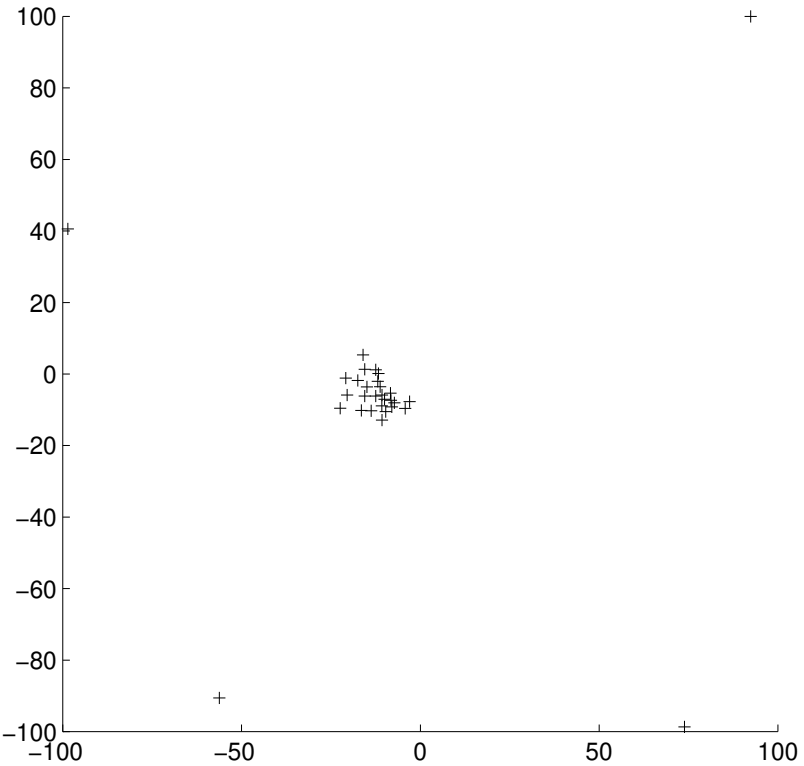

Figure 2.6: Information-based global optimization of 2-D circular paraboloid

applications of MLLO.

2.7.3 MLLO-IQ and MLLO-RIQ

`MLLO-IQ` (Algorithm 4) is a 2-level MLLO with a simple information-based approach (Algorithm 3) atop quasi-Newton local optimization. With each iteration, `MLLO-IQ` chooses a point x_1, locally optimizes f from x_1 to x_2, and associates $f(x_2)$ with both x_1 and x_2 in order to "plateau" the space. In doing so, we limit the number of function values involved in decision making. Still, we may wish to further limit such growth in computational complexity. By limiting our information-based search to a hypersphere containing a maximum limit of previously evaluated points, we limit the complexity to a constant. Such is the approach taken in `MLLO-RIQ`.

`MLLO-RIQ` (Algorithm 5) begins with a locally minimized random point and a

Algorithm 4 MLLO-IQ

MLLO-IQ($lBound$, $uBound$)
▷ **Input:** the lower bounding corner of the search space, and
the upper bounding corner of the search space
$H \leftarrow \{\}$
$newx1 \leftarrow$ random point in search space
$\{newx2, fx\} \leftarrow \text{LO}(f, newx1)$
if $(fx = 0)$ **then**
terminate with success
$H \leftarrow$ concatenate(H, $\{\{newx1, fx\}, \{newx2, fx\}\}$)
while (**true**) **do**
$newx1 \leftarrow$ DECISION1(H, $lBound$, $uBound$)
$\{newx2, fx\} \leftarrow \text{LO}(f, newx1)$
if $(fx = 0)$ **then**
terminate with success
$H \leftarrow$ concatenate(H, $\{\{newx1, fx\}, \{newx2, fx\}\}$)

maximum search radius. Together these define our initial hypersphere. With each iteration, a decision procedure (`DECISION2`) finds an approximately optimal next point to locally optimize within this hypersphere. If the new point has a lesser function value than the center, it becomes the new center and the distance between the two points becomes the new hypersphere radius. If too many points are being considered in `DECISION2`, a lesser amount of points closest to center are retained and the search radius is adjusted. This information-based local optimization terminates when the number of times the center minimum is found by local optimization exceeds a threshold. Then the process repeats with a new random point. Thus we perform a random search of information-based local optimizations of quasi-Newton local optimizations.

2.8 Experimental results

We now compare our information-based approaches to those considered in our previous comparative study. Our first tests all made use of the same quasi-Newton local optimization method where applicable. As before, 100 optimization trials were performed for each objective function with a maximum of 10000 function evaluations

Algorithm 5 MLLO-RIQ

```
MLLO-RIQ(lBound, uBound, maxRadius)
▷ Input: the lower bounding corner of the search space,
         the upper bounding corner of the search space, and
         maximum radius of local hypersphere search
H ← {}
radius ← maxRadius
while (true) do
    x ← random point in search space
    {center, centerVal} ← LO(f, x)
    if (centerVal = 0) then
        terminate with success
    H ← concatenate(H, {{x, centerVal}, {center, centerVal}})
    sort pairs in H in ascending order of ||first(pair) − center||
    H' ← up to first minPts pairs of H
    centerHits ← 0
    while (centerHits > maxCenterHits) do
        recenter ← false
        newx1 ← DECISION2(H', center, radius)
        {newx2, fx} ← LO(f, newx1)
        if (fx = 0) then
            terminate with success
        if (||newx2 − center|| < tolerance1) then
            centerHits ← centerHits + 1
        if (centerVal − fx > tolerance2) then
            radius ← min(maxRadius, ||newx2 − center||)
            center ← newx2
            centerVal ← fx
            centerHits ← 0
            recenter ← true
        H ← concatenate(H, {{newx1, fx}, {newx2, fx}})
        H' ← concatenate(H, {{newx1, fx}, {newx2, fx}})
        if (length(H') > maxPts) then
            recenter ← true
        if (recenter) then
            sort pairs in H in ascending order of ||first(pair) − center||
            H' ← up to first minPts pairs of H
```

permitted per trial. Each entry in the table of results (Table 2.6) shows the number of successful trials (upper left) and the average number of function evaluations for such trials (lower right).

	RAST	HUMP	G-P	GW1	GW100	SWISS
AMEBSA	16 39	**100** **40**	90 222	100 86	0 N/A	100 1340
ASA	100 404	100 225	100 1042	100 197	2 6003	100 903
SALO	100 585	100 65	100 97	100 85	**95** **4501**	100 163
LMLSL	100 847	100 105	100 118	100 96	50 4508	100 187
RANDLO	100 706	100 70	100 96	100 85	58 4008	**100** **146**
MLLO-IQ	100 286	100 71	100 97	**100** **83**	57 4493	100 157
MLLO-RIQ	**100** **161**	100 57	**100** **92**	**100** **83**	46 4536	100 148

Table 2.6: Successful global optimization trials and average function evaluations

Both MLLO-IQ and MLLO-RIQ perform very well in general. What is most instructive from these results are the cases where the strengths and weaknesses of these methods are most prominently displayed. Let us first consider RAST, the Rastrigin function. RAST is a 2-D, sinusoidally-modulated, shallow paraboloid with 49 local minima within the search bounds. The quasi-Newton local optimization layer of MLLO-IQ and MLLO-RIQ effectively transforms this objective function f into f', a shallow paraboloid of plateaux. MLLO-IQ's global information-based search of f' finds the lowest plateau very quickly, and the local information-based search of MLLO-RIQ does a focused descent which leads it to the global minimum with even greater efficiency. This suggests that these searches are particularly well-suited to global optimization of functions with a moderate number of local minima. For functions with fewer local minima (HUMP, G-P, and GW1), there is little to be gained by such extra computation. Random local optimization (RANDLO) will suffice.

Now let us consider the weaknesses of these methods shown in failed cases with GW100. Indeed the performance of these methods is worse than random local optimization. Why? GW100 is a 6-D, sinusoidally-modulated, shallow paraboloid with

about 4×10^7 local minima. For this function, our quasi-Newton local optimization exhibits interesting and unexpected behavior: In all but the lowest points of the surface, local optimization most often leads to local minima that are far from those nearby the initial point. In this example, we are reminded that "local" in "local optimization" refers to properties of the optimum itself and not the "nearness" of the optimum location. Without such nearness, the search landscape is not simply transformed into a landscape of plateaux. Our quasi-Newton local optimization did not optimize to near minima, and so created a landscape which was not suited for information-based global optimization.

`MLLO-RIQ` also has difficulty with `GW100`, but for different reasons. After quickly finding the region containing the global minimum, the method spends much of the remainder of its search effort first searching many points mutually far apart near the boundary of the 6-D hypersphere. Perhaps randomly sampling f or f' within the search hypersphere might encourage convergence. `SALO` remains our best option for functions with a large number of local minima.

While these functions may give a general indication of the relative strengths of these methods (without tuning), the functions share a common property undesirable for our purposes: The *unconstrained* global minimum is never located at or beyond the bounds of the search space. Therefore, our optimization methods need not perform well along the bounds of our search space. It is for this reason that unconstrained quasi-Newton local optimization was suitable for use with such global optimizations. We used this as an opportunity to try two constrained LO procedures `CONSTR` and `YURETMIN` for the stepper motor test problems `STEP1` and `STEP2`. (See Figures 2.2 and 2.3.) For this testing, we performed 10 trials to find a function value of 0 with a maximum of 1000 function evaluations per trial. The results appear in Table 2.7.

These results were very pleasing. `MLLO-IQ` is the first technique we have observed that has succeeded in every `STEP1` and `STEP2` trial. It does so with excellent efficiency as well. Since the decision procedure computation time was also dominated by simulation time, it was also easily the fastest algorithm for these trials. `MLLO-RIQ` did surprisingly well considering that most of the search space of these functions slopes downward and away from the corner of the space where the rare failure cases occur.

	STEP1	STEP2
ASA	0 N/A	2 497
SALO	10 80	5 202
LMLSL	10 163	**10** **137**
RANDLO	10 78	10 359
MONTE	0 N/A	6 469
MLLO-IQ	**10** **46**	10 219
MLLO-RIQ	10 60	8 330

(a) CONSTR

	STEP1	STEP2
ASA	0 N/A	2 497
SALO	7 387	9 198
LMLSL	3 389	10 169
RANDLO	9 501	10 172
MONTE	0 N/A	6 469
MLLO-IQ	**10** **108**	**10** **109**
MLLO-RIQ	8 301	9 239

(b) YURETMIN

Table 2.7: Results for STEP1 and STEP2

2.9 Conclusions

A powerful approach to initial safety verification is to transform the problem into an optimization problem and leverage the power of efficient optimization methods. This is accomplished by

- providing a good heuristic evaluation function f,
- choosing an efficient local optimization procedure well suited to f, and
- applying a global optimization procedure for which one's local optimization procedure is well suited.

While no global optimization procedure in our studies was generally dominant, we note that random local optimization seems best suited for heuristic functions with few minima, SALO[10] seems best suited for heuristic functions with very many local minima, and MLLO-IQ and MLLO-RIQ seem best suited for heuristic functions with a moderate number of local minima. MLLO-IQ is better suited for problems where the global minima are expected to occur at parameter extremes, whereas MLLO-RIQ is

better suited to low-dimensional problems where global minima are found within the space. Our decision procedure approximates an optimal sequence of trials over the class of continuous heuristic functions for which lesser local Lipschitz constants are more likely. Furthermore, we have empirically demonstrated their effective use with functions having many discontinuities in the context of multi-level local optimization.

Finally, we note that the computational effort invested toward efficient optimization should be compensated for by reduced overall runtime. For our problem, the computational expense of our simulation justified such effort. But what of initial safety problems for which simulation requires less runtime? Setting `maxpts` $= 0$ for Algorithm 1[7] yields random local optimization. As `maxpts` $\rightarrow \infty$, our decisions approach optimality and the decision-making effort exceeds the search effort it saves. Where is the happy medium in this tradeoff? In future research, we hope to investigate means of dynamically adjusting the level of strategic effort of such information-based algorithms in order to address a larger class of problems efficiently.

[7] Algorithm 1 is called by Algorithm 3.

Chapter 3

SASAT Game-Tree Search

Extending discrete game-tree search to hybrid system game-tree search introduces two new decisions in optimization: action discretization and action timing discretization. These correspond to the decisions of *how* to act and *when* to act. When a discretization is supplied to the search algorithm, we call it a "static" discretization, i.e. the search algorithm cannot affect the discretization choice. We call such a search a "SASAT Search", as it has both Static Action and Static Action Timing discretizations. A SASAT search is essentially a discrete search applied to a hybrid or piecewise continuous system. Thus, we can benefit directly from AI discrete game-tree search techniques.

In this chapter, we will formally define a SASAT Hybrid System Game and its solitaire case, a SASAT Hybrid System Search Problem. A magnetic levitation control problem is introduced, and we show how the control problem may be posed as a game to achieve robust control. We then examine three ways of using simulation and game-tree search to inform robust control of a magnetic levitation controller. In the first, we present a dynamic-programming approach with an augmented cell-map or game-graph. Next, we discuss current techniques for alpha-beta search (without approximation) and show the similarity of the resulting control policy of both approaches.

Combining the best of both algorithms, we present a synthesis called Game-Graph Alpha-Beta, which has a novel form of caching results of alpha-beta search for future

reuse. This synthesis provides a more efficient means of online hybrid system control for low-dimensional state spaces, assuming that a good discretization can be found. We conclude with a summary and discussion of future directions.

3.1 SASAT Hybrid System Game and Search Problem

Formally, a SASAT Hybrid System Game is defined as a 7-tuple

$$\{S, s_0, A, p, l, m, d\}$$

where

- S is the hybrid state space with a finite number of finite discrete variable domains, and a finite-dimensional continuous space,
- $s_0 \in S$ is the initial state,
- A is the finite discrete action space,
- p is the number of players,
- $l : S \times \{1, \ldots, p\} \rightarrow \{a_1, \ldots, a_n\} \in A$ is a *legal move* function mapping from a state and player number to a finite set of legal actions that may be executed in that state by that player,
- $m : S \times A^p \rightarrow S \times \Re^p$ is a *move* function mapping from a state and simultaneous player actions to a resulting state and the utility of the combined actions for each player,
- $d : S \rightarrow S \times \Re^p$ is a *delay* function mapping from a state to the resulting state and the utility of the trajectory segment for each player. This delay governs the evolution of the system through time between moves.

The total utility of any finite trajectory is computed as the sum of the trajectory move and delay utilities. In this time-invariant formalism, time can easily be encoded in a continuous clock variable, and time invariant behavior could thus be easily achieved.

Although not addressed in this chapter, a *SASAT Hybrid System Search Problem* is a special case of the SASAT Hybrid System Game where we are interested in finding a trajectory from the initial state to a goal state. Usually such problems are stated in terms of path cost rather than utility. Formally, a SASAT Hybrid System Search Problem is defined as a 7-tuple

$$\{S, s_0, S_g, A, l, m, d\}$$

where

- S is a hybrid state space with a finite number of finite discrete variable domains, and a finite-dimensional continuous space,
- $s_0 \in S$ is an initial state,
- $S_g \subset S$ is a set of goal states,
- A is a finite discrete action space,
- $l : S \rightarrow \{a_1, \ldots, a_n\} \in A$ is a *legal move* function mapping from a state to a finite set of legal actions that may be executed in that state,
- $m : S \times A \rightarrow S \times \Re$ is a *move* function mapping from a state and action to a resulting state and cost of the action,
- $d : S \rightarrow S \times \Re^p$ is a *delay* function mapping from a state to the resulting state and the cost of the trajectory segment for each player. This delay governs the evolution of the system through time between moves.

We next describe a SASAT Hybrid System Game in the domain of magnetic levitation.

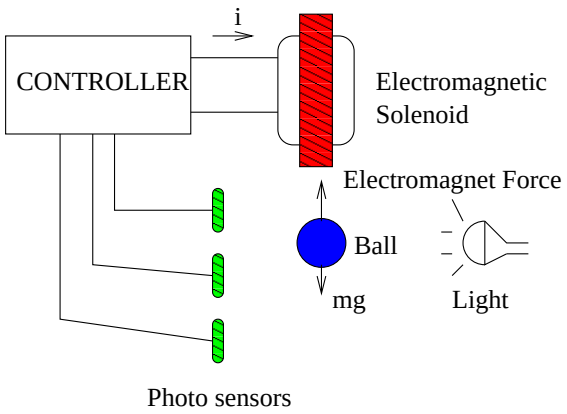

Figure 3.1: Schematic of magnetic levitation system. Courtesy of Feng Zhao: phase-space based magnetic levitation control experiment

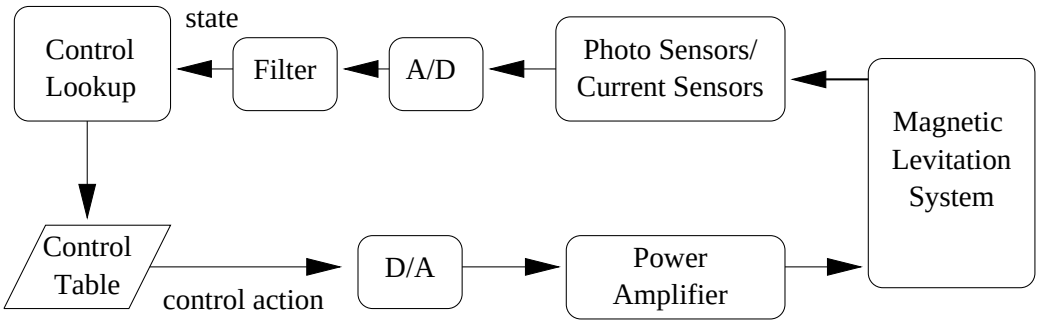

Figure 3.2: Block diagram of magnetic levitation system operation. Courtesy of Feng Zhao: phase-space based magnetic levitation control experiment

3.2 Magnetic Levitation Problem

We seek to use simulation and game-theoretic techniques to design a safe control policy for the magnetic levitation (maglev) system of [55, 28] in which the goal is to suspend a metal ball beneath an electromagnet. This nonlinear, unstable system requires an active controller for stabilization, and is representative of magnetic levitation systems found on high-speed transportation systems such as the German Transrapid system. The schematic for Zhao's maglev system is given in Figure 3.1.

Figure 3.2 shows a block diagram of maglev system operation. The system state is estimated from photosensors and sampled at a rate of about 5000Hz. The controller maps system state to the control power output which affects the electromagnetic

coil current. This in turn affects the system state, so this is a closed loop system. System state includes the distance x and velocity v *from* the electromagnetic solenoid downward to the ball, and the coil current i. The differential equations describing the dynamics of this system are

$$\begin{cases} \frac{dx}{dt} &= v \\ \frac{dv}{dt} &= g - \frac{L_0 x_0 I^2}{2mx^2} \end{cases}$$

where

- $g = 9.81\text{m/s}^2$ is gravitational acceleration,
- $L_0 = 0.00802\text{H}$ is the solenoid-ball system inductance at equilibrium,
- $x_0 = 0.0116\text{m}$ is the desired vertical gap between solenoid and ball,
- I is the coil current control parameter, and
- $m = 0.008432\text{Kg}$ is the ball mass.

We take a game-theoretic approach for the purpose of synthesizing safe maglev control in the face of external perturbation and error introduced through modeling approximations and numerical simulation. The problem is thus described as a game where the controller may change the magnetic coil current while the adversary may perturb the behavior of the system in the period between controller actions. Specifically, using a fifth-order Cash-Karp Runge-Kutta method to simulate x and v over 0.01 sec to x' and v' within the region $0.005m \leq x \leq 0.018m$, $-0.3m/s \leq v \leq 0.3m/s$, and $0.03A \leq I \leq 0.83A$, the adversary may introduce relative error of at most 10%. Since we assume that actions are discretized, we constrain the controller to a uniform discretization of 20 currents from 0.03A to 0.83A, and we constrain the adversary to 8 perturbations of 10% in uniformly-distributed directions in the position-velocity plane of the state space.

3.3 SASAT Dynamic Programming Game-Graph Method

Cell mapping methods [20] have been used to perform state-space analysis of dynamical systems. In such methods the state-space is divided into cells. Each cell is mapped to another cell to which it will evolve after a fixed time interval. The resulting graph approximation of the system dynamics is then analyzed. One advantage of cell mapping is that one can form an approximation of the state space according to computational space limits, and perform an efficient, polynomial-time, global state-space analysis.

Dynamic programming, cell-mapping techniques for computing optimal control date back to the work of Wang[53] for systems described by first-order ordinary differential equations. For each quantized control vector, differential equations specify a directional field which can be discretized and used to compute cell-map transitions. Wang used a dynamic programming approach for the computation of optimal control policies. In this chapter, we augment his technique for multiple players, taking a more general simulation-based approach to cell-map discretization, and allowing for both discrete and continuous transition utilities.

In seeking to extend such methods to n-player games, we augment the cell map with set-valued mappings from a {cell, player} pair to a set of cells, circumscribing the possible effects of a player's actions in that cell. For each player, each cell is now mapped to a set of cells to which it may evolve after a fixed time interval. We refer to this augmented cell-map as a *game-graph*. Rather than performing minimax on a tree, we perform minimax on the approximating game-graph instead, thus reducing the exponential complexity of a minimax tree search to the polynomial complexity of a minimax graph search. Our generalization of minimax for n-players follows [27] where each player seeks to maximize its component of a score vector.

Algorithm 6 is the core procedure for our dynamic programming game-graph method. Following initialization, this procedure is iterated on the game-graph in

Algorithm 6 Iteration of Dynamic Programming Game-Graph Method

```
DynamicProgrammingIteration(gameGraph, player)
▷ Input: game-graph (augmented cell-map),
         current player number
▷ Output: game-graph with scores updated for one level of search
foreach cell in gameGraph do
    cell.newScoreVector ← negativeInfinityVector
    foreach destCell in cell.playerMap[player] do
        newScoreVector ← moveScore(cell, player, destCell) + destCell.scoreVector
        if (newScoreVector[player] > cell.newScoreVector[player]) then
            cell.newScoreVector ← newScoreVector
foreach cell in gameGraph do
    cell.scoreVector ← cell.newScoreVector
return gameGraph
```

reverse turn order in the dynamic programming style[1]. To initialize, first zero the game-graph score vectors. Then initialize the individual set-valued player maps which indicate the possible actions of each player at each cell. In applying this method to the maglev problem, the controller player map maps each cell to all other cells that differ only in controller input (current). The adversary player map maps each cell to the set of cells possibly reachable during the continuous system evolution phase, taking into account perturbation and error.

Since players need not necessarily alternate turns, let us for ease of analysis define b as the effective branching factor of the player mappings as used over successive calls to Algorithm 6. Let c be the number of cells and p be the number of players. Then the time and space complexity of Algorithm 6 are $O(cb)$ and $O(cpb)$, respectively. With player maps compactly represented and/or conservatively approximated, the space complexity may be reduced to $O(cp)$.

What we have not figured into this analysis is the "curse of dimensionality" in the state-space. If we divide a state-space into a uniform grid of cells, the number of cells will grow exponentially with the dimension of the space. Thus this method is only applicable to systems with low-dimensional state-spaces.

[1]Evaluation takes place from terminal states at some time horizon backwards in time through decision stages.

This method also places the burden of cell-partitioning and time discretization on the user. Too coarse a cell-partition, and such computation yields little information. Too fine a partition, and we violate computational space constraints. While adaptive techniques for cell-decomposition are being developed [4], these discretization issues are far from resolved.

The granularity of the cell-partition dictates the granularity of the approximated control policy. For our maglev problem, it would be desirable to have a finer discretization of the state space close to the desired goal state. Given that the goal state is a single point in the space, we might use some distance measure from this point to perform variable-size partitioning of the state space. We have not explored domain-specific improvements in this research in the interest of generality, and such domain-specific improvements are left as open problems.

The size of a simulation time-step used to build the augmented cell-map is another burden on the user. If too large a time-step is chosen in sampling behavior, there may be a number of undesirable consequences. A coarse sampling can result in an uninformative and unhelpful mapping. In skipping over too many cells, single limit cycles may appear to be multiple limit cycles, obscuring underlying system dynamics. Also, a system that may be stabilized when sampled above a certain rate may not be stabilizable below that rate. A coarse sampling can also result in an undesirably inaccurate mapping as simulation numerical errors can compound exponentially with simulation time. In choosing a small enough time step to avoid these problem, one must be careful not to pick so small a time step that cells that actually evolve to other cells begin mapping only to themselves. For further discussion of sample rate selection issues, see [12, Ch. 11].

One assumption of these techniques is that each successive layer of a tree or graph contains nodes that all occur at the same time. Search to a given depth is search to a given time horizon. If adaptive discretization techniques were to be applied to the choice of time-steps, then we would need to deal with evaluation of a tree without uniform time horizons.

We note that this method is not suited for real-time online use. While such a method could be used offline to form a control policy a priori, it is not designed

to focus on an immediately relevant control decision. Rather, its computation is distributed across the entire game-graph. This limitation is addressed in the graph-based negamax Algorithm 7. Negamax is an equivalent, alternate representation of minimax for two-player zero-sum games, where each player seeks a path that maximizes the negated return values of the next deeper level of search.

Algorithm 7 Negamax on a Game-Graph

```
tbh GAME-GRAPH-NEGAMAX(node, player, depth)
▷ Input: current node (or cell) of game-graph (augmented cell-map),
         current player number,
         depth of search at node
▷ Output: score returned by search
if (depth = 0 or leafNode(node) or node.complete[depth][player]) then
    return node.scoreVector[depth][player]
nextPlayer ← (player + 1) mod 2
bestNode ← null
bestScore ← −∞
foreach destNode in node.playerMap[player] do
    score ← moveScore(node, player, destNode)+
              − Game-Graph-Negamax(destNode, nextPlayer, depth)
    if (bestNode = null or score > bestScore) then
        bestNode ← destNode
        bestScore ← score
atomic:
    node.scoreVector[depth][player] ← bestScore
    node.bestNode[depth][player] ← bestNode
    node.complete[depth][player] ← true
return bestScore
```

As input, Algorithm 7 takes the current node, player, and depth of the search below the current node. As output, it returns the value of the subtree of the given depth at the given node for the given player. This algorithm could be used in real-time as an interruptible anytime algorithm that is called with sequentially greater depths as time remains. Over time, as more and more search results are cached, the algorithm is able to reuse these results to achieve deeper search over time. Memory would be preallocated and a depth limit set. As searches become complete to the given depth limit, search can be directed to other areas of the state space.

In summary, the dynamic programming game-graph method has polynomial time

and space complexity and is applicable to offline control design for low-dimensional state spaces, assuming that a good discretization can be found. For real-time applications, one would want to focus search relevant to the current situation. For such a situation, we describe a simple means of caching results from iteratively deepening negamax searches. We now turn our attention to the generalized hybrid alpha-beta methods in order to explore an even greater focusing of search along relevant lines of game-play.

3.4 SASAT Generalized Hybrid Alpha-Beta Method

In minimax search, a game-tree is generated with two players MAX and MIN, alternately maximizing and minimizing the score at alternating depths of the tree. However, much of the tree need not be generated (i.e. it can be "pruned") since it is provably irrelevant given information gained during search.

The origin of alpha-beta pruning is not clear. The following accounts of its early history are taken from Nils Nilsson [34, pp. 151-152] and Judea Pearl [35, p. 286]. Nilsson claims that alpha-beta pruning is "usually thought to be a rather obvious elaboration of the minimaxing technique" and conjectures that many people "discovered" it independently. Pearl claims that John McCarthy was the first to "recognize the potential for alpha-beta-type pruning" in 1956 and coined the name "alpha-beta". Nilsson points to an article by Newell, Shaw, and Simon [33] as the first description of alpha-beta, whereas Pearl points to a memorandum of McCarthy's students Hart and Edwards [14] which includes description of "deep cutoffs". Pearl notes that the 1958 chess-playing program of Newell, Shaw, and Simon (and probably the 1959 checker-playing program of Samuel) used only shallow cutoffs. Pearl claims that a full description of the algorithm with deep cutoffs was not published until Slagle and Dixon in 1969 [45]. Nilsson additionally points to Samuel's second checkers paper [43].

The core idea is this: If, in evaluating a node of a game tree, one can prove that a rational player will not choose the path to that node, one can avoid examination of (i.e. "prune") the subtree rooted at that node. By simple dynamic bookkeeping of the best score that each player can achieve, asymptotic optimality is gained for

such searches. In [23], it was shown that the asymptotic branching factor of search is $b/\log b$, where b is the effective branching factor without pruning. Thus, the asymptotic time complexity of alpha-beta search is $O((b/\log b)^d)$, where d is the search depth.

A recent description of alpha-beta search can be found in [41]. Alpha-beta search was generalized to n-players by Richard Korf in [24]. Korf proved that if one assumes an upper bound on the sum of player scores and a lower bound on each individual score, then deep pruning cannot occur for $n > 2$. Deep pruning of a node is based on a scoring bound inherited from a great-grandparent or more distant ancestor[2]. Only shallow pruning is possible for $n > 2$. In the best-case, shallow pruning reduces the asymptotic branching factor to $(1 + \sqrt{4b - 3})/2$. However, shallow pruning does not reduce the asymptotic branching factor. Thus we focus our attention on two-player alpha-beta search, noting that it can be generalized for n-players.

The zero-sum algebraic constraint over the scores provides the rational basis for alpha-beta pruning, but what if the game is not zero-sum? Interestingly, knowledge of one's problem domain may provide even more useful constraints. If it can be proved that one player will choose a move in a state that is guaranteed to cause another player to preclude the possibility of reaching that state out of preference for another line of play, all search beyond that state may be pruned. For instance, consider a cooperative form of the aircraft collision avoidance problem of [52], where all scores are identically the minimum distance between any two aircraft over time. Once all aircraft are receding from one another, we may obviously conclude that the scores will remain fixed. This is an example of a constraint on future scores which enables pruning without ever reaching cutoff states. Pruning constraints may take on other forms as well. If, for instance, it can be proved that the best adversarial maglev perturbation is a maximal perturbation, we reduce the dimensionality of relevant adversary actions. In broadening the constraints one considers, one may introduce far more significant forms of pruning to minimax search.

[2]That is, three or more nodes towards the root.

For real-time control, such an algorithm could be used within an iterative deepening, or iterative refinement anytime algorithm. By iterative refinement, we mean that we start with a coarse discretization of player decision points and compute an approximate solution (recommended control action) with our hybrid alpha-beta algorithm. We store the action, refine our discretization (i.e. allow more frequent turns), and iterate, computing successively better approximate solutions until the algorithm is halted and the stored action is returned. See Chapter 5 for a description of several iterative refinement approaches.

Although this approach does not require discretization of the state-space, the user still has to supply discretizations of continuous ranges of actions and decision times. Possible ways of dynamically choosing such discretizations are investigated in all chapters that follow.

One limitation of this approach is one shared by all tree-based methods: High branching factors quickly force shallow search. Since we are dealing with a minimax search on a tree rather than a graph, the time complexity is $O(b^d)$, where b is the effective branching factor and d is the maximum search depth. However, the space complexity is $O(d)$, so we have significantly traded off time for space. We have not only under-utilized computational space resources, but we have saved no information for future use and cannot expect search performance to improve over time. Given the infinite state-space of the search, and the approximate nature of simulation, it would make sense to use approximation and/or abstraction in order to achieve better performance over time. One possible step in this direction is to use alpha-beta with iterative deepening on a game-graph, caching results of partial alpha-beta computations in order to speed-up future minimax searches and allow greater depth of search over time. We introduce this new synthesis of techniques in Section 3.6.

3.5 Experimental Results

We have performed experimentation with the dynamic programming game-graph method and the alpha-beta pruning method. In both cases, the results were qualitatively comparable to those of Zhao[55, 28].

For the dynamic programming game-graph method, we chose to discretize the position, velocity, and output current to a $20 \times 20 \times 20$ uniform grid within the bounds given earlier. The controller takes a turn every 0.01 sec. These discretization choices were arbitrary. We have not experimented with other discretizations to see how performance would be affected.

Figures 3.3–3.6 show the mapping from input state x, v to output current I for the dynamic programming game-graph method iterated to depth 2, 4, 6, and 8. Figures 3.7–3.10 show trajectories from these respective control policies. As one can see, the depth 2 mapping gives the general qualitative behavior desired, and the depth 4 mapping is very similar to those for depth 6 and 8. For this problem, behavior appears to converge quickly in a few iterations, so it seems fortunate to have chosen such a time interval in our discretization. It would be interesting to experiment with adaptive step sizing for this method.

To apply the resulting policy to a controller, we simply perform a nearest-neighbor mapping at each time interval. Each input state is mapped to its corresponding cell, and the cell is mapped to an output current. The current is maintained for the next time interval[3], and the process is repeated indefinitely.

The front and back corners of these figures are losing cells (i.e. states from which the controller is guaranteed to lose), so 0.03A output current is as good as any other. However, not all 0.03A current cell outputs indicate a losing cell. Figures 3.11 and 3.12 indicate the cell scores for different cells. Since we have given cells that lead outside the game-graph bounds an arbitrary large negative score, these figures mainly differentiate between winning and losing states, that is, those states that can be kept within the game-graph region and those that cannot.

All states kept within the game-graph region are guaranteed to evolve to a small subset of cells about the desired cell. In practice, one could bring the system to the exact desired equilibrium state by switching to a control law derived by small-signal linearization as soon as the state came within a neighboring region about the

[3]This is called a *zero-order hold* in control terminology.

equilibrium state for which there exists a positive definite Lyapunov function[4]. Small-signal linearization of a magnetic levitation controller is demonstrated in [11, § 2.6.1].

The alpha-beta method did not, of course, need to be generalized to n-players for this problem domain. Our experimentation with it provided two significant pieces of information: (1) Memory allocation issues are significant to the efficiency of real-time applications. In comparing two implementations with different memory management, we found that preallocating memory and managing it was significantly faster than the allocating and deallocating memory through normal means. (2) The state-space discretization we used to approximate maglev system dynamics for the dynamic programming game-graph method did not significantly degrade performance, that is, we chose a good approximation earlier. While there may be analytic means of deriving appropriate discretizations for simple dynamical systems such as this, such choices are not obvious for complex systems. Again, it would be interesting to research adaptive discretization of the state space, so that the designer need not simply guess at what might be correct for complex systems.

Sample trajectories of the alpha-beta method can be seen in Figures 3.13–3.14. The arrows in the x-v plane are adversarial moves, while the vertical arrows are instantaneous controller current changes. These match up very nicely with Figures 3.3–3.4. Figures 3.15–3.16 show piecewise continuous trajectory segments and more clearly illustrate the global dynamics.

3.6 SASAT Alpha-Beta on a Game Graph

In this section, we introduce an algorithm for performing two-player alpha-beta on a game-graph. It could be argued that alpha-beta has long since been applied to discrete games with different means of reaching the same states. However, this approach is distinctive for a couple reasons.

First, alpha-beta search results are stored for *each sequential depth* of search previously performed. In literature on transposition tables, we have not found methods

[4]For an introduction to stability in dynamical systems, see [50, § 1.3].

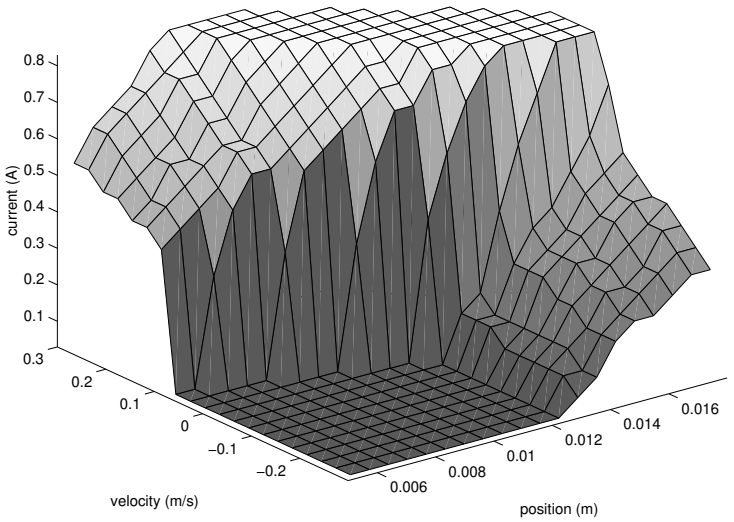

Figure 3.3: Maglev output currents from the SASAT dynamic programming game-graph method, depth 2

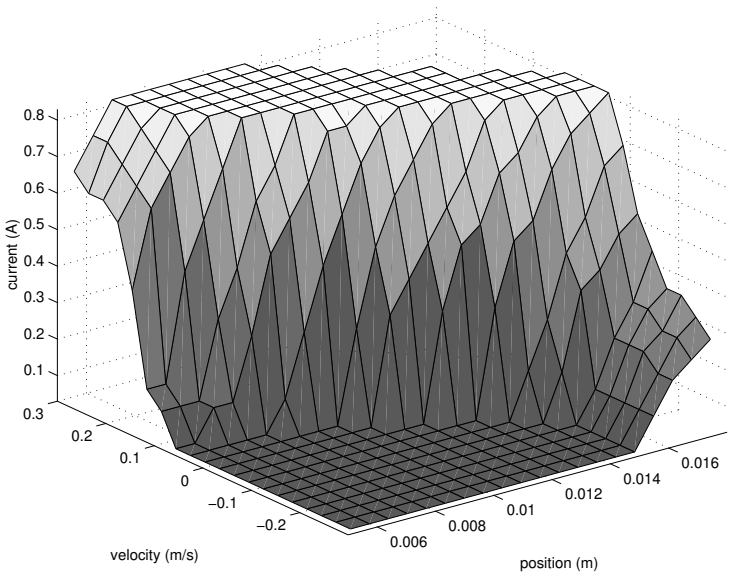

Figure 3.4: Maglev output currents from the SASAT dynamic programming game-graph method, depth 4

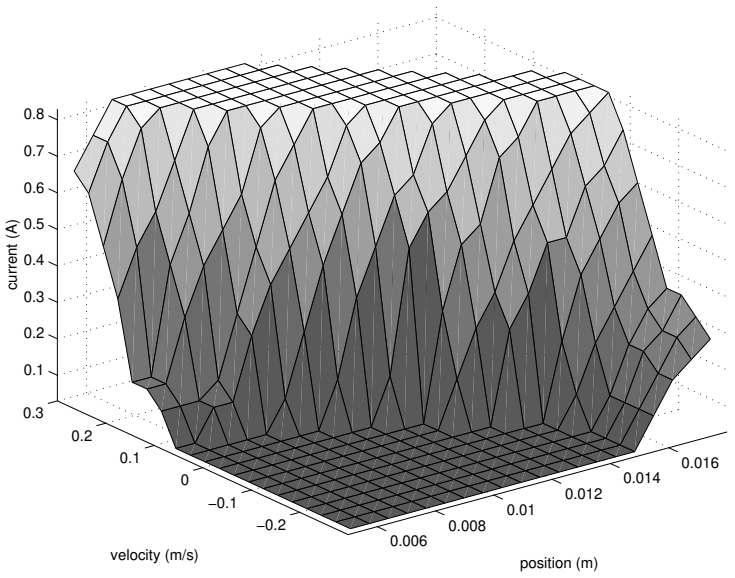

Figure 3.5: Maglev output currents from the SASAT dynamic programming game-graph method, depth 6

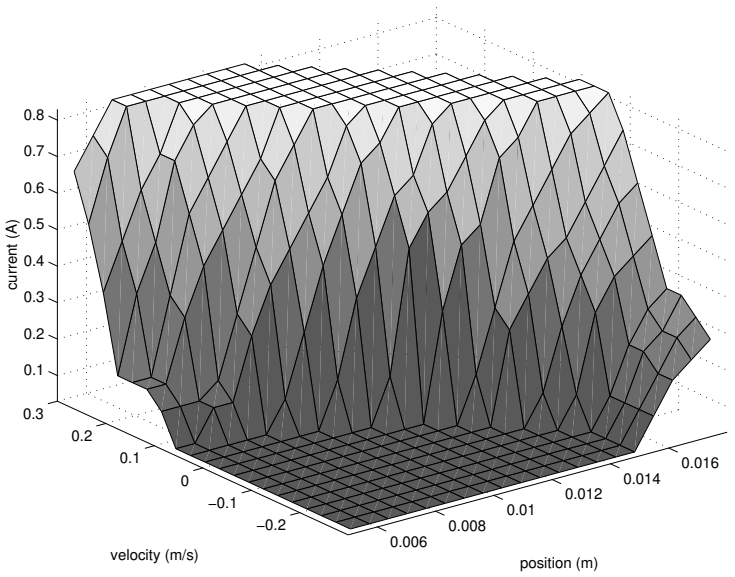

Figure 3.6: Maglev output currents from the SASAT dynamic programming game-graph method, depth 8

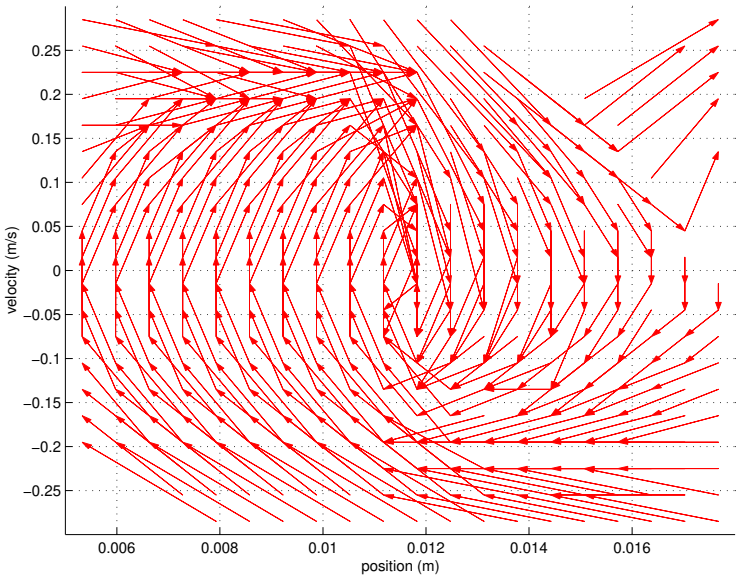

Figure 3.7: Maglev trajectories from the SASAT dynamic programming game-graph method depth 2

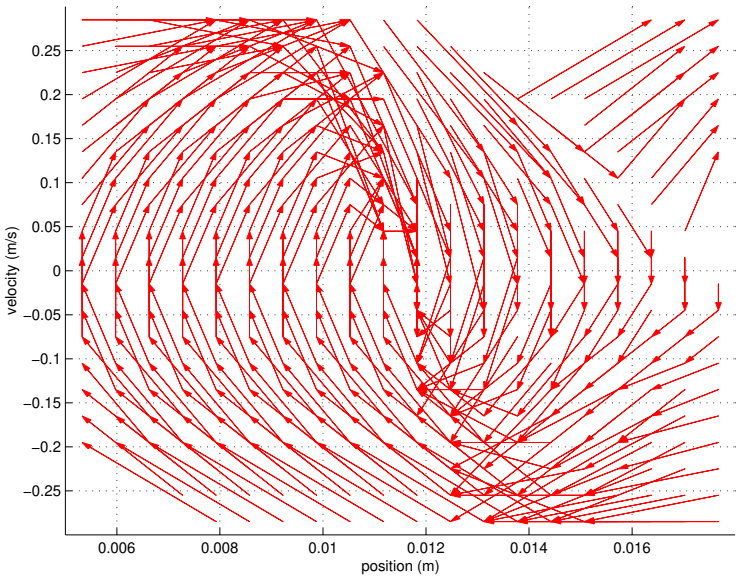

Figure 3.8: Maglev trajectories from the SASAT dynamic programming game-graph method, depth 4

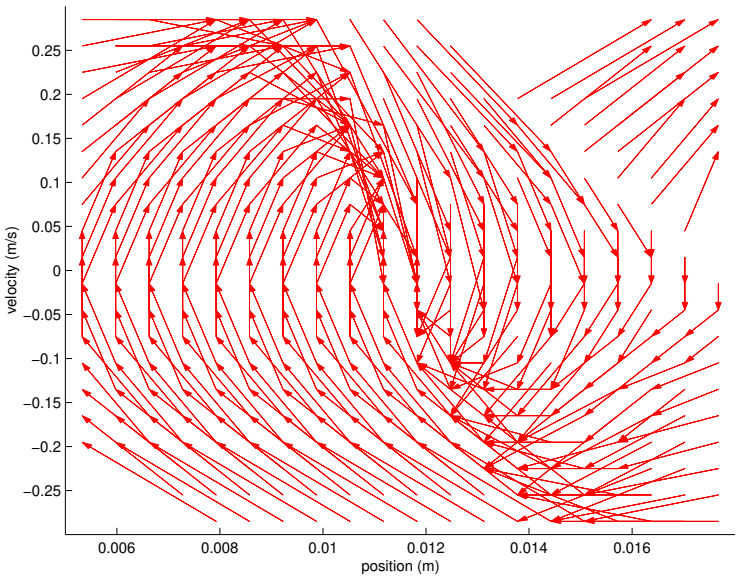

Figure 3.9: Maglev trajectories from the SASAT dynamic programming game-graph method, depth 6

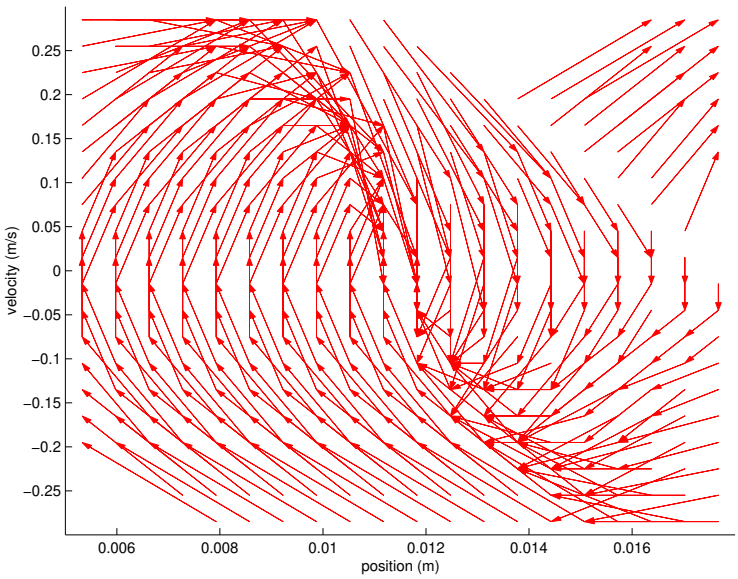

Figure 3.10: Maglev trajectories from the SASAT dynamic programming game-graph method, depth 8

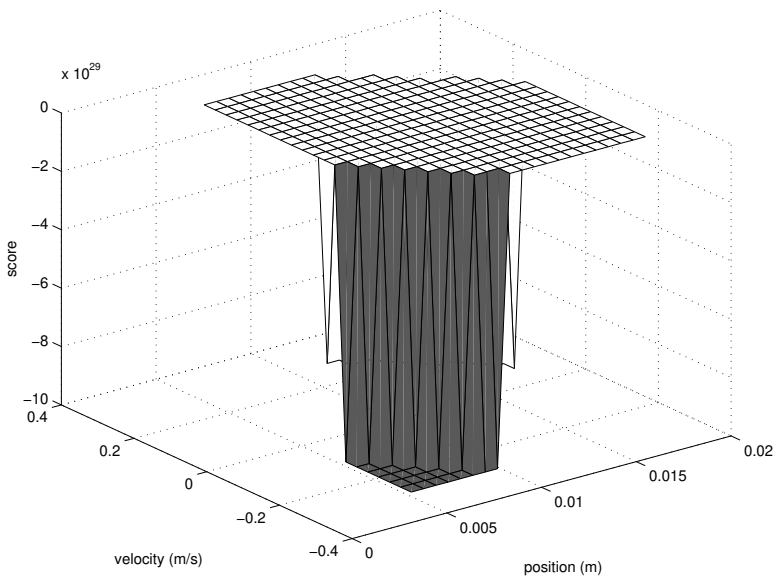

Figure 3.11: Maglev trajectory scores from the SASAT dynamic programming game-graph method, depth 2

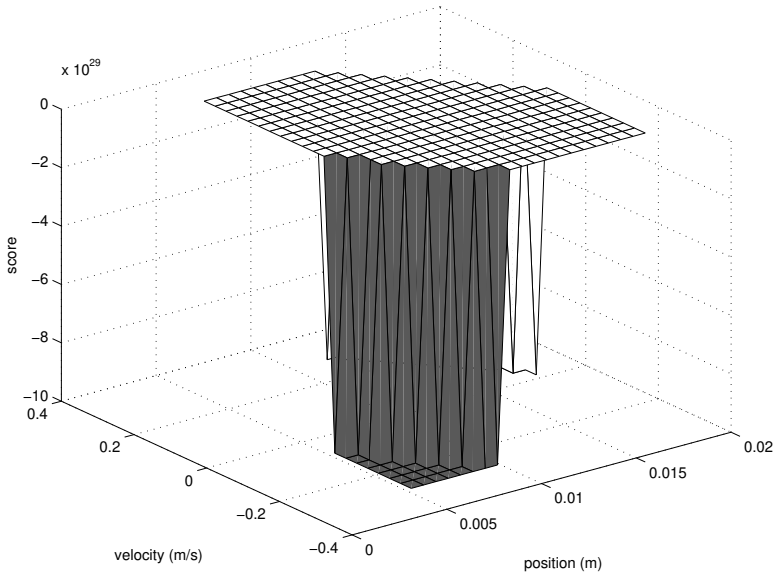

Figure 3.12: Maglev trajectory scores from the SASAT dynamic programming game-graph method, depth 8

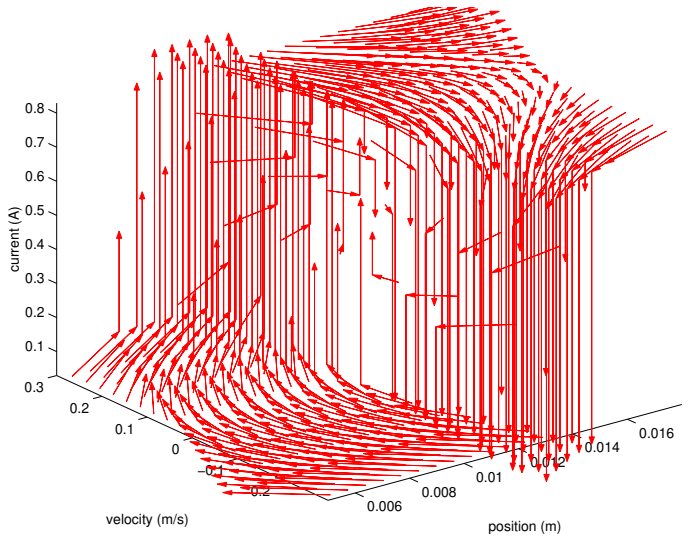

Figure 3.13: Maglev trajectories from the SASAT alpha-beta method, depth 2 (with current changes)

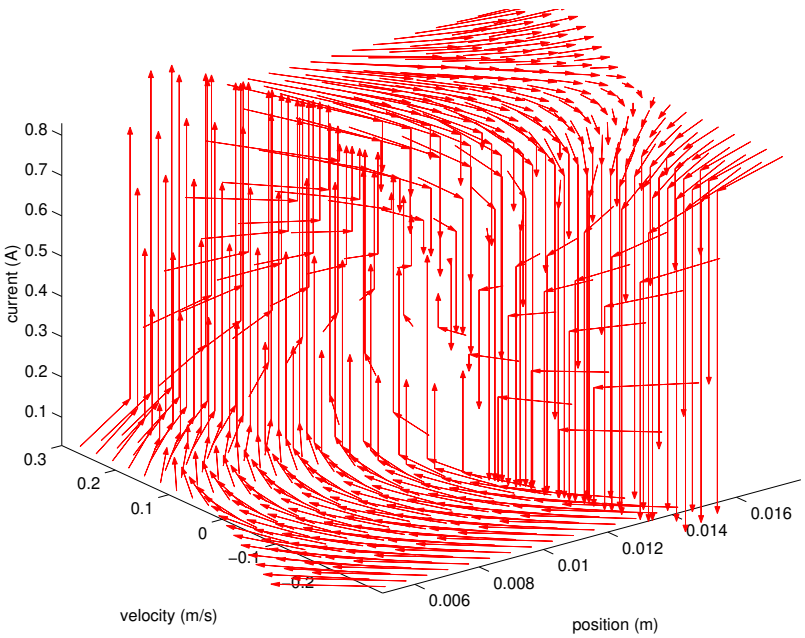

Figure 3.14: Maglev trajectories from the SASAT alpha-beta method, depth 4 (with current changes)

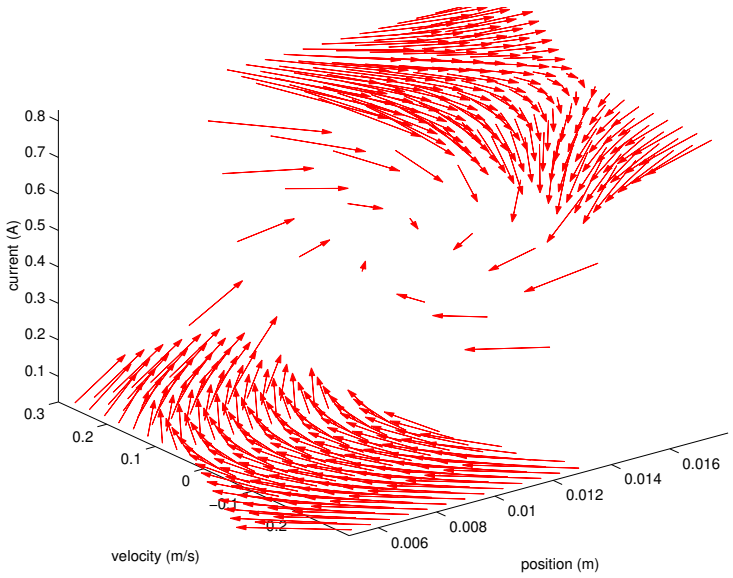

Figure 3.15: Maglev trajectories from the SASAT alpha-beta method, depth 2 (without current changes)

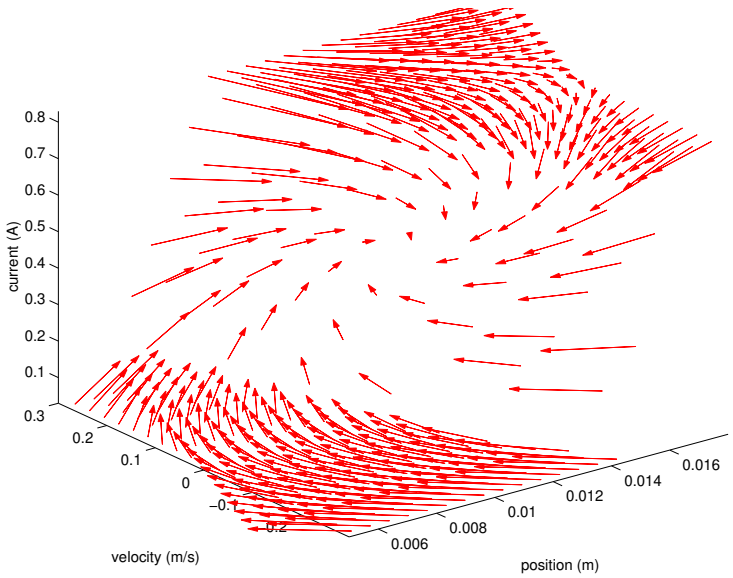

Figure 3.16: Maglev trajectories from the SASAT alpha-beta method, depth 4 (without current changes)

that store more than the deepest search performed at a node. Generally, applications of transposition tables are zero-sum games where players perform a sort of tug of war around an even score. In such a case, a deeper search will yield more useful information than a shallower search, so it makes sense to only store the deepest search performed. Zero-sum games with monotonically increasing/decreasing scores are not served well by such an approach. In this case, searches of equal depth should be compared at each node. Comparing scores from searches of different depths would bias moves in poor directions. Put simply, moves for such games should be evaluated with respect to a fixed time horizon.

The second distinctive feature of this search is our assumption that the entire game-graph can be enumerated and stored in memory. This is unusual in that most discrete games of interest to researchers do not have such small state spaces.

The pseudocode for our Game-Graph Alpha-Beta algorithm can be seen in Algorithm 8. Given a zero-sum game, one player (usually called MAX) maximizes score while their adversary (usually called MIN) minimizes score. Rather than write two procedures for the two players, we again take a negamax approach.

As input, Algorithm 8 takes the current node and player, scores for each player that can be guaranteed according to search so far, and depth of the search below the current node. The guaranteed scores are a vector $(\alpha, -\beta)$, where α is the lower bound and $-\beta$ is the negated upper bound of relevant search values at that node. As output, it returns the weakest pruning conditions used in the search. This algorithm is used in real-time as an interruptible anytime algorithm that is called with sequentially greater depths as time remains. Over time, as more and more search results are cached, the algorithm is able to reuse these results to achieve deeper search over time. Memory would be preallocated and a depth limit set. As searches become complete to the given depth limit, search can be directed to other areas of the state space.

The Game-Graph Alpha-Beta algorithm begins by checking if (1) search is at its depth limit, or (2) the current node is a leaf node. If so, a vector of worst possible scores are returned, indicating that no pruning conditions were used from previous search in searching the subtree at that node. Recall that both players are maximizing the negated scores of the subtrees at each level.

Algorithm 8 Alpha-Beta on a Game-Graph

```
GAME-GRAPH-ALPHA-BETA(node, player, prevGuaranteeVector, depth)
▷ Input: current node (or cell) of game-graph (augmented cell-map),
         current player number,
         guaranteed player scores from previous search (α, −β),
         depth of search at node
▷ Output: weakest pruning conditions used in search
if (depth = 0 or leafNode(node)) then
    return {−∞, −∞}
if (prevGuaranteeVector ≥ node.pruneCondVector[depth][player]) then
    return node.pruneCondVector[depth][player]
otherPlayer ← (player + 1) mod 2
scoreGuaranteeVector ← prevGuaranteeVector
pruneCondVector ← {−∞, −∞}
bestNode ← null
bestScore ← −∞
foreach destNode in node.playerMap[player] do
    childPruneCondVector ← Game-Graph-Alpha-Beta(destNode, otherPlayer,
                                                 scoreGuaranteeVector,
                                                 depth)
    pruneCondVector ← max(pruneCondVector, childPruneCondVector)
    s ← moveScore(node, player, destNode) + −destNode.abScore[depth][otherPlayer]
    if (bestNode = null or s > bestScore) then
        bestNode ← destNode
        bestScore ← s
        if (s ≥ −prevGuaranteeVector[otherPlayer]) then
            pruneCondVector[otherPlayer] ← max(pruneCondVector[otherPlayer],
                                               prevGuaranteeVector[otherPlayer])
            goto prune
        if (s > scoreGuaranteeVector[player]) then
            scoreGuaranteeVector[player] ← s
prune:
if (s ≥ pruneCondVector[player]) then
    pruneCondVector[player] ← −∞
atomic:
    node.abScore[depth][player] ← bestScore
    node.bestNode[depth][player] ← bestNode
    node.pruneCondVector[depth][player] ← pruneCondVector

return pruneCondVector
```

Next, we check the weakest preconditions of previous cached search information to see if the results can be reused. If so, we return those weakest preconditions. The weakest precondition for all searches must be initialized to the best possible scores (∞, ∞) in order to ensure that an initial search occurs. When a search is completed without relying on given score guarantees for pruning, the weakest pruning conditions will be $(-\infty, -\infty)$. Hence that search is complete and stored results will always be reused.

After initialization of a number of variables, we then turn our attention to each possible destination node for the player from the current node. For each, we perform a recursive call to Game-Graph Alpha-Beta, record the strongest pruning conditions used in the subtree search, and record the score. If the score is the best seen at this node, we note the new best score and destination node. If the score violates a zero-sum constraint with the guarantees, then we have proven that the rational adversary will not allow the game to progress to this point and thus prune the remaining searches, making note of the pruning condition. Otherwise, we update the current player score guarantee if necessary.

After searching destination nodes as necessary, we check if the current player's subtree search score satisfies the weakest pruning condition for that player in the subtree search. If so, then no guarantees for the player's score above the subtree were necessary for the pruning, and we set the weakest pruning condition for that player to $-\infty$.

Finally, we record the results of the search. This block of code is marked "atomic" to indicate that interruption of the algorithm within this block would potentially leave the data in an erroneous state.

One straightforward heuristic for speeding up such search is to use the best node of previous search (of similar depth) as the first node for exploration. By looking at a strong potential best move first, we are more likely to set tighter pruning bounds earlier in the search.

It should be noted that for a given node, player, and search depth, successive calls with overlapping bounds would result in a search never being complete. One could construct pathological global search and calling conditions such that asymptotic

global behavior over time would be better served by avoiding pruning altogether. It is not clear how often such situations could arise in practice. In Chapter 4, we will see that pruning can yield such significant search speedup in this domain, so that even without storage and reuse of search results, alpha-beta pruning is well-applied to this problem domain.

3.7 Relation to Memory-Based Techniques

In [32], Moore, Atkeson, and Schaal present a collection of *memory-based* techniques for learning control. Of particular relevance to the work of this chapter is their research into optimal control with nonlinear dynamics and costs[32, §7]. In this section, we give an overview of their memory-based approach, compare and contrast it with our own, and note possible directions for future work.

Developed independently, memory-based approaches explicitly remember all previous experiences and apply such knowledge to the problem of learning control. Prediction and generalization are performed online in real-time by building a local model to answer any query, where a query is a current state and desired resulting system behavior, and an answer to a query is an action mapping the current state to the desired behavior. Although the idea is more general, stored experiences are used to build local models represented as polynomial approximations of system evolution. Parameters for the polynomial are estimated using linear weighted regression (LWR). Such techniques are said to provide explicit parameters to control smoothing, outlier rejection, and forgetting. The last process is particularly important for the development of memory-bounded variants.

Moore et al describe system dynamics as an unknown function

$$\mathbf{x}(t+1) = \mathbf{f}(\mathbf{x}(t), \mathbf{u}(t)) + \text{noise}(t)$$

with a known cost function

$$c(t) = \text{cost}(\mathbf{x}(t), \mathbf{u}(t)).$$

The task is minimization of one of the following cost summations:

$$\sum_{t=0}^{\infty} c(t) \;\; or \;\; \sum_{t=0}^{t_{max}} c(t) \;\; or \;\; \sum_{t=0}^{\infty} \gamma^t c(t) \; where \; 0 < \gamma < 1 \;\; or \;\; \lim_{n\to\infty} \frac{1}{n} \sum_{t=0}^{n} c(t)$$

The authors note that there is a large literature on such problems in the context of *reinforcement learning*. The state space is discretized into a multidimensional array of cells, and system dynamics are approximated to cell centers as with cell map methods. They present the following basic approach, called *Memory-Based Reinforcement Learning* which uses a dynamic programming value iteration to compute an optimal value function:

1. Observe the current state $\mathbf{x}(t)$ and choose action $\mathbf{u} = \pi(\mathbf{x})$, where π is the current estimated optimal control policy.

2. Perform action and observe next state $\mathbf{x}(t+1)$.

3. Add $(\mathbf{x}(t), \mathbf{u}) \rightarrow \mathbf{x}(t+1)$ to the memory base.

4. Recompute the optimal value function and policy using value iteration with the new information.

Value iteration is computationally expensive, so this algorithm would not be suited to fast, real-time application. Experimentally, it was used with a simulated system that had its state frozen while updating its policy. The authors suggest that for normal usage one would update the value function and policy at the end of each trial or in an incremental parallel process.

Convergence of reinforcement learning is dependent on the system visiting each state-action pair infinitely often. Memory-based reinforcement learning does not probabilistically explore as do most reinforcement learning algorithms. The result of this lack of exploration is that it converges to correct behaviors faster when the learned model does not contain significant errors. The authors point out that significant noise can introduce errors that steer the system in significantly suboptimal directions while such memory persists. Thus, the guarantee of convergence to an optimal solution is

traded off for speed of convergence to a solution, much the same way that simulated quenching does in the context of simulated annealing. In practice, this can be quite sensible. In fact, simulated quenching with random restarts is in popular use among those who use simulated annealing. We suggest that one might combine the resulting policies of multiple runs of memory-based reinforcement learning to synthesize a policy augmented with risk information.

Two experiments were performed with a simple nonlinear dynamical system involving the positioning of a puck on a curved one-dimensional surface. In the first experiment, unvisited states were assumed to have a cost of zero. In the second experiment, transitions between cells were predicted using locally weighted linear regression from previous observations. The second achieved behavior within 3% of optimal with two orders of magnitude fewer steps than in the first experiment.

There are a number of similarities and differences between this approach and ours that are worth noting. First, we note that the system model includes noise and is nondeterministic. Our approaches assume determinism. However, this difference is not so significant when one considers that memory-based approaches treat system behavior as deterministic. In not visiting state-action pairs infinitely often, there is an underlying assumption that what has been observed need not be re-observed for different behavior. In this sense there is little difference between how information is treated in memory-based and simulation-based approaches. In contrast, we choose to treat nondeterminism pessimistically. Rather than treating possible system perturbations or errors as random, we imbue such behavior with intelligence and design for the worst case. Different treatment of nondeterminism will be appropriate for different tasks. It would be interesting to see memory-based reinforcement learning methods extended for Markov games and see how such approaches work in the context of multi-player games.

The authors stress that memory-based approaches are model-free and only construct local models of behavior as is necessary. Simulation-based techniques assume a simulatable model is given. This would again seem to be a significant difference. However, we note that memory-based experiments relied on the use of simulations. Modifications to such approaches (e.g. that decide when to perform computationally

expensive dynamic programming) are necessary for physical experimental use. In our SASAT work, we have focused on means of reducing the amount of and maximizing the immediate utility of dynamic programming computation between each action in real-time. The algorithms described in [32, §7] are not so model-free as those referenced in the same section. In practice, the authors suggest that dynamic programming should be performed at the end of each trial, or as an incremental parallel process.

What is perhaps most valuable and instructive from their approach is the powerful use of prediction based on previous experience. Such predictive interpolation based on previous experience could potentially find powerful application in the alpha-beta approaches of this chapter if storage, retrieval, and local model construction did not introduce too much computational overhead. For example, it is well known that node ordering can significantly increase pruning and thus the speed of alpha-beta search. This will be seen experimentally in the next chapter. If such prediction can be efficiently used for intelligent node-ordering, then our approach could be significantly improved.

3.8 Summary and Discussion

In this chapter, we examined three ways of using simulation and game-tree search to inform robust control of a magnetic levitation controller. In the first, we used a dynamic-programming approach with an augmented cell-map or game-graph. In searching a graph approximation of the dynamic game, we reduce search time complexity from exponential to polynomial. Our dynamic programming method for augmented cell maps has polynomial time and space complexity and is applicable to offline control design for low-dimensional state spaces, assuming that a good discretization can be found.

Next, we discussed current techniques for alpha-beta search (without approximation) and showed that the resulting control policy of earlier approximation is indeed close to that found using alpha-beta search. Alpha-beta pruning is a form of irrelevance reasoning which increases efficiency of minimax search. We discussed the

history of alpha-beta and the reason why it is best applied to two-player games.

Finally, we combined the best of both algorithms in an algorithm called Game-Graph Alpha-Beta, which has a novel form of caching results of alpha-beta search for future reuse. This provides a more efficient means of online hybrid system control for low-dimensional state spaces, assuming that a good discretization can be found.

From our experimental results we note that our choice of discretization was fortunate, as a depth-four (two turn) game-tree search yields a control policy nearly convergent with the optimal policy yielded by Algorithm 6 when iterated to convergence. As this was accidental, we do believe that future work should be done to dynamically adapt discretization stepsize. First steps in this direction are made in the context of tree-search in Chapters 5 and 6.

One might ask where such techniques are most usefully applied. First, we observe that search is a complex generalization of generate-and-test optimization. Global optimization techniques of the previous chapter are most usefully applied to functions that do not have properties assumed by more specialized techniques that take advantage of such problem-domain-specific knowledge. In the same way, game-tree or tree search techniques are most usefully applied to informing intelligent control of systems that do not have properties assumed by the more specialized techniques of classical control.

Second, we note that many techniques of control require the system to have a specific analytical form. In contrast to control techniques such as feedback linearization, we do not constrain our system to a specific analytical form. For most of our algorithms, we assume that a system simulator is given. However, the augmented cell-map techniques we have presented require only sufficient time-series data to approximate system dynamics. Furthermore, in reviewing the memory-based control work of Moore, Atkeson, and Schaal, we note that simulation can be approximated through the interpolation of time-series data. From this perspective, our techniques not only enable model-based control, but can also be applied without explicit models given an appropriate means of interpolating unseen system behavior.

Two issues concerning minimax and alpha-beta motivate future research in reasoning about uncertainty and relevance in game-tree search. First, minimax search assumes no uncertainty in node evaluations, so small errors in node-evaluations may significantly misinform decisions. Second, alpha-beta pruning is concerned entirely with provable irrelevance given such an assumption. Without the ability to focus search direction according to probable relevance to the root decision, alpha-beta search is ill-equipped to handle large branching factors, forcing an arbitrary, pre-determined pruning or discretization (for continuous ranges of actions). Automatically choosing state-space or action-space discretizations according to the task of real-time reasoning about control is an open problem. Even given a good discretization of a hybrid system control game, a large branching factor can force an impractically shallow search and yield poor decisions.

Probabilistic game-playing methods [42] have been developed to handle uncertainty and to direct search with relevance to maximizing expected utility of the decision. This still leaves overarching discretization questions concerning continuous state-spaces, ranges of actions, and decision points in intervals of time. In future chapters, we show that previous work on information-based optimization (Chapter 2) will be relevant in addressing such questions. Briefly, information-based optimization is concerned with using the information from previously sampled points to inform the choice of future sample points. Using such optimization to dynamically choose the sampling of actions and decision points provides an interesting study in the tradeoff between cost and benefit of metalevel reasoning in search.

As algorithms employ increasingly computationally complex meta-level reasoning, computational overhead will grow to the point of diminishing returns in overall utility. Over time, we expect to develop a suite of methods that lie along a spectrum of computational complexities of meta-level reasoning, and describe their applicability to different classes of hybrid system control games. We hope that these will contribute to development of algorithms for real-time control and bounded rationality.

Chapter 4

DASAT Game-Tree Search

Extending discrete search to hybrid system search introduces two new decisions in optimization: action discretization and action timing discretization. In this chapter we choose to address the former decision: How could a search algorithm choose how to branch the search tree considering continuous spaces of possible actions parameters? We will assume that action timing, i.e. when decisions are made, is already given. From the perspective of the search algorithm, action discretizations are dynamic, i.e. a sample of possible actions for each search node is chosen by the search algorithm. However, from the perspective of the search algorithm, action timing discretizations are static, i.e. the search algorithm cannot affect the action timing discretization. For this reason, we will call such searches "DASAT searches" as they have Dynamic Action and Static Action Timing discretization.

In this chapter, we formally define a DASAT Hybrid System Game and its solitaire case, a DASAT Hybrid System Search Problem. We continue to examine the magnetic levitation problem of the previous chapter, and compare the relative merits of random, uniform, and information-based discretizations in the context of alpha-beta search. We present information-based alpha-beta search, a novel application of information-based optimization which uses the α lower bound and β upper bound of alpha-beta search to optimize for pruning. The resulting algorithm exceeds the good speed and pruning performance of random discretization while matching the control policy quality of uniform discretization.

4.1 DASAT Hybrid System Game and Search Problem

Formally, a DASAT Hybrid System Game is defined as a 7-tuple

$$\{S, s_0, \mathbf{A}, p, l, m, d\}$$

where

- S is the hybrid state space with a finite number of finite discrete variable domains, and a finite-dimensional continuous space,
- $s_0 \in S$ is the initial state,
- $\mathbf{A}$ is a finite set $\{A_1, \ldots, A_n\}$ of continuous action regions indexed $\{1, \ldots, n\}$,
- p is the number of players,
- $l : S \times \{1, \ldots, p\} \rightarrow \mathbf{A}'$ where $\mathbf{A}' \subset \mathbf{A}$ is a *legal move* function mapping from a state and player number to a finite set of legal continuous action regions which contain points representing all legal actions that may be executed in that state by that player,
- $m : S \times \mathbf{a}^p \rightarrow S \times \Re^p$ is a *move* function mapping from a state and simultaneous player actions (region index, region point pairs) to a resulting state and the utility of the combined actions for each player,
- $d : S \rightarrow S \times \Re^p$ is a *delay* function mapping from a state to the resulting state and the utility of the trajectory segment for each player. This delay governs the evolution of the system through time between moves.

The total utility of any finite trajectory is computed as the sum of the trajectory move and delay utilities. In this time-invariant formalism, time can easily be encoded in a continuous clock variable, and time invariant behavior could thus be easily achieved.

Although not addressed in this chapter, a *DASAT Hybrid System Search Problem* is a special case of the DASAT Hybrid System Game where we are interested in finding a trajectory from the initial state to a goal state. Usually such problems are stated in terms of path cost rather than utility. Formally, a DASAT Hybrid System Search Problem is defined as a 7-tuple

$$\{S, s_0, S_g, \mathbf{A}, l, m, d\}$$

where

- S is a hybrid state space with a finite number of finite discrete variable domains, and a finite-dimensional continuous space,
- $s_0 \in S$ is an initial state,
- $S_g \subset S$ is a set of goal states,
- $\mathbf{A}$ is a finite set $\{A_1, \ldots, A_n\}$ of continuous action regions indexed $\{1, \ldots, n\}$,
- $l : S \rightarrow \mathbf{A}'$ where $\mathbf{A}' \subset \mathbf{A}$ is a *legal move* function mapping from a state to a finite set of legal continuous action regions which contain points representing all legal actions that may be executed in that state,
- $m : S \times \mathbf{a} \rightarrow S \times \Re$ is a *move* function mapping from a state and action (region index, region point pair) to a resulting state and cost of the action,
- $d : S \rightarrow S \times \Re^p$ is a *delay* function mapping from a state to the resulting state and the cost of the trajectory segment for each player. This delay governs the evolution of the system through time between moves.

We next describe a DASAT Hybrid System Game in the domain of magnetic levitation.

4.2 DASAT Magnetic Levitation Problem

The DASAT version of the SASAT Magnetic Levitation Problem of Section 3.2 is the same with only one modification: action discretizations are no longer given. The magnetic levitation unit can now choose any current between 0.03A and 0.83A. The adversary can now perturb the system 10% in any direction in the position-velocity plane of the state space.

In this chapter, we focus solely on comparisons of discretization quality in the context of alpha-beta search. In all cases, we retain the same branching factors of the discretization of the previous chapter, thus facilitating ease of comparison. Three different discretizations are studied: random, uniform, and information-based.

4.3 DASAT Alpha-Beta Search with Random Discretization

DASAT Alpha-Beta Search with Random Discretization is a simple augmentation of SASAT Hybrid Alpha-Beta Search (Section 3.4) with moves being randomly chosen rather than given as a fixed discretization of possible action parameter regions. We globally fix a maximum number of samples for each action parameter region. For each recursive call of the algorithm for a node, samples are randomly chosen from action parameter regions. For each sampled move, a new child (possible future node) is generated, recursively searched, and results of the search are returned. This continues until either (1) we reach the maximum number of samples, or (2) the result of search indicates that we can prune future search from this node.

Experimental results of DASAT Alpha-Beta Search with Random Discretization on the magnetic levitation problem are shown in Table 4.1. Figures 4.1, 4.2, and 4.3 show the control policies (mappings from position and velocity to current) resulting from searches to depths 2, 4, and 6, respectively. From the control policy, we see that the outputs are rough. Results of the previous chapter indicate that much of the control policy space should have currents at extreme values. Given the random nature of discretization, we only approximate such extreme values.

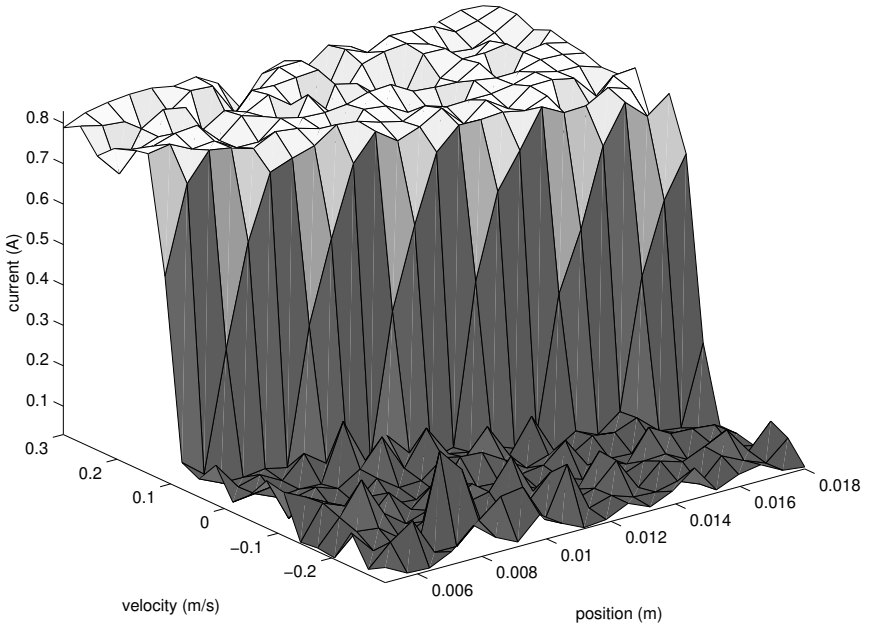

Figure 4.1: Maglev output currents from DASAT Alpha-Beta with Random Discretization, depth 2

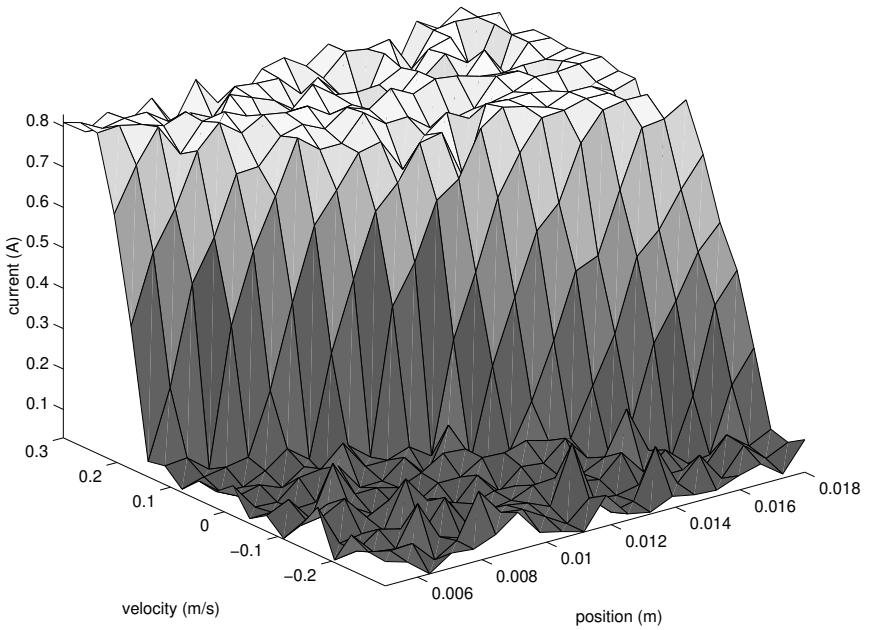

Figure 4.2: Maglev output currents from DASAT Alpha-Beta with Random Discretization, depth 4

Depth	Trials	Average Time (msec)	Average Nodes	Average Pct. Pruned	Average Nodes/Sec	Average Score
1	400	1	21	0.00	21,538	-1.59E-7
2	400	1	66	63.30	80,275	-1.59E-7
3	400	36	748	77.89	20,958	-3.43E-7
4	400	43	2,057	92.90	47,918	-3.42E-7
5	400	867	21,806	95.97	25,153	-5.73E-7
6	400	1,124	66,042	98.58	58,778	-5.70E-7

Table 4.1: Results for DASAT Alpha-Beta Search with Random Discretization

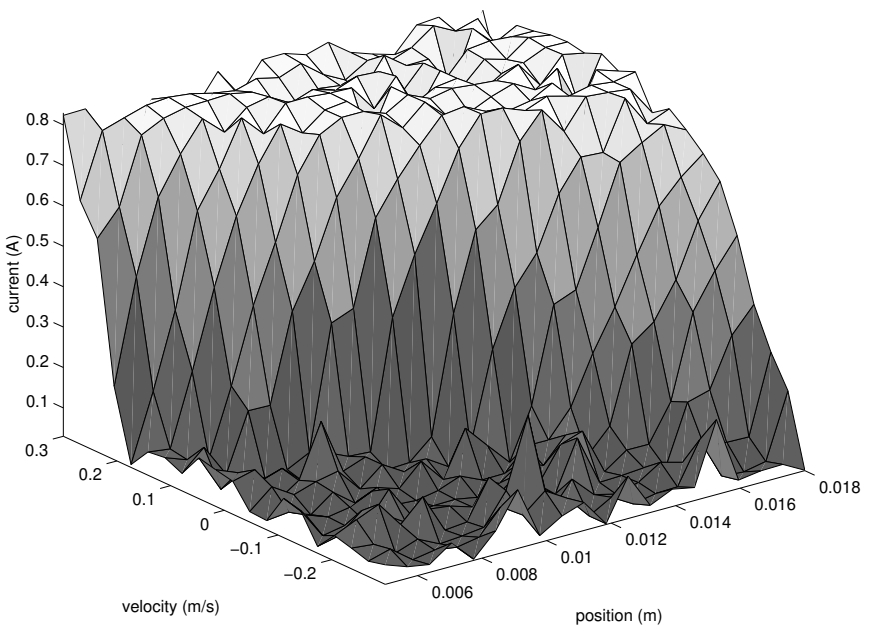

Figure 4.3: Maglev output currents from DASAT Alpha-Beta with Random Discretization, depth 6

4.4 DASAT Alpha-Beta Search with Uniform Discretization

In global optimization of Lipschitzian functions with an unknown constant, it has been shown that a uniform grid on a compact feasible set provides the best selection of candidate points for optimization[51]. In a sense, this is much like information-based optimization over a compact feasible set where the functions are finite-valued and the target is infinite. In this extreme case, each next best candidate point is the point which is farthest from all previously evaluated points. Thus, from two points of view, uniform discretization is the best approach to choosing a set of points for evaluation when one lacks information about a function extreme.

DASAT Alpha-Beta Search with Uniform Discretization is another simple augmentation of SASAT Hybrid Alpha-Beta Search (Section 3.4) with moves being uniformly chosen rather than given as a fixed discretization of possible action parameter regions. In fact, this yields the same discretization which was used in the previous chapter. A globally fixed maximum number of samples are uniformly chosen from the lower bound to the upper bound of a one-dimensional action parameter region. The general case of multidimensional, arbitrarily-shaped, closed regions is treated later in Section 6.5. For each action region, the globally fixed maximum number of uniformly sampled moves are generated. For each recursive call of the algorithm for a node, we try each successive move sampled from each successive legal move region until either (1) all moves have been considered, or (2) the result of a search indicates that we can prune future search from this node.

Experimental results of DASAT Alpha-Beta Search with Uniform Discretization on the magnetic levitation problem are shown in Table 4.2. Figures 4.4, 4.5, and 4.6 show the control policies (mappings from position and velocity to current) resulting from searches to depths 2, 4, and 6, respectively. From the data, we can see that search execution is slower and pruning is less than that achieved by random discretization. Since the discretization is as in the previous chapter, the control policy is identical to that of alpha-beta search of the previous chapter.

Pruning is considerably less than that achieved by the random discretization.

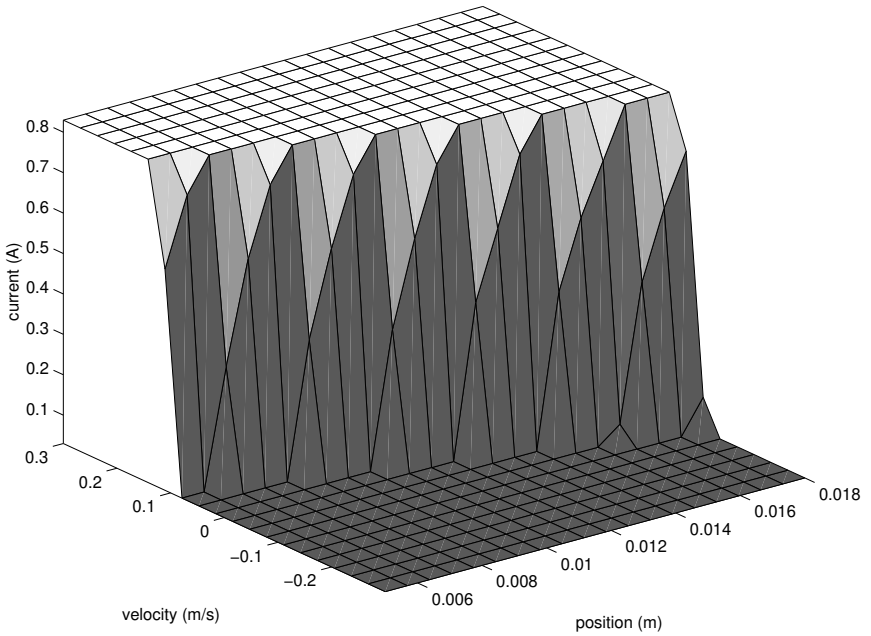

Figure 4.4: Maglev output currents from DASAT Alpha-Beta with Uniform Discretization, depth 2

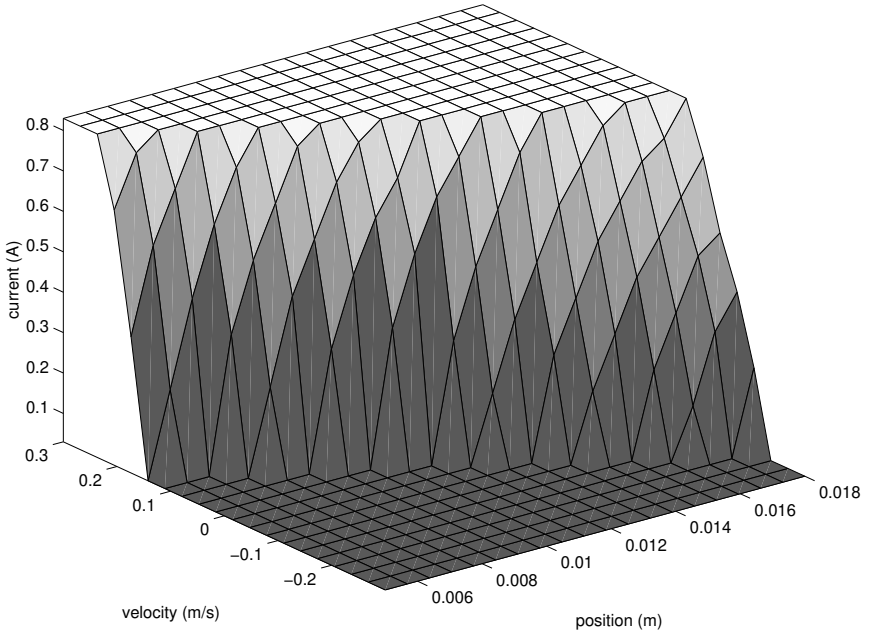

Figure 4.5: Maglev output currents from DASAT Alpha-Beta with Uniform Discretization, depth 4

Depth	Trials	Average Time (msec)	Average Nodes	Average Pct. Pruned	Average Nodes/Sec	Average Score
1	400	1	21	0.00	19,047	-1.58E-7
2	400	1	113	37.34	92,395	-1.58E-7
3	400	51	1,957	42.11	38,154	-3.31E-7
4	400	69	7,156	75.31	103,378	-3.31E-7
5	400	1,598	81,678	84.90	51,125	-5.26E-7
6	400	2,145	264,020	94.31	123,112	-5.26E-7

Table 4.2: Results for DASAT Alpha-Beta Search with Uniform Discretization

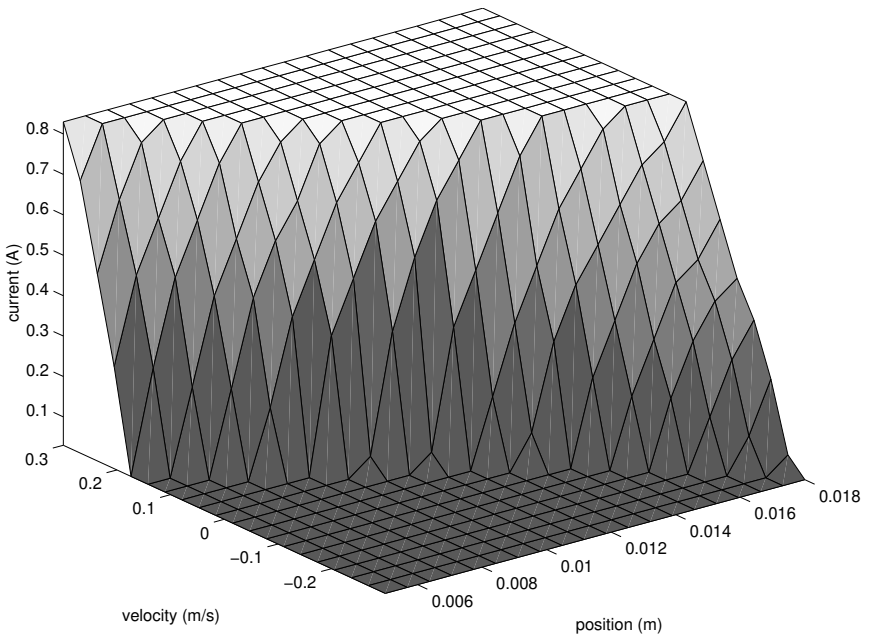

Figure 4.6: Maglev output currents from DASAT Alpha-Beta with Uniform Discretization, depth 6

In the space of mappings, extreme current values are the most common output. Pruning will naturally be greater for algorithms which sample both extremes in earlier expansions. Information-based discretization checks extreme values first, random discretization checks randomly, and uniform discretization checks uniformly from one extreme to another. Uniform discretization will start checking possible moves at the wrong extreme for pruning roughly half of the time that an extreme value will be optimal for pruning. This accounts for the poor pruning results. For this problem domain, we conjecture that a greedy node ordering heuristic would yield much better pruning results. We will discuss this point further in the next section.

4.5 DASAT Information-Based Alpha-Beta Search

DASAT Information-Based Alpha-Beta Search is our third augmentation of SASAT Hybrid Alpha-Beta Search (Section 3.4) with moves being chosen according to previous choices and their respective subtree search results. A pseudocode description of this method is given in Algorithm 9. In alpha-beta search, α and β represent the lower and upper bound of possible local game-tree search respectively. At the current node under evaluation, we have a guarantee that MAX can score at least α while MIN will limit MAX to scoring at most β. If we wish to maximize pruning, then α and β provide appropriate target values for information-based discretization.

Uniform discretization provides the best discretization if our target is not bounded. Indeed, in the extreme case where we have no guaranteed α or β, information-based optimization becomes uniform discretization, always choosing the next point to be farthest from those previously evaluated. However, if we are given bounds to possible values for game-tree search, then we can use such target values to inform intelligent search. Information-based optimization is a natural choice for this application for two reasons: (1) The objective function (subtree evaluation) is computationally intensive compared to information-based optimization[1], and (2) We have natural target values to inform optimization.

[1]This holds for the one-dimensional case. As we will see in Chapter 6, the computational complexity of multidimensional information-based optimization can be overly burdensome.

Algorithm 9 Information-Based Alpha-Beta Search

```
INFO-BASED-ALPHA-BETA(node, player, prevGuaranteeVector, depth)
▷ Input: current node,
         current player number,
         guaranteed player scores from previous search (α, −β),
         depth of search at node
▷ Output: current node with search results
if (depth = 0 or leafNode(node)) then
    node.abScore ← score(node)
    if (player = 1) then
        node.abScore ← −node.abScore
    node.bestMove ← null
    return node
otherPlayer ← (player + 1) mod 2
scoreGuaranteeVector ← prevGuaranteeVector
bestMove ← null
bestScore ← −∞
foreach region in legalMoveRegions(node, player) do
    optimizer ← new InfoBasedOptimizer(region,
                                       −prevGuaranteeVector[otherPlayer])
    for i ← 1 to regionSamples(region) do
        point ← nextPoint(optimizer)
        move ← createMove(region.index, point)
        child ← nextTurn(makeMove(clone(node), move), player)
        child ← Info-Based-Alpha-Beta(child, otherPlayer, scoreGuaranteeVector,
                                      depth − 1)
        score ← −child.abScore
        if (bestMove = null or score > bestScore) then
            bestMove ← move
            bestScore ← score
            if (bestScore ≥ −prevGuaranteeVector[otherPlayer]) then
                goto prune
            if (bestScore > scoreGuaranteeVector[player]) then
                scoreGuaranteeVector[player] ← bestScore
        addData(optimizer, point, score)

prune:
node.abScore ← bestScore
node.bestMove ← bestMove
return node
```

Rather than write two procedures for the two players, Algorithm 9 uses negamax representation. Algorithm 9 takes as input the current search node and player, the guaranteed score bounds from previous search (represented as $(\alpha, -\beta)$), and the depth of search remaining. It returns the current node with search results (best score and move). If the node is at terminal search depth or is a leaf node, then we evaluate the node score (negated for the adversary) and return.

After initializing variables, we perform an information-based optimization on each action parameter region for a predefined sample limit. If, before we reach that sample limit, an evaluated subtree yields a score which indicates that a rational player will not allow play through the current node (i.e. the lower bound exceeds the upper bound), then all remaining search is unnecessary and we prune it.

For each information-based optimization, we pick a point in the action parameter region, create a move and child node resulting from that move, and perform a recursive call to search the subtree rooted at that child. The return results are negated because of our negamax representation; each player maximizes negated scores of the other player. If the return score is the best yet, we record it. If it also affects α or β, we update the guarantees and prune if appropriate. At the end of each iteration, we supply the return data to the information-based optimization for use in choosing a move for the next iteration.

Experimental results of DASAT Information-Based Alpha-Beta Search on the magnetic levitation problem are shown in Table 4.3. Figures 4.7, 4.8, and 4.9 show the control policies (mappings from position and velocity to current) resulting from searches to depths 2, 4, and 6, respectively. From the data, we can see that search execution is faster and pruning is greater than that achieved by random discretization. From the control policies, we see that the results are very similar to those achieved by uniform discretization. The quality of control policies will be explored further in the next section as we play these methods against one another.

One final important note about this chapter concerns a comparison to uniform discretization with node ordering. In practice, the heuristic of ordering subtree searches according to the preferred score/utility of child nodes can be a source of significant speedup. One might wonder when such a technique would be preferred to this

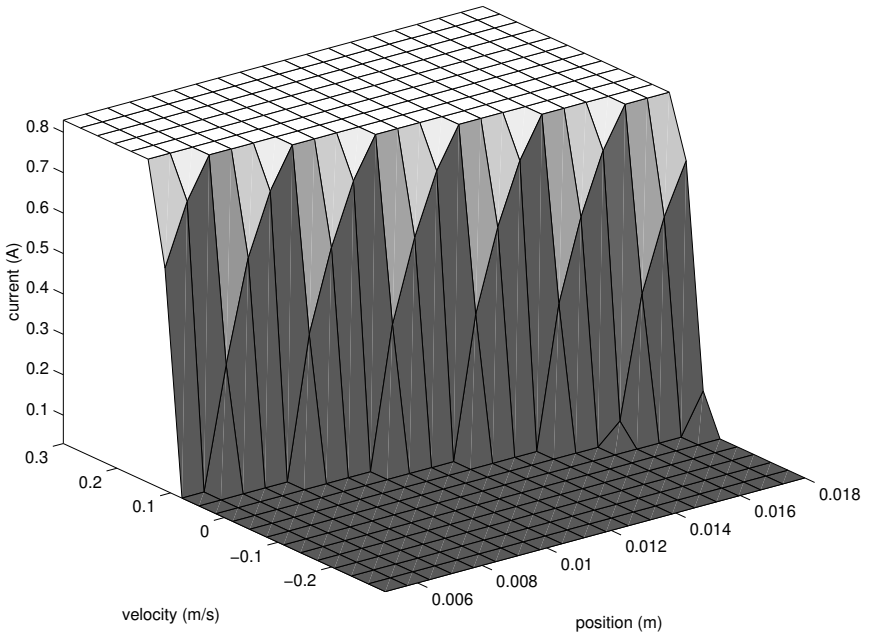

Figure 4.7: Maglev output currents from DASAT Information-Based Alpha-Beta, depth 2

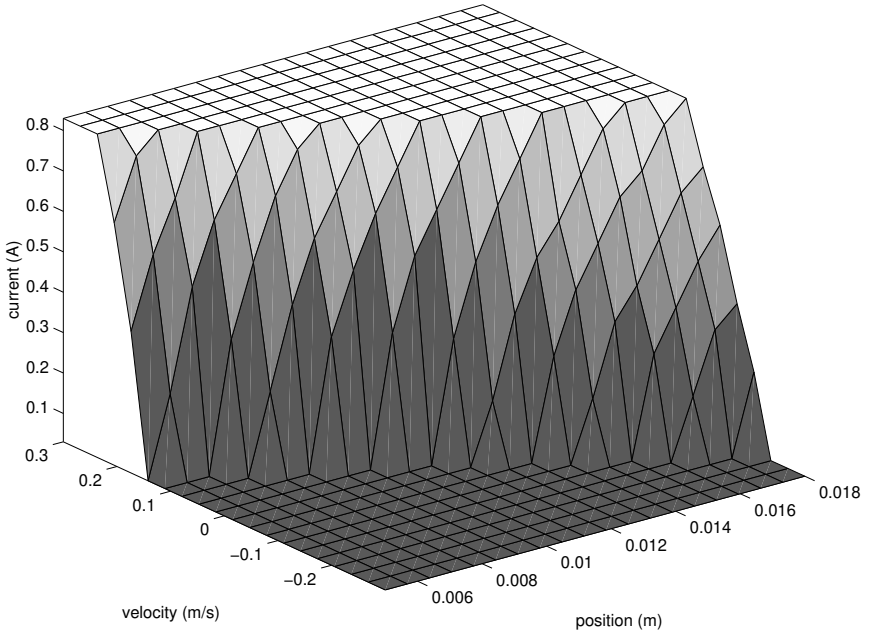

Figure 4.8: Maglev output currents from DASAT Information-Based Alpha-Beta, depth 4

Depth	Trials	Average Time (msec)	Average Nodes	Average Pct. Pruned	Average Nodes/Sec	Average Score
1	400	1	21	0.00	18,667	-1.58E-7
2	400	1	52	71.14	35,354	-1.58E-7
3	400	30	497	85.30	16,295	-3.31E-7
4	400	40	1,243	95.71	31,157	-3.31E-7
5	400	719	16,787	96.90	23,347	-5.26E-7
6	400	1,081	55,185	98.81	51,032	-5.26E-7

Table 4.3: Results for DASAT Information-Based Alpha-Beta Search

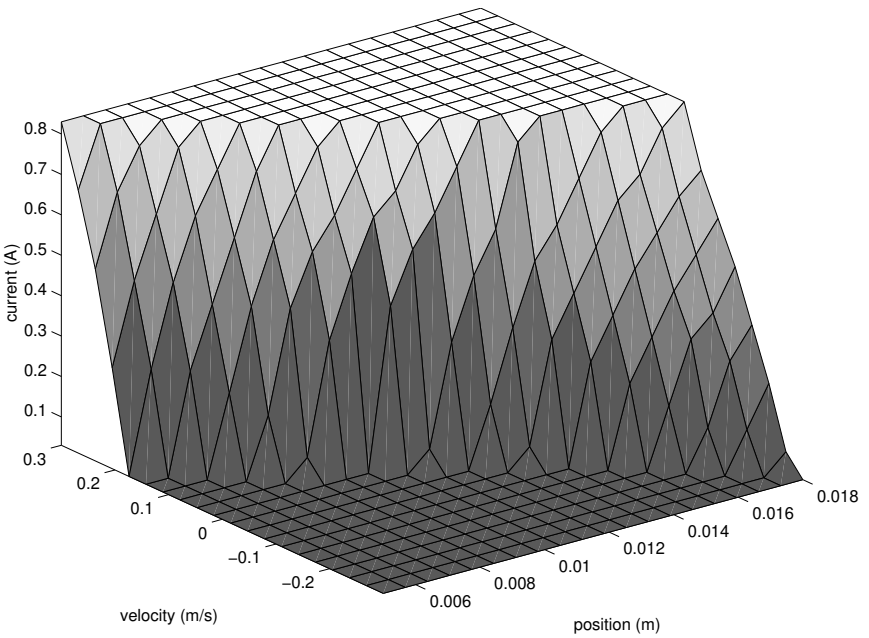

Figure 4.9: Maglev output currents from DASAT Information-Based Alpha-Beta, depth 6

information-based approach and vice versa. The answer is simple: If the problem domain is such that local scores are poor indicators of the relative quality of moves, then information-based optimization would be preferred. Information-based optimization chooses successive points based on full evaluations of subtrees so performance is not degraded by poor local information. However, if the local scores of immediate children provide good indication of the relative quality of moves, then uniform discretization with node ordering may be simpler and preferable.

4.6 Comparison of Methods

In comparing these algorithms to one another, let us first turn our attention towards effective branching factor reduction. The actual branching factor may vary considerably when searching to a fixed depth d. In the case of the maglev problem, the actual branching factor for a full search alternates between 20 and 8 on successive levels. One desires a simple means of comparing the effective branching of search given depth and node count.

The effective branching factor b is defined as the branching factor for which $1+b+b^2+\ldots+b^d$ equals the node count[34]. That is, b is the branching factor that effectively results in the same search node count for a given search depth. A comparison of effective branching factors for each algorithm on the maglev problem is given in Table 4.4.

	Effective Branching Factor b				% of Full b		
Depth	Random	Uniform	Info-Based	No Prune	Random	Uniform	Info-Based
1	20.00	20.00	20.00	20.00	100	100	100
2	7.58	10.09	6.66	12.93	59	78	52
3	8.72	12.16	7.56	14.66	60	83	52
4	6.46	8.93	5.66	12.78	51	70	44
5	7.16	9.39	6.78	13.81	52	68	49
6	6.17	7.83	5.99	12.74	48	61	47

Table 4.4: Comparison of Effective Branching Factor Reduction

Information-based Alpha-Beta Search yields significantly lower effective branching factors than alpha-beta with either random or uniform discretization. Uniform discretization yields the highest effective branching factors. As mentioned in the previous section, a node ordering heuristic would address this weakness for the maglev problem since local information is a good indicator of relative long-term quality of actions.

Previous experimentation is not adequate for comparing the relative quality of the resulting control policies. If any search happened to perform a good controller search and poor adversary search, it would appear to be a stronger game-tree search algorithm than it is. For this reason, we have played each algorithm against each other algorithm in order to give a true comparison of relative strength.

At each sampled position and velocity point in a uniform 20×20 grid, we play a game where each algorithm searches to depth four in choosing four successive moves. One algorithm chooses moves for the controller and the other chooses moves for the adversary. The two algorithms are switched and the process is repeated.

Results for random versus uniform discretization are given in Table 4.5. On average, search with random discretization takes 47% of the time taken using uniform discretization while searching 26% of the nodes. Negative player scores are trajectory costs. Search with uniform discretization yields lower cost trajectories on average and thus better quality play.

Player	Trials	Average Time (msec)	Average Nodes	Average Pct. Pruned	Average Nodes/Sec	Average Score
Random	400	99.87	2,946	89.83	29.50	-3.41E-7
Uniform	400	214.16	11,533	60.20	53.86	-3.31E-7

Table 4.5: Results for Random versus Uniform Discretization

Results for random versus information-based discretization are given in Table 4.6. On average, search with information-based discretization takes 67% of the time taken using random discretization while searching 92% of the nodes. Information-Based Alpha-Beta Search yields better play than Alpha-Beta with Random Discretization.

Player	Trials	Average Time (msec)	Average Nodes	Average Pct. Pruned	Average Nodes/Sec	Average Score
Random	400	99.94	2,950	89.82	29.52	-3.40E-7
Info-Based	400	66.76	2,722	90.61	40.77	-3.31E-7

Table 4.6: Results for Random versus Information-Based Discretization

Results for uniform versus information-based discretization are given in Table 4.7. On average, search with information-based discretization takes 34% of the time taken using uniform discretization while searching 26% of the nodes. Information-Based Alpha-Beta Search and Alpha-Beta Search with Uniform Discretization yield roughly equivalent quality play.

Player	Trials	Average Time (msec)	Average Nodes	Average Pct. Pruned	Average Nodes/Sec	Average Score
Uniform	400	218.99	11,341	60.87	51.79	-3.31E-7
Info-Based	400	73.68	2,924	89.91	39.68	-3.31E-7

Table 4.7: Results for Uniform versus Information-Based Discretization

4.7 Conclusions

In the beginning of this chapter, we formalized DASAT Hybrid System Games and DASAT Hybrid Systems Search Problems. We continued study of the magnetic levitation problem of Zhao, which takes a game-theoretic approach using an adversary to model worst-case effects of bounded model error, numerical simulation error, environmental perturbation, etc. In this chapter, we removed the assumption of having given action parameter region discretizations, and studied three different ways of dynamically discretizing action parameter regions.

Information-based alpha-beta is a novel application of information-based optimization which uses the α lower bound and β upper bound of alpha-beta search to

optimize for pruning. The resulting algorithm exceeded the good speed and pruning performance of random discretization while matching the control policy quality of uniform discretization.

It should be noted that uniform discretization with a node ordering heuristic should perform quite well in problem domains where local score information is a good long-term indicator of relative move quality. In contrast, Information-Based Alpha-Beta Search is not prone to poor local score information, as decisions are based on the results of full subtree search.

We next address hybrid system search problems where action timing discretizations are not given.

Chapter 5

SADAT Search

Extending discrete search to hybrid system search introduces two new decisions in optimization: action discretization and action timing discretization. In this chapter we choose to address the latter decision: How could a search algorithm choose when to branch the search tree and consider possible actions? We will thus assume that continuous action spaces are already discretized. From the perspective of the search algorithm, action discretizations are static, i.e. the search algorithm cannot affect the action discretization. However, from the perspective of the search algorithm, action timing discretizations are dynamic, i.e. branching points are chosen by the search algorithm. For this reason, we will call such searches "SADAT searches" as they have Static Action and Dynamic Action Timing discretization.

In this chapter, we will formally define a SADAT Hybrid System Game and its solitaire case, a SADAT Hybrid System Search Problem. A submarine detection avoidance problem is introduced as a focus for designing real-time control deliberation. We present iterative refinement, a new search algorithm perhaps most simply described as similar to iterative deepening search within a limited time interval. We also present a new variation on best-first search which allows for more flexible action timing. Then, we show how iterative refinement can work quite well under heuristic monotonicity and admissibility assumptions. Finally, we introduce ϵ-optimal Iterative Refinement Recursive Best-First Search.

5.1 SADAT Hybrid System Game and Search Problem

Formally, a SADAT Hybrid System Game is defined as a 7-tuple

$$\{S, s_0, A, p, l, m, d\}$$

where

- S is the hybrid state space with a finite number of finite discrete variable domains, and a finite-dimensional continuous space,
- $s_0 \in S$ is the initial state,
- A is the finite discrete action space,
- p is the number of players,
- $l : S \times \{1, \ldots, p\} \rightarrow \{a_1, \ldots, a_n\} \in A$ is a *legal move* function mapping from a state and player number to a finite set of legal actions which may be executed in that state by that player,
- $m : S \times A^p \rightarrow S \times \Re^p$ is a *move* function mapping from a state and simultaneous player actions to a resulting state and the utility of the combined actions for each player,
- $d : S \times \Re^+ \rightarrow S \times \Re^p$ is a *delay* function mapping from a state and non-negative time delay to the resulting state and the utility of the trajectory segment for each player. We require that $d(s, 0) = \{s, \{0, \ldots, 0\}\}$. Letting $d(s_1, t_1) = \{s_2, \{u_{1,1}, \ldots, u_{1,p}\}\}$ and $d(s_2, t_2) = \{s_3, \{u_{2,1}, \ldots, u_{2,p}\}\}$, we also require that $d(s_1, t_1 + t_2) = \{s_3, \{u_{1,1} + u_{2,1}, \ldots, u_{1,p} + u_{2,p}\}\}$.

The total utility of any finite trajectory is computed as the sum of the trajectory move and delay utilities. In this time-invariant formalism, time can easily be encoded in a continuous clock variable, and time invariant behavior could thus be easily achieved.

A *SADAT Hybrid System Search Problem* is a special case of the SADAT Hybrid System Game where we are interested in finding a trajectory from the initial state to a goal state. Usually such problems are stated in terms of path cost rather than utility. Formally, a SADAT Hybrid System Search Problem is defined as a 7-tuple

$$\{S, s_0, S_g, A, l, m, d\}$$

where

- S is a hybrid state space with a finite number of finite discrete variable domains, and a finite-dimensional continuous space,
- $s_0 \in S$ is an initial state,
- $S_g \subset S$ is a set of goal states,
- A is a finite discrete action space,
- $l : S \rightarrow \{a_1, \ldots, a_n\} \in A$ is a *legal move* function mapping from a state to a finite set of legal actions which may be executed in that state,
- $m : S \times A \rightarrow S \times \Re$ is a *move* function mapping from a state and action to a resulting state and cost of the action,
- $d : S \times \Re^+ \rightarrow S \times \Re^p$ is a *delay* function mapping from a state and non-negative time delay to the resulting state and the cost of the trajectory segment. We require that $d(s, 0) = \{s, \{0, \ldots, 0\}\}$. Letting $d(s_1, t_1) = \{s_2, \{u_{1,1}, \ldots, u_{1,p}\}\}$ and $d(s_2, t_2) = \{s_3, \{u_{2,1}, \ldots, u_{2,p}\}\}$, we also require that $d(s_1, t_1 + t_2) = \{s_3, \{u_{1,1} + u_{2,1}, \ldots, u_{1,p} + u_{2,p}\}\}$.

We next describe a SADAT Hybrid System Search Problem in the domain of submarine tactical planning for detection avoidance.

5.2 Submarine Channel Problem

The Submarine Channel Problem is not unlike a Sega™ video game of the 1980's called Frogger. A submarine seeks a path through a channel such that it avoids being detected by a number of patrolling ships.

5.2.1 The Submarine Tactical Planning Assistant

The choice of this problem is motivated by the submarine tactical planning assistance work of Thomas C. Smith and David P. Watson (Johns Hopkins Laboratory Applied Physics Laboratory (JHUAPL)) and Peter W. Jacobus (SONALYSTS, Inc.)[46]. The Generative Layer of their Tactical Planning Associate[46, § 2.4.2] uses Recursive Best-First Search (RBFS)[25] to "produce an ordered set of way-points that inscribe an optimal path through a field of predictably moving and stationary obstacles having arbitrary avoidance areas." See Figure 5.1 for a screenshot of the interface.

Further details of the problem representation were obtained through personal correspondence with Adam V. Peterson of JHUAPL. The action space is discretized with 8 headings and 3 speeds (full speed, half speed, stop). The action timing space is discretized as well according to a uniform simulation update interval. The problem is formulated as a discrete search.

Enemy vessels each have inner and outer detection radii. Within the circle defined by the vessel position and inner detection radius, the submarine is detected and penalized heavily. Beyond the circle defined by the outer detection radius, the submarine is safe from detection. Between the circles, probability of detection increases along with an associated penalty for such risk. Speed and patrol trajectories of enemy vessels are known a priori. There is neither uncertainty nor change in enemy vessel patrolling; this is a solitaire game of perfect information.

In using RBFS, the heuristic weight is set to 1.75, and the cost to the current state is the sum of the time to the current state and a penalty calculated if the submarine has passed within the outer radius of a ship.

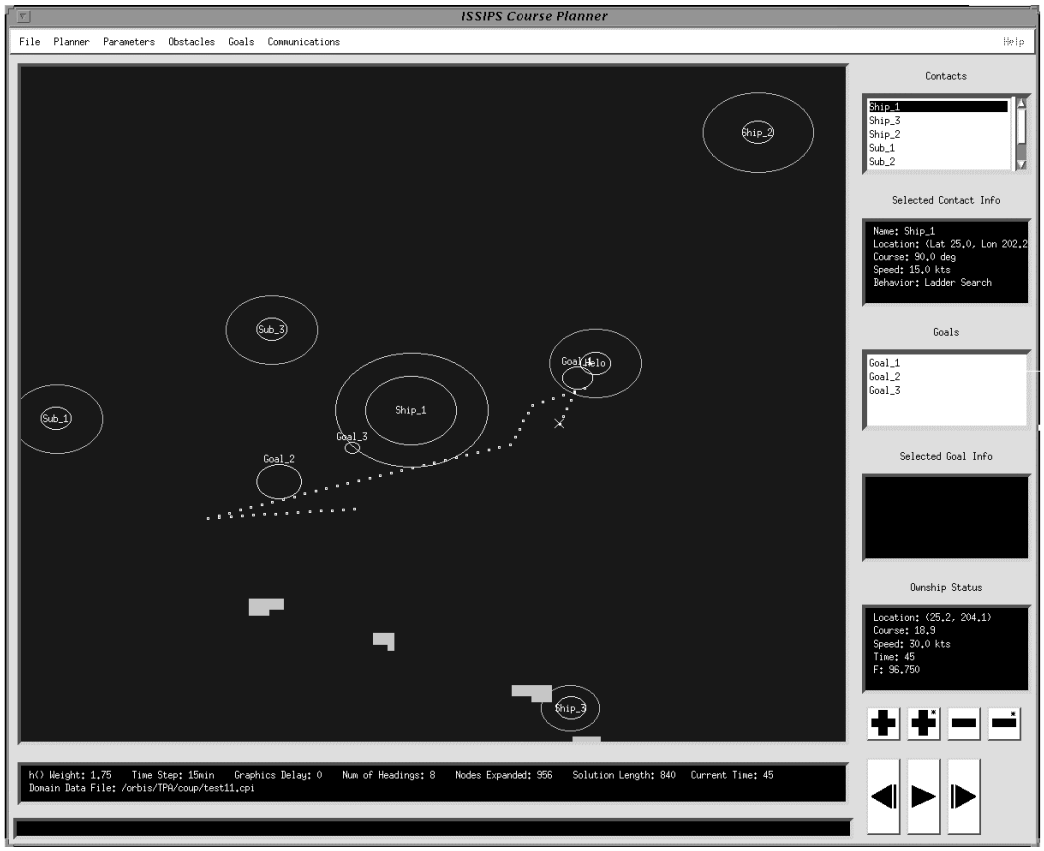

Figure 5.1: Tactical Planning Associate Man-Machine Interface illustrating Generative Layer, from [46, Figure 6]

5.2.2 The SADAT Submarine Channel Problem

We have chosen a specific class of submarine tactical planning problems for ease of adjusting difficulty. Just as the $n^2 - 1$ sliding tile puzzle has served as a benchmark for discrete search techniques, we have chosen a simple problem easily scaled and modified for greater difficulty.

In the *Submarine Channel Problem*, the submarine starts at position $(x, y) = (0, 0)$ with eastward heading and at full stop. To the east along an east-west channel of width w (centered along $y = 0$) are n ships patrolling across the width of the channel. This is pictured in Figure 5.2.

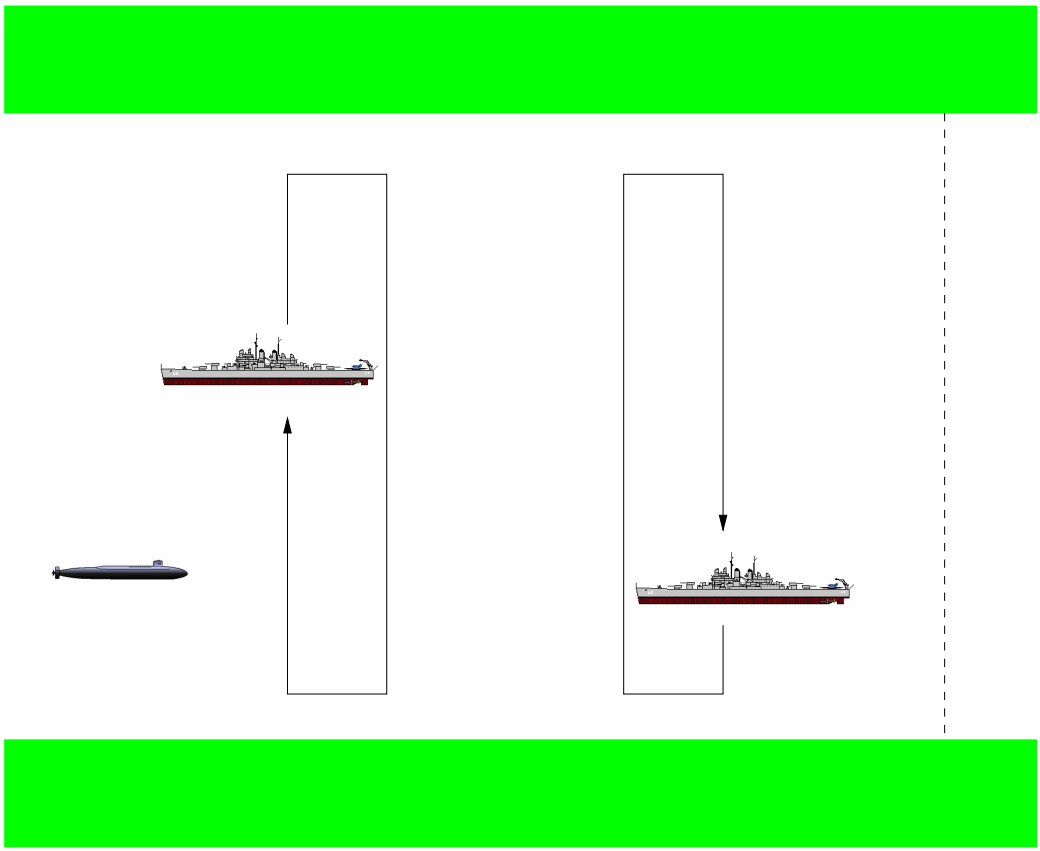

Figure 5.2: Submarine Channel Problem

Each ship j has an inner detection radius $r_{\mathrm{i},j}$ and an outer detection radius $r_{\mathrm{o},j}$. Within a proximity of $r_{\mathrm{i},j}$, ship j will detect the submarine and the submarine will be penalized with a detection penalty. Within a proximity of $r_{\mathrm{o},j}$ and beyond $r_{\mathrm{i},j}$, the submarine incurs a proximity penalty scaling linearly from 0 at the outer radius to the full detection penalty at the inner radius. Beyond the outer radius, there is no penalty. If the submarine collides with the sides of the channel, there is a collision penalty. In the case of collision or detection, the submarine is halted and allowed no further legal moves. The first ship patrols at an x-offset $\mathrm{xOffset}_1$ of $r_{\mathrm{o},1}$. Each ship i thereafter has $\mathrm{xOffset}_i = \mathrm{xOffset}_{i-1} + 3r_{\mathrm{i},i-1} + r_{\mathrm{i},i}$. Ship i has a patrolling route defined by cycling linearly between the following points: $(\mathrm{xOffset}_i, w/2 - r_{\mathrm{i},i})$,

$(\text{xOffset}_i + 2r_{\text{i},i}, w/2 - r_{\text{i},i})$, $(\text{xOffset}_i + 2r_{\text{i},i}, -w/2 + r_{\text{i},i})$, and $(\text{xOffset}_i, -w/2 + r_{\text{i},i})$. Each ship begins at a given percentage along this cycle. For n ships, the goal states are all states within the channel with $x > \text{xOffset}_i + 2r_{\text{i},n} + r_{\text{o},n}$, i.e. all channel points to the right of the rightmost outer detection radius.

The submarine can travel in 8 headings (multiples of $\pi/4$ radians), and 3 speeds: full speed, half speed, and full stop. Together these define 17 distinct actions the submarine can take at any point which it has incurred neither collision nor full detection penalty.[1] Each ship travels at a single predefined speed.

For this chapter, we have chosen $w = 1$ length unit. The outer radius of every ship is $0.2w$. The inner radius of each ship is $0.1w$. The maximum velocity of the submarine is $w/(1$ time unit$)$. All ship velocities are also $w/(1$ time unit$)$. Ships are started at random percentages through their patrol cycles. The detection penalty is set at 10000. Figure 5.3 shows a demonstration software animation frame from a solution to an instance of the 4-ship problem.

Since we use SADAT Iterative Refinement Search (§ 5.3) as a baseline for comparison, we chose a number of ships such that it would be challenging for Iterative Refinement to find a solution within 10 seconds in our experimental context. All programming was done in Java[2], and all experimentation was done in MS-DOS using a Dell Dimension XPS T450 with a 450 MHz Pentium CPU. It was found that the 10-ship problem (Figure 5.4) was sufficiently challenging for Iterative Refinement so as to serve as a useful challenge problem for SADAT and DADAT searches.

5.3 SADAT Iterative Refinement Search

In this section, we limit search to a fixed time horizon t_f. For these approaches, we start with the simplest of search trees over the time interval: a search tree of depth one with a root at the initial state, a branch for each legal action and leaves at $t = t_f$.

[1] Since we assume discrete, instantaneous changes to headings and speeds, all full stop actions are effectively equivalent.

[2] Programming was done with minimal optimization, since rapid prototyping and clarity were desired.

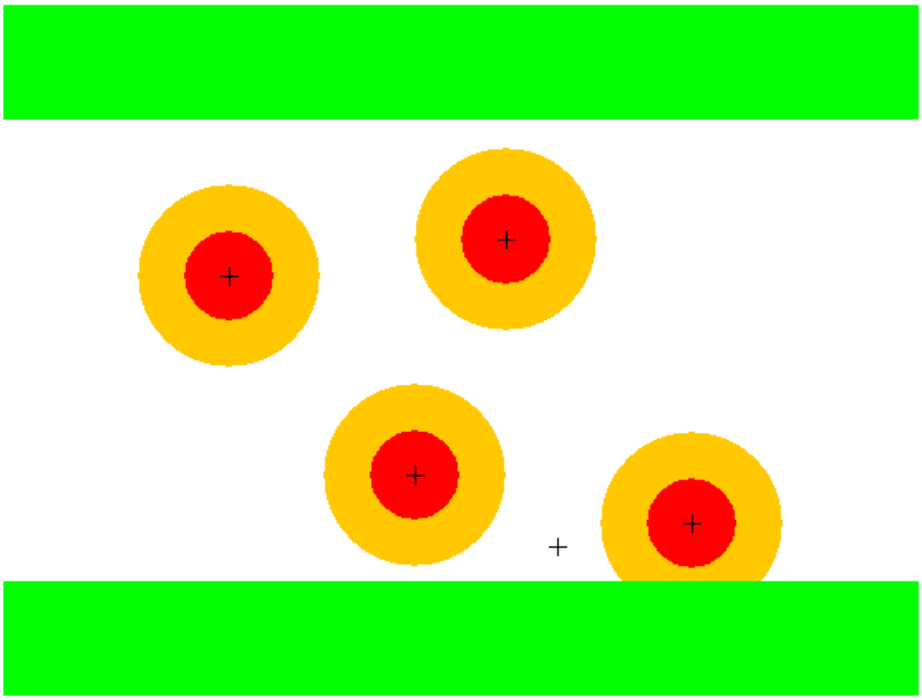

Figure 5.3: Submarine Channel Problem Demo, 4 Ships

This tree, pictured leftmost in Figure 5.5, represents the possible outcomes if the agent were to only act at $t = 0$.

With standard tree search techniques, a search tree is grown by expanding leaf nodes. One looks forward from leaf nodes to further inform one's action. Starting with our simple search tree, there is no need to look forward. We are evaluating all possible trajectories with respect to nodes at the search time horizon, and we have already looked forward to the search time horizon. Rather, we wish to look within. There are many ways one can choose action timings to search possible trajectories from $t = 0$ to $t = t_f$. We begin with a simple method called Iterative Refinement which is perhaps most simply described as similar to iterative deepening search within a limited time interval.

Like iterative deepening, Iterative Refinement consists of a series of searches. Each search is a depth-first search where the tree is branched at a set of time points. In

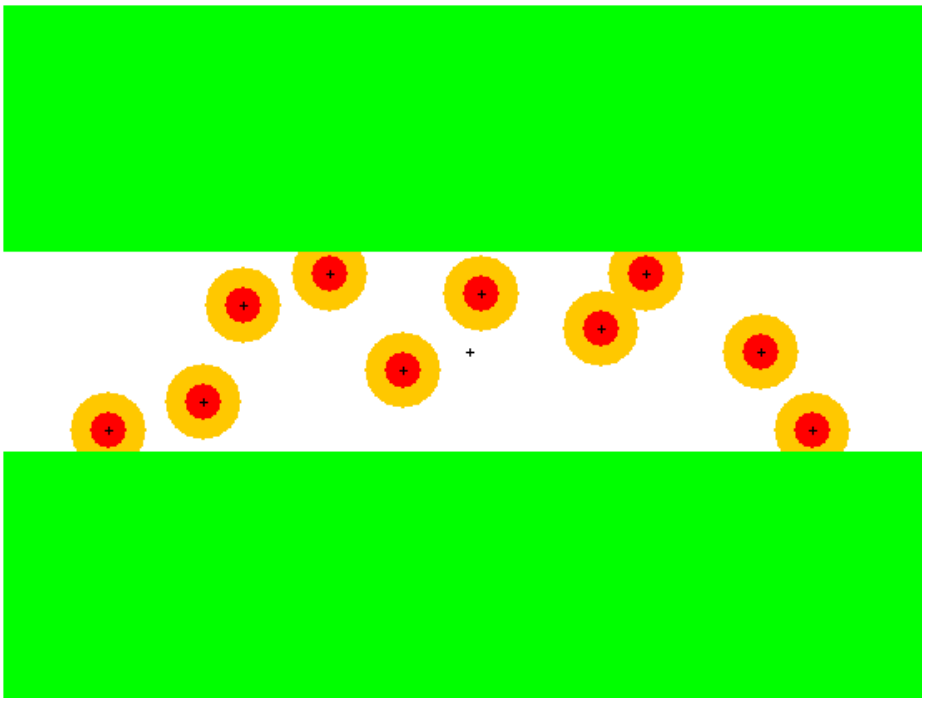

Figure 5.4: Submarine Channel Problem Demo, 10 Ships

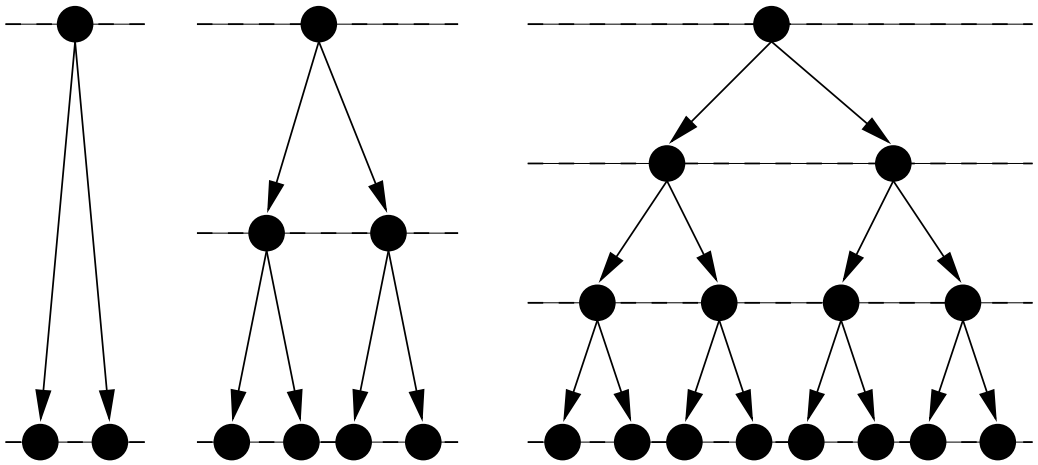

Figure 5.5: Iterative Refinement

the ith iteration, iterative refinement breaks the time interval $[0, t_f]$ into i equal time intervals and performs a search to depth i. The resulting search is pictured in Figure 5.5. The algorithm pseudocode is shown in Algorithms 10 and 11. It has the same computational time and space complexity as iterative deepening: $O(b^d)$ and $O(d)$ respectively, where b is effective branching factor, and d is maximum search depth.

Algorithm 10 SADAT Iterative Refinement Depth-First Search

```
SADATIterativeRefinementDFS(rootNode, initialDelay, refinementLimit)
▷ Input: root node,
         initial list of branching times,
         limit on number of refinement iterations
▷ Output: best leaf node at time horizon
bestNode ← null
refinement ← 1
while (not refinement > refinementLimit) do
    newBestNode ← SADAT-DFS(rootNode, initialDelay/refinement, refinement)
    if (bestNode = null or g(newBestNode) < g(bestNode)) then
        bestNode ← newBestNode
    refinement ← refinement + 1
return bestNode
```

Algorithm 11 SADAT Depth-First Search

```
SADAT-DFS(node, delay, depthLimit)
▷ Input: search node,
         simulation delay,
         depth of search below node
▷ Output: best subtree leaf node at time horizon
if (depthLimit = 0) then
    return node
bestNode ← null
foreach move m[i] of legalMoves(node) do
    child ← wait(makeMove(clone(node), m[i]), delay)
    newBestNode ← SADAT-DFS(child, delay, depthLimit − 1)
    if (bestNode = null or g(newBestNode) < g(bestNode)) then
        bestNode ← newBestNode
return bestNode
```

The results, shown in Table 5.1, are generally poor, ranging from 0 to 47 percent

depending on the given time horizon. While the rate of nodes/sec is relatively much higher than other approaches, the primary problem with such a search is that each iteration searches the full tree. The branching factor and effective branching factor of each search is the same. A lot of unnecessary search is done quickly, and the net result is weak.

Time Horizon	Results	% Goal	Time to Goal			Cost to Goal			Nodes/Sec
			Min	Avg	Max	Min	Avg	Max	
4.20	100	0	N/A	N/A	N/A	N/A	N/A	N/A	10,271
4.83	100	1	10.06	10.06	10.06	4.82	4.82	4.82	9,232
5.46	100	14	10.02	10.04	10.08	4.79	5.17	5.45	8,329
6.09	100	15	10.02	10.04	10.06	4.99	5.44	5.85	7,955
6.72	100	47	10.02	10.04	10.08	5.07	6.11	6.69	7,303
7.35	100	0	N/A	N/A	N/A	N/A	N/A	N/A	7,831

Table 5.1: Results for SADAT Simple Iterative Refinement DFS

If we modify Algorithm 10 such that search terminates as soon as a goal node is found, we observe the results shown in Table 5.2. Although search returns with a goal node much more frequently, the utility of the trajectory to the goal node is generally poor. On average the submarine incurs high proximity penalties along the trajectory. Without goal node termination, the algorithm returns the lowest cost trajectory to the time horizon for the entire iterated search. Iterative refinement depth-first search with goal node termination offers no such solution quality guarantee. All future algorithms of this chapter have some form of solution quality guarantee.

Simple iterative refinement search is presented as a baseline for comparison for the SADAT search techniques that follow. In each successive subsection, we make a tradeoff of assumed a priori knowledge versus performance.

5.4 SADAT Best-First Search

In this section, we introduce a novel variation of Best-First Search (BFS) which allows limited flexibility in varying action timing. We begin by describing a simplified version

Time Horizon	Results	% Goal	Time to Goal Min	Time to Goal Avg	Time to Goal Max	Cost to Goal Min	Cost to Goal Avg	Cost to Goal Max	Nodes/Sec
4.20	100	4	0	0.01	0.03	2,090	5,745	8,911	9,470
4.83	100	33	0	1.23	4.93	5	6,661	9,906	8,608
5.46	100	84	0	1.01	9.05	5	6,313	10,001	7,646
6.09	100	89	0	2.22	9.91	5	6,851	9,927	7,234
6.72	100	100	0	1.62	7.32	5	6,714	10,000	6,513
7.35	100	60	0	0.73	2.92	2,090	7,793	9,996	7,362

Table 5.2: Results for SADAT Simple Iterative Refinement DFS with Goal Node Termination

of the algorithm in order to communicate both key concepts of the search and the reason for the limitation in timing flexibility.

As BFS is a heuristic search, we assume the existence of a heuristic evaluation function to estimate the cost from any state to a goal state. Such information is used to make the search *selective*, i.e. to direct search in the direction which is estimated to have the "optimal" solution. The term "optimal" may be rightly used in a discrete setting, but in this continuous problem domain, the search is generally incomplete and therefore at most an approximation to optimal behavior. Theoretically, given unbounded computing resources, as the step-size approaches zero, an admissible (underestimating) heuristic function would give a solution approaching the optimal solution.

For the Submarine Channel Problem, there is a very simple heuristic estimate of cost to goal state: the x distance to the end of the patrolled region divided by the maximum submarine speed.

5.4.1 Simple SADAT Best-First Search

A detailed description of Best-First Search (BFS) can be found in [41, § 4.1]. A function f' is defined over all nodes as the sum of the cost function g and the heuristic function h'. Whereas $g(n)$ is the path cost from the root node to n, $h'(n)$ is an estimate of the minimum cost from n to a goal node. For each node n, $f'(n) = g(n) + h'(n)$.

The accents of f' and h' indicate that they are estimates of the unknown actual evaluation functions f and h. Starting with a heap containing only the root, best-first search iteratively selects the minimum node according to f' and checks to see if that node is a goal node. If so, it terminates. If not, it evaluates all children of the node, places them in the heap, and repeats the process.

In our variation of BFS, we (1) assume a given largest time-step between actions, and (2) redefine node expansion to allow new open nodes along existing branches. Regarding (1), we take as a parameter Δt, a real-valued number of time units, which serves as a default delay time between an expanded node and its new leaf children. Regarding (2), we redefine node expansion for three cases: the root node case, leaf node case, and internal node case. These cases correspond respectively to a node having no parent and no children, having a parent and no children, and having a parent and a child. One can prove inductively that these are the only three cases which can occur for our method of expansion.

Simple SADAT Best-First Search pseudocode is given in Algorithms 12–15 . It begins as normal BFS with the root node in the open heap. With each iteration, the node with the lowest f'(node) is extracted from the heap. If the node is a goal node, the algorithm terminates with success. Otherwise, its children are generated and placed on the open heap. The key difference is how new nodes are generated.

For a root node, we simply generate its children. Each child is computed by cloning its parent, making the associated legal move, and simulating forward Δt. The child is then placed in a heap according to f'(child). This is pictured in the first transition of Figure 5.6.

For a leaf node, there is a slight difference. In addition to generating its children, we also generate a new parent node halfway (with respect to time delay) between the leaf node and its current parent node. This is pictured in the second transition of Figure 5.6.

For an internal node, there is yet another difference. In addition to generating new children, i.e. all children but its single existing child, and a new parent (as with the leaf node), it generates a new child halfway between itself and its pre-existing child. This is pictured in the third transition of Figure 5.6.

Algorithm 12 SADAT Simple Best-First Search

```
SADAT-SIMPLE-BFS(root)
▷ Input: root node
▷ Output: goal node if one exists, otherwise no termination
node ← root
node.parent ← null
node.child ← null
while (not isGoal(node)) do
    if (node.parent = null) then
        ▷ Root node case
        simple-expand-root(node, empty-heap)
    else
        if (node.child = null) then
            ▷ Leaf node case
            simple-expand-leaf(node, heap)
        else
            ▷ Internal node case
            simple-expand-internal-node(node, heap)
    node ← extractMin(heap)
return node
```

Algorithm 13 Simple Expansion of Root

```
SIMPLE-EXPAND-ROOT(node, heap)
▷ Input: root node,
         heap of unexpanded nodes
▷ Output: none
foreach move m[i] of legalMoves(node) do
    child ← wait(makeMove(clone(node), m[i]), delay)
    child.parent ← node
    child.child ← null
    child.previousDelay ← delay
    insert(heap, child, f(child))
```

Algorithm 14 Simple Expansion of Leaf

```
SIMPLE-EXPAND-LEAF(node, heap)
▷ Input: leaf node,
         heap of unexpanded nodes
▷ Output: none
foreach move m[i] of legalMoves(node) do
    child ← wait(makeMove(clone(node), m[i]), delay)
    child.parent ← node
    child.child ← null
    child.previousDelay ← delay
    insert(heap, child, f(child))
newParent ← wait(clone(node.parent), node.previousDelay/2)
newParent.parent ← node.parent
newParent.child ← node
newParent.previousDelay ← node.previousDelay/2
node.parent.child ← newParent
node.parent ← newParent
insert(heap, newParent, f(newParent))
```

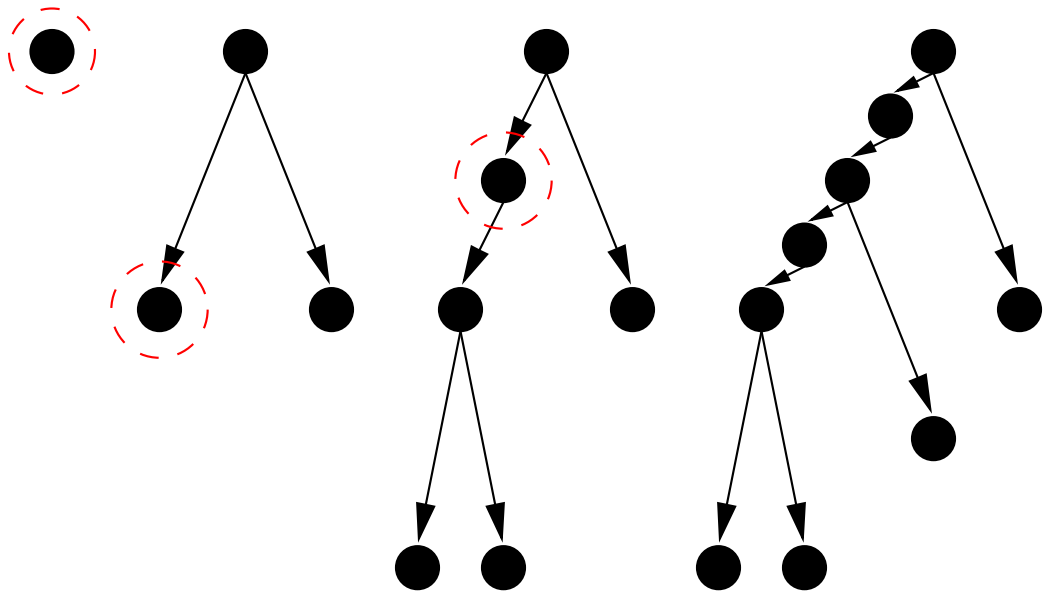

Figure 5.6: SADAT Best-First Search

Algorithm 15 Simple Expansion of Internal Node

```
SIMPLE-EXPAND-INTERNAL-NODE(node, heap)
▷ Input: internal node,
         heap of unexpanded nodes
▷ Output: none
foreach non-null move m[i] of legalMoves(node) do
    child ← wait(makeMove(clone(node), m[i]), delay)
    child.parent ← node
    child.child ← null
    child.previousDelay ← delay
    insert(heap, child, f(child))
newParent ← wait(clone(node.parent), node.previousDelay/2)
newParent.parent ← node.parent
newParent.child ← node
newParent.previousDelay ← node.previousDelay/2
node.parent.child ← newParent
node.parent ← newParent
insert(heap, newParent, f(newParent))
newChild ← wait(clone(node), node.child.previousDelay/2)
newChild.parent ← node
newChild.child ← node.child
newChild.previousDelay ← node.child.previousDelay/2
node.child.parent ← newChild
node.child ← newChild
insert(heap, newChild, f(newChild))
```

The first important thing to note about this algorithm is that it allows a more refined temporal search than best-first search with a fixed delay. This is both a strength and a weakness under different circumstances. While it can sometimes better approximate optimal solutions or find solutions which cannot be found without such refinement, one can easily generate pathological cases where SADAT Simple Best-First Search cannot find solutions which *can* be found using best-first search with a fixed delay.

The second important thing to note is one such significant pathological case which motivates the final piece of the full algorithm. Suppose we have the case where our cost function g *monotonically increases* along any path of the search tree, and our function f' always *underestimates* actual cost to a goal node through any non-goal node. Without looking far, we easily find an example: any submarine channel problem with $h'(n) = 0$ for all n.

Given an f' with such characteristics, then for any open (non-expanded) node n_1 preceding another open node n_2 along a path, $f'(n_1) < f'(n_2)$. Put simply, earlier possibilities always look better along a path in the tree. The ramification of this fact and our method of node expansion, is that this case will result in infinite refinement from a root child back toward the root.

Given these characteristics, the best node generated by the best root child will be the new parent between the root and that child. The best node generated by the new parent will be its new parent, and so forth infinitely. Clearly, such a method has need of some means to restrict path refinement so that such infinite refinement does not trap the search in a local minimum.

5.4.2 SADAT Best-First Search with Refinement Limits

One simple means of restricting refinement is to limit the number of refinements performed along any path. More specifically, we keep count of the number of times a new internal node was introduced in order to make a given path possible. Algorithmically, we associate with each node n a refinement level *n.refinementLevel*. The root has a refinement level of 0. A new leaf child inherits the refinement level of its parent. A new

internal node n' generated by node n has a refinement level of $n.refinementLevel + 1$.

The full algorithm of SADAT Best-First Search (Algorithms 16–19) is Simple SADAT Best-First Search augmented with the node refinement levels and the restriction that new nodes with refinement levels which would exceed a given refinement limit are not generated. The worst-case computational time and space complexity of SADAT Best-First Search is bounded by that of a Best-First Search performed on the full SADAT Best-First Search tree with maximal refinement. If f never overestimates the cost to a goal node, then Best-First Search is called A^* and is known to be both optimal[9] and complete[3][41] in searching the tree. However, computational time complexity is still exponential unless error in the heuristic function has a growth rate less than the logarithm of the actual path cost[35]. However, the most important complexity issue for modern computing is that of computational space complexity. Exponential growth of the heap exhausts memory resources in little time for modern computers. One way of dealing with exponential complexity is use of recursive best-first search, which is discussed in Section 5.6.

Results for the 10-Ship Submarine Channel Problem are shown in Table 5.3. For these trials, Δt was arbitrarily set to 1/4 of the initial distance to goal divided by the maximum submarine speed. The general tradeoff to note here is that of quality versus speed of solution. While more refinement yields better average solutions, fewer such solutions are found within the allotted 10-second time limit.

Refinement Limit	Results	% Goal	Time to Goal			Cost to Goal			Nodes/Sec
			Min	Avg	Max	Min	Avg	Max	
0	100	82	0.01	1.31	9.60	4.88	7.14	9.17	268
1	100	77	0.02	1.24	8.32	4.88	6.75	8.15	418
2	100	78	0.03	1.71	7.12	4.88	6.36	7.44	458
3	100	57	0.06	1.75	6.09	4.81	5.92	6.64	455

Table 5.3: Results for SADAT Best-First Search, $\Delta t = 1.05$

What this data does not show is how sensitive the performance is to the choice of

[3]Completeness is proven on locally finite graphs.

Algorithm 16 SADAT Best-First Search

```
SADAT-BFS(root)
▷ Input: root node
▷ Output: goal node if one exists, otherwise no termination
node ← root
node.parent ← null
node.child ← null
node.refinementLevel ← 0
while (not isGoal(node)) do
    if (node.parent = null) then
        ▷ Root node case
        expand-root(node, empty-heap)
    else
        if (node.child = null) then
            ▷ Leaf node case
            expand-leaf(node, heap)
        else
            ▷ Internal node case
            expand-internal-node(node, heap)
    node ← extractMin(heap)
return node
```

Algorithm 17 Expansion of Root

```
EXPAND-ROOT(node, heap)
▷ Input: root node,
         heap of unexpanded nodes
▷ Output: none
foreach move m[i] of legalMoves(node) do
    child ← wait(makeMove(clone(node), m[i]), delay)
    child.parent ← node
    child.child ← null
    child.previousDelay ← delay
    child.refinementLevel ← node.refinementLevel
    insert(heap, child, f(child))
```

Algorithm 18 Expansion of Leaf

```
EXPAND-LEAF(node, heap)
▷ Input: leaf node,
         heap of unexpanded nodes
▷ Output: none
foreach move m[i] of legalMoves(node) do
    child ← wait(makeMove(clone(node), m[i]), delay)
    child.parent ← node
    child.child ← null
    child.previousDelay ← delay
    child.refinementLevel ← node.refinementLevel
    insert(heap, child, f(child))
newParent ← wait(clone(node.parent), node.previousDelay/2)
newParent.parent ← node.parent
newParent.child ← node
newParent.previousDelay ← node.previousDelay/2
newParent.refinementLevel ← node.refinementLevel + 1
node.parent.child ← newParent
node.parent ← newParent
insert(heap, newParent, f(newParent))
```

Algorithm 19 Expansion of Internal Node

EXPAND-INTERNAL-NODE(*node*, *heap*)
▷ **Input:** internal node,
heap of unexpanded nodes
▷ **Output:** none
foreach non-null move *m[i]* of legalMoves(*node*) **do**
    child ← wait(makeMove(clone(*node*), *m[i]*), *delay*)
    child.parent ← *node*
    child.child ← **null**
    child.previousDelay ← *delay*
    child.refinementLevel ← *node.refinementLevel*
    insert(*heap*, *child*, f(*child*))
newParent ← wait(clone(*node.parent*), *node.previousDelay*/2)
newParent.parent ← *node.parent*
newParent.child ← *node*
newParent.previousDelay ← *node.previousDelay*/2
newParent.refinementLevel ← *node.refinementLevel* + 1
node.parent.child ← *newParent*
node.parent ← *newParent*
insert(*heap*, *newParent*, f(*newParent*))
newChild ← wait(clone(*node*), *node.child.previousDelay*/2)
newChild.parent ← *node*
newChild.child ← *node.child*
newChild.previousDelay ← *node.child.previousDelay*/2
newChild.refinementLevel ← *node.refinementLevel* + 1
node.child.parent ← *newChild*
node.child ← *newChild*
insert(*heap*, *newChild*, f(*newChild*))

Δt. Looking at Tables 5.4 and 5.5, we see that performance is very dependent on the choice of Δt.

Refinement Limit	Results	% Goal	Time to Goal			Cost to Goal			Nodes/Sec
			Min	Avg	Max	Min	Avg	Max	
0	100	100	0.01	0.40	7.09	5.10	7.08	10.99	186
1	100	100	0.02	0.43	7.29	5.10	6.83	9.59	353
2	100	99	0.00	0.70	7.75	5.10	6.53	8.60	359
3	100	97	0.03	1.03	7.57	5.10	6.31	7.62	359

Table 5.4: Results for SADAT Best-First Search, $\Delta t = 1.40$

Refinement Limit	Results	% Goal	Time to Goal			Cost to Goal			Nodes/Sec
			Min	Avg	Max	Min	Avg	Max	
0	250	0.8	0.04	0.04	0.04	7.22	7.60	7.98	348
1	250	0.8	0.05	0.07	0.09	7.22	7.60	7.98	359
2	250	1.6	0.12	1.70	6.11	6.51	8.05	10.48	109
3	250	2.0	0.13	1.02	4.07	5.05	6.56	8.78	215

Table 5.5: Results for SADAT Best-First Search, $\Delta t = 1.51$

SADAT Best-First Search provides a novel means of finding better solutions than can be found with Best-First Search with a fixed delay. This comes at a cost of time to solution, however, so that this algorithm is better suited to offline applications than real-time control. It should also be noted that both of these best-first search algorithms have exponential computational space complexity.

5.5 SADAT Iterative Refinement with Strong Pruning, Node Ordering, and Upper Bound

In previous experimentation with Iterative Refinement, we saw that performance was poor, but not as sensitive to choice of time horizon as SADAT Best-First Search. As

long as the goal was within the time horizon and the time horizon did not extend too far, the algorithm was more forgiving of an uninformed parameter choice.

In this section, we introduce a variant of Iterative Refinement which trades off generality for performance. By making a few simple assumptions about our problem domain for pruning, and applying heuristic node ordering, we achieve considerable speedup. The main novelty lies in how information from one iteration is used for pruning in the next.

Weak and Strong Pruning: Unlike iterative deepening and other standard search algorithms, the root node evaluation we are approximating through search is the minimum f'-value of all nodes on the horizon. After the first path to a leaf is searched, we have a best path ending with a best leaf n_{best}.

If we assume that our cost function g is monotonically increasing, then we can prune subtrees rooted at any node n such that $g(n) > f'(n_{\text{best}})$. Further, such pruning conditions can be carried from one iteration to the next, since *all searches are with respect to the same time horizon.* Put simply, each better path we find focuses the search thereafter through all iterations.

In this context, we refer to the assumption that g is monotonically increasing as a "weak" assumption. We refer to the associated pruning as "weak" pruning. The stronger assumption that can be made is that f' is monotonically increasing. Then we can prune subtrees rooted at any node n such that $f'(n) > f'(n_{\text{best}})$. We refer to this assumption and pruning as "strong".

Node Ordering: A standard technique for speeding up search is called *node ordering.* The basic intuition is that one orders the expansion of nodes in such a way as to have greater probability of finding a goal node sooner. In order for the cost of such ordering to be beneficial, the ordering technique must incur little computational cost. A common technique which is used here is to simply expand a node's children in increasing order of their f'-values. Note that this heuristic complements our desire to increase pruning.

Upper Bound: Finally, we note that for this problem domain, not every solution is a good solution. While the simulator halts the movement of the submarine when it passes within any inner radius of a ship, it does not halt the submarine when it

has passed within the outer radius and received a proximity penalty. Thus, some solutions are poor solutions.

Specifying an allowable upper bound on solution cost not only ensures that Iterative Refinement will not stop with an undesired solution, it also aids search by providing pruning conditions from the beginning of search.

Iterative Refinement with Strong Pruning, Node Ordering, and Upper Bound is described in pseudocode in Algorithms 20 and 21.

Algorithm 20 SADAT Iterative Refinement with Strong Pruning, Node Ordering, and Upper Bound

```
SADATIRwSPNOUB(rootNode, initialDelay, refinementLimit, upperBound)
▷ Input: root node,
         initial list of branching times,
         limit on number of refinement iterations,
         upper bound on solution cost
▷ Output: goal node with cost beneath upper bound if found,
          best leaf node found otherwise
globalUpperBound ← upperBound
globalGoalFound ← false
globalBestNode ← null
refinement ← 1
while (not globalGoalFound and not refinement > refinementLimit) do
    SADAT-DFS-SPNOUB(rootNode, initialDelay/refinement, refinement)
    refinement ← refinement + 1
return globalBestNode
```

Trials for the 10-Ship Submarine Channel Problem were performed with an upper bound cost of 10. This would mean that allowable solutions could only pass a very small amount within the outer radius of a ship on the way to a solution. Results are given in Table 5.6.

One key point to observe from these results is the tradeoff of generality in the form of domain knowledge for performance. However, this tradeoff should be made when it can, as such assumptions about f' can often be either proven or enforced in the design of f'. Solutions tend to be found more quickly with this technique than other techniques seen so far, so it is well suited to real-time tactical planning assistance. Compared to the computational gains, we have traded off little in the way

Algorithm 21 SADAT Depth-First Search with Strong Pruning, Node Ordering, and Upper Bound

```
SADAT-DFS-SPNOUB(node, delay, depthLimit)
▷ Input: search node, simulation delay, and depth of search below node
▷ Output: none
if (isGoal(node)) then
    globalGoalFound ← true
    globalBestNode ← node
    return
if (depthLimit = 0 or numOfChildren(node) = 0) then
    if (f(node) < f(globalBestNode)) then
        globalBestNode ← node
    return
foreach move m[i] of legalMoves(node) do
    child[i] ← wait(makeMove(clone(node), m[i]), delay)
Sort child[i] in increasing order of f(child[i])
i ← 1
done ← false
while (not done and not globalGoalFound) do
    ▷ Do not expand a node with f-value exceeding the global upper bound
    if (f(child[i]) > globalUpperBound) then
        done ← true
    else
        SADAT-DFS-SPNOUB(child[i], delay, depthLimit − 1)
    i ← i + 1
    if (i > numOfChildren(node)) then
        done ← true
return
```

Time Horizon	Results	% Goal	Time to Goal			Cost to Goal			Nodes/Sec
			Min	Avg	Max	Min	Avg	Max	
4.20	500	0.0	N/A	N/A	N/A	N/A	N/A	N/A	938
4.83	500	57.4	0.06	2.08	10.40	4.29	4.68	4.83	598
5.46	500	88.2	0.04	1.73	10.06	4.40	5.05	5.45	411
6.09	500	93.6	0.11	2.21	10.30	4.40	5.46	6.09	315
6.72	500	95.6	0.07	1.35	10.20	4.73	6.01	6.72	281
7.35	500	92.8	0.06	1.87	10.46	4.99	6.36	7.35	281

Table 5.6: Results for SADAT Iterative Refinement with Strong Pruning, Node Ordering, and Upper Bound

of generality.

Another key point to observe are the conditions under which the algorithm can reliably find a solution. We must choose an appropriate time horizon for which solutions are not so rare that our search is likely to find one. From this data one might think that one has only to choose a large enough time horizon to guarantee good results. However, it is also the case that one can choose too large a time horizon. Considering this Submarine Channel Problem, assuming that there is no straight-line solution through the patrolling ships, then there is a search delay parameter above which no solution exists. Given a time limit, one may set the time horizon sufficiently high as to have all search within the time limit performed with delay parameters too high to find a solution. Put simply, if the time horizon is too high, then the granularity of search is too high, and there is a performance penalty.

5.6 SADAT Iterative Refinement with Recursive Best-First Search

In Section 5.4, we saw that Best-First Search techniques have unfavorable, exponential space complexity. In [25], Richard Korf introduced a linear space complexity algorithm called Recursive Best-First Search (RBFS) which expands new nodes in same order as Best-First Search and thus has the same optimality guarantees. RBFS was the technique of choice for the Submarine Tactical Planning Assistant described in Section 5.2.1.

In this Section, we introduce an approximately optimal version of RBFS for SADAT problems, called SADAT ϵ-RBFS. We show that its performance is very sensitive to the input delay parameter. We then introduce SADAT Iterative Refinement ϵ-RBFS. Compared to other general-applicability SADAT algorithms which do not require a monotonicity assumption, SADAT Iterative Refinement ϵ-RBFS yields the best behavior with the least sensitivity to initial parameters.

5.6.1 SADAT ϵ - Recursive Best-First Search with Fixed Delay

In order to apply Recursive Best-First Search (RBFS) to continuous domains, there are two issues which must first be addressed. The first concerns action timing discretization. In this section, we choose the simplest solution and assume that for any call to RBFS, a fixed delay is used to generate children.

The second issue to address is the nature of floating point node evaluations. This was not an issue in Best-First Search, because nodes are only expanded once. RBFS uses a local cost threshold for each recursive depth-first search call. The cost threshold is updated using the least cost value of frontier nodes beyond the threshold. If the same subtree is searched again, it is with this updated value. In this way, nodes are expanded in best-first order, using a depth-first technique which can expand the same node many times. This is a tradeoff of computational time for space.

The fact that so many nodes will have distinct floating-point costs means that nodes will be expanded many times more than in discrete domains where evaluations are integer-valued and in a concentrated distribution. This same issue arises when applying iterative deepening search to continuous domains.

The way this issue is dealt with for iterative deepening techniques in complex domains is to increase the iterative deepening cost limit by a fixed amount ϵ on each iteration. Then the total number of iterations is proportional to $1/\epsilon$ and the algorithm is called ϵ-admissible[41, § 4.3, IDA*].

We can do something similar for RBFS. When each subtree is searched and the child is replaced in the heap, we make sure that its evaluation is increased by at least ϵ. ϵ-RBFS is given in pseudocode in Algorithm 22.

The result of applying ϵ-RBFS to the 10-Ship Submarine Channel Problem is shown in Table 5.7. Observing these results, one is struck by the extreme sensitivity of the search success to the fixed delay parameter.

Algorithm 22 SADAT ϵ - Recursive Best-First Search

SADATeRBFS(*node*, *nodeF*, *bound*, *delay*, *espilon*)
▷ **Input:** node, calling stored search value of node, local cost upper bound, simulation delay, and epsilon minimum bound increment
▷ **Output:** return stored search value of node
if (f(*node*) > *bound*) **then**
    return f(*node*)
if (isGoal(*node*)) **then**
    goalNode ← *node*
    exit algorithm
if (numOfChildren(*node*) = 0) **then**
    return ∞
foreach move *m[i]* of legalMoves(*node*) **do**
    c[i] ← wait(makeMove(clone(*node*), *m[i]*), *delay*)
    if (f(*node*) < *nodeF*) **then**
        cF[i] ← max(*nodeF*, f(*c[i]*))
    else
        cF[i] ← f(*c[i]*)
    insert(*heap*, *c[i]*, *cF[i]*)
{*c*, *cF*} ← extractMin(*heap*)
while (*cF* ≤ *bound* and *cF* < ∞) **do**
    ▷ The new local upper bound must increase by at least epsilon.
    if (numofChildren(*node*) > 1) **then**
        cF ← max(SADATeRBFS(*c*, *cF*, min(*bound*, minValue(*heap*))), *cF* + *epsilon*)
    else
        cF ← max(SADATeRBFS(*c*, *cF*, *bound*), *cF* + *epsilon*)
    insert(*heap*, *c*, *cF*)
    {*c*, *cF*} ← extractMin(*heap*)
return *cF*

5.6.2 SADAT Iterative Refinement with ϵ - Recursive Best-First Search

The sensitivity of the success of ϵ-RBFS to the delay parameter motivates an attempt to use ϵ-RBFS with different delays. In this section, we apply the idea of iterative refinement to ϵ-RBFS and find that the resulting algorithm has excellent performance across a broad range of initial parameters.

In Section 3.2 of [25], Korf directs the user of RBFS to make a top-level call to RBFS with an upper bound of ∞. Indeed, an upper bound of ∞ makes perfect sense

Delay	Results	% Goal	Time to Goal Min	Time to Goal Avg	Time to Goal Max	Cost to Goal Min	Cost to Goal Avg	Cost to Goal Max	Nodes/Sec
1.00	500	99.4	0.00	0.41	10.02	4.70	6.28	8.99	496.12
1.25	500	91.0	0.01	0.29	4.93	5.01	7.13	9.90	423.40
1.50	500	0.2	0.05	0.05	0.05	7.95	7.95	7.95	349.71
1.75	500	0.4	0.03	0.06	0.10	5.95	7.70	9.45	234.24

Table 5.7: Results for SADAT ϵ - Recursive Best-First Search, $\epsilon = 0.25$

when one has only one possible search space. In our case, we have infinite ways of discretizing action timing, and therefore infinite possible spaces to explore.

Keeping with the principle of trying simple solutions first, we seek to reapply the idea of Iterative Refinement to ϵ-RBFS. However, if we use an upper-bound of ∞, the first iteration with the initial delay will never terminate if it does not find a solution. Fortunately, Korf's algorithm is designed such that it also makes sense to use values other than ∞ in the top-level call.

If we simply provide an upper bound on cost as we did with Iterative Refinement with Strong Pruning, Node Ordering, and Upper Bound, then we have an algorithm which does an ϵ-admissible search of all nodes within the cost upper bound on each iteration, successively refining until the granularity is fine enough for a solution to be found within that bound if it exists. Iterative Refinement with ϵ-RBFS is described in pseudocode in Algorithm 23.

Algorithm 23 SADAT Iterative Refinement with ϵ - Recursive Best-First Search

```
SADATIReRBFS(rootNode, bound, initialDelay, espilon, refinementLimit)
▷ Input: root node, upper bound on solution cost,
         initial simulation delay, epsilon minimum bound increment,
         limit on number of refinement iterations
▷ Output: goal node if solution found, null if not
goalNode ← null
refinement ← 1
while (goalNode = null and not refinement > refinementLimit) do
    SADATeRBFS(rootNode, f(rootNode), bound, initialDelay/refinement, espilon)
    refinement ← refinement + 1
return goalNode
```

The result of applying Iterative Refinement with ϵ-RBFS to the 10-Ship Submarine Channel Problem is shown in Table 5.8. Now we are able to achieve excellent results across a broad range of initial delay values.

Initial Delay	Results	% Goal	Time to Goal Min	Time to Goal Avg	Time to Goal Max	Cost to Goal Min	Cost to Goal Avg	Cost to Goal Max	Nodes/Sec
4.20	100	99	0.08	0.28	3.66	5.10	7.03	9.63	379.62
4.83	100	91	0.11	0.73	4.75	4.98	7.16	9.74	421.16
5.46	100	100	0.17	0.51	5.92	5.08	7.06	9.98	357.03
6.09	100	92	0.27	0.64	5.82	4.99	7.09	9.86	419.84
6.72	100	95	0.20	0.58	6.05	5.07	6.96	9.77	401.76
7.35	100	95	0.37	0.82	9.64	4.99	7.18	10.00	403.07

Table 5.8: Results for SADAT Iterative Refinement with ϵ - Recursive Best-First Search, $\epsilon = 0.25$

In contrast to Iterative Refinement with Strong Pruning, Node Ordering, and Upper Bound, we do not need to make any assumptions about properties of f' for this algorithm to be applicable. We also do not need to be concerned with picking a large enough time horizon, since our search is not limited to a time horizon.

Furthermore, Iterative Refinement with ϵ-RBFS provides a guarantee for the quality of the solution: Given initial delay Δt and admissible f', then any solution returned by the algorithm on iteration i will have a cost at most ϵ above the optimal solution in the full tree with delay $\Delta t/i$. If one can further prove a bound on the approximate optimality of the search tree of each iteration, then one can skip overly coarse iterations and set local ϵ_i parameters for finer iterations such that one can guarantee ϵ-optimal solutions.

SADAT Iterative Refinement with ϵ - Recursive Best-First Search provides a general, efficient, and successful method for SADAT search provided one can supply a useful heuristic evaluation function f' and an initial delay parameter which does not make search overly coarse or overly fine. As one can see in Table 5.8, the initial delay parameter can vary considerably and still allow excellent performance.

5.7 Summary and Conclusions

In the beginning of this chapter, we formalized SADAT Hybrid System Games and SADAT Hybrid Systems Search Problems. After describing the current Submarine Tactical Planning Assistance work of Smith, Jacobus, and Watson, we defined a class of problems for use as a benchmark in comparing approaches to SADAT search.

We first introduced SADAT Iterative Refinement Search, a generally applicable method which limits search to a time horizon with iteratively finer timing granularity. While performance is relatively poor with respect to the other algorithms of this chapter, this non-selective, brute-force search serves as a good baseline for comparison. In contrast to the research of this and the next chapter, almost all tree-based search research assumes a fixed action timing discretization. A small amount of research concerning search with different timing granularities has been presented within the abstraction, reformulation, and approximation research community. However, after searching literature and talking with several experts in robotics search and AI, it appears that iterative refinement with respect to a time horizon is unique.

SADAT Best-First Search is a novel variation of Best-First Search. Although one could argue that Genetic Algorithms allow branches to be split through mutation, SADAT Best-First Search appears to be the first systematic search to split branches and dynamically generate new internal nodes. This is contrasted with hierarchical decomposition in planning where such "internal" nodes are predefined. While performing much better than SADAT Iterative Refinement, SADAT Best-First Search showed a tradeoff of time to solution versus quality of solution. As such, it is better suited to offline design applications than real-time control applications. Unlike SADAT Iterative Refinement, SADAT Best-First Search and all the following algorithms of this chapter require a heuristic evaluation function f' which takes each node as input and returns an estimate of the cost to reach a goal node through that node. For our problem domain, a simple heuristic is easy to come by, but in general a good heuristic is not necessarily straightforward.

Next, we augmented SADAT Iterative Refinement Search with strong pruning,

node ordering, and an upper bound on solution cost. Strong pruning and node ordering are standard search speedup techniques. However, our use of the upper bound is novel and interesting. Since this tree search is unusual in that all iterations search with respect to the same time horizon, the upper bound does not merely focus search within an iteration as increasingly better leaf nodes are found. It *also* focuses search across all searches in future iterations. Ability to find solutions to the 10-Ship Submarine Channel Problem was excellent for a broad selection of time horizons. However, this algorithm assumes that (1) one knows a good time horizon a priori, and (2) that f' monotonically increases and is admissible. Generality of applicability is again traded off for performance.

Finally, we presented a new ϵ-admissible variant of Recursive Best-First Search (ϵ-RBFS). Seeing that its performance is very sensitive to the initial time delay, we make novel use of the ϵ-RBFS upper bound input parameter and again apply iterative refinement, The resulting algorithm, Iterative Refinement with ϵ-RBFS, had excellent performance across a broad range of input parameters. Furthermore, the solution comes with a guarantee that it has a cost at most ϵ greater than the optimal solution in the full tree of the last iteration. All of this comes without the monotonicity assumption of SADAT Iterative Refinement Search with Strong Pruning, Node Ordering, and Upper Bound.

Thus, we have made a series of novel forays into a new and challenging class of search problems. Notice that these approaches make very few assumptions about the problem domain beyond the simulation model. Most robotics navigation and motion planning algorithms make good use of the structure and constraints of the robot and environment. Generally speaking, the more one can efficiently make use of knowledge and structure of a problem domain, the greater the performance of the approach. "Knowledge is power." These algorithms seek to make minimal use of domain-specific knowledge in order to provide general kernels from which many future advances can grow.

One possible future direction is to dynamically discretize action timing according to a measure of "quiescence", or lack of immediate change in score. If the problem

domain can provide an indication of the importance of action frequency (e.g. distance to a threat for the submarine problem), then we have an additional source of knowledge to levy for search efficiency. In the future, we hope to identify simple ways of improving dynamic discretization without confining ourselves to narrow problem domains.

In the next chapter, we apply these same general action timing discretization ideas to problems where we do not assume a given action discretization.

Chapter 6

DADAT Search

Extending discrete search to hybrid system search introduces two new decisions in optimization: action discretization and action timing discretization. In this chapter we choose to address both decisions: How could a search algorithm choose both when and how to branch the search tree in order to consider possible actions? From the perspective of the search algorithm, both action discretization and action timing discretization are dynamic, i.e. both discretizations are chosen by the search algorithm. For this reason, we will call such searches "DADAT searches" as they have Dynamic Action and Dynamic Action Timing discretization.

In this chapter, we formally define a DADAT Hybrid System Game and its solitaire case, a DADAT Hybrid System Search Problem. We continue to examine the submarine channel problem, and compare the relative merits of random, information-based, and dispersed discretizations in augmenting the iterative refinement searches of the previous chapter. The dispersed discretization is presented as a compromise between the fast speed of random discretization, and the intelligent, slow decision procedure of information-based discretization. We find that the orientation of the headings in the given discretization of the previous chapter is very significant to performance. Dispersed discretization yields far better results than the given discretization of the previous chapter with randomly-rotated submarine headings.

6.1 DADAT Hybrid System Game and Search Problem

Formally, a DADAT Hybrid System Game is defined as a 7-tuple

$$\{S, s_0, \mathbf{A}, p, l, m, d\}$$

where

- S is the hybrid state space with a finite number of finite discrete variable domains, and a finite-dimensional continuous space,
- $s_0 \in S$ is the initial state,
- $\mathbf{A}$ is a finite set $\{A_1, \ldots, A_n\}$ of continuous action regions indexed $\{1, \ldots, n\}$,
- p is the number of players,
- $l : S \times \{1, \ldots, p\} \rightarrow \mathbf{A}'$ where $\mathbf{A}' \subset \mathbf{A}$ is a *legal move* function mapping from a state and player number to a finite set of legal continuous action regions which contain points representing all legal actions that may be executed in that state by that player,
- $m : S \times \mathbf{a}^p \rightarrow S \times \Re^p$ is a *move* function mapping from a state and simultaneous player actions (region index, region point pairs) to a resulting state and the utility of the combined actions for each player,
- $d : S \times \Re^+ \rightarrow S \times \Re^p$ is a *delay* function mapping from a state and non-negative time delay to the resulting state and the utility of the trajectory segment for each player. We require that $d(s, 0) = \{s, \{0, \ldots, 0\}\}$. Letting $d(s_1, t_1) = \{s_2, \{u_{1,1}, \ldots, u_{1,p}\}\}$ and $d(s_2, t_2) = \{s_3, \{u_{2,1}, \ldots, u_{2,p}\}\}$, we also require that $d(s_1, t_1 + t_2) = \{s_3, \{u_{1,1} + u_{2,1}, \ldots, u_{1,p} + u_{2,p}\}\}$.

An action is represented by the index $\{1, \ldots, n\}$ of the relevant action space, and a point within the space. The total utility of any finite trajectory is computed as the

sum of the trajectory move and delay utilities. In this time-invariant formalism, time can easily be encoded in a continuous clock variable, and time invariant behavior could thus be easily achieved.

A *DADAT Hybrid System Search Problem* is a special case of the DADAT Hybrid System Game where we are interested in finding a trajectory from the initial state to a goal state. Usually such problems are stated in terms of path cost rather than utility. Formally, a DADAT Hybrid System Search Problem is defined as a 7-tuple

$$\{S, s_0, S_g, \mathbf{A}, l, m, d\}$$

where

- S is a hybrid state space with a finite number of finite discrete variable domains, and a finite-dimensional continuous space,
- $s_0 \in S$ is an initial state,
- $S_g \subset S$ is a set of goal states,
- $\mathbf{A}$ is a finite set $\{A_1, \ldots, A_n\}$ of continuous action regions indexed $\{1, \ldots, n\}$,
- $l : S \rightarrow \mathbf{A}'$ where $\mathbf{A}' \subset \mathbf{A}$ is a *legal move* function mapping from a state to a finite set of legal continuous action regions which contain points representing all legal actions that may be executed in that state,
- $m : S \times \mathbf{a} \rightarrow S \times \Re$ is a *move* function mapping from a state and action (region index, region point pair) to a resulting state and cost of the action,
- $d : S \times \Re^+ \rightarrow S \times \Re^p$ is a *delay* function mapping from a state and non-negative time delay to the resulting state and the cost of the trajectory segment. We require that $d(s, 0) = \{s, \{0, \ldots, 0\}\}$. Letting $d(s_1, t_1) = \{s_2, \{u_{1,1}, \ldots, u_{1,p}\}\}$ and $d(s_2, t_2) = \{s_3, \{u_{2,1}, \ldots, u_{2,p}\}\}$, we also require that $d(s_1, t_1 + t_2) = \{s_3, \{u_{1,1} + u_{2,1}, \ldots, u_{1,p} + u_{2,p}\}\}$.

6.2 DADAT Submarine Channel Problem

The DADAT version of the SADAT Submarine Channel Problem of Section 5.2 is the same with only one modification. The submarine may now turn to any heading and travel at any speed up to its maximum speed. Thus the sole legal action region is a circle centered at the origin with radius equal to the magnitude of the maximum speed. Any point within the circle defines a legal heading and speed for the submarine.

As the algorithms in this chapter are variations of previous SADAT search algorithms with different means of selecting actions, we will be judging such means with respect to the previous results where an explicit action discretization is given. In all cases, we will use the previous branching factor of 17 so that in comparing DADAT search results to SADAT search results, we can learn something of the quality of the dynamic action discretizations.

6.3 DADAT Iterative Refinement with Random Action Discretization

In this section, we introduce a simple variation of SADAT Iterative Refinement with Strong Pruning, Node Ordering, and Upper Bound (§ 5.5) in which we randomly sample actions from the legal action regions. In addition to the previous parameters, we require the caller to indicate the number of samples used to sample each action region. Thus, the pseudocode is as shown in Algorithms 24 and 25.

In comparing the results of DADAT Iterative Refinement with Random Action Discretization in Table 6.1 with the algorithm's SADAT counterpart in Table 5.6, the most noticable difference is that a larger time horizon is needed for the algorithm to achieve comparable success. This is due in part to two main reasons.

First, the given SADAT discretization had eight actions at full speed in different headings. If one were to compare maximum speeds and headings of paths in our SADAT searches and this DADAT search, one would notice a much different distribution. The SADAT search will search faster trajectories than those randomly generated from possible legal moves.

Algorithm 24 DADAT Iterative Refinement with Strong Pruning, Node Ordering, Upper Bound, and Random Discretization

DADAT-IR-SPNOUB-RANDOM(*rootNode*, *initialDelay*, *refinementLimit*, *upperBound*, *sampleVector*)

▷ **Input:** root node,
initial list of branching times,
limit on number of refinement iterations,
upper bound on solution cost,
vector of samples for each possible action parameter region

▷ **Output:** goal node with cost beneath upper bound if found,
best leaf node found otherwise

globalUpperBound ← *upperBound*
globalGoalFound ← **false**
globalBestNode ← **null**
refinement ← 1
while (not *globalGoalFound* and not *refinement* > *refinementLimit*) **do**
  DADAT-DFS-SPNOUB-Random(*rootNode*, *initialDelay*/*refinement*, *refinement*, *sampleVector*)
  refinement ← *refinement* + 1
return *globalBestNode*

Time Horizon	Results	% Goal	Time to Goal Min	Time to Goal Avg	Time to Goal Max	Cost to Goal Min	Cost to Goal Avg	Cost to Goal Max	Nodes/Sec
4.20	100	0	N/A	N/A	N/A	N/A	N/A	N/A	1,231.64
4.83	100	1	0.30	0.30	0.30	4.76	4.76	4.76	1,200.95
5.46	100	38	0.07	3.27	10.06	4.96	5.29	5.46	928.31
6.09	100	61	0.10	2.26	10.07	5.01	5.77	6.09	770.25
6.72	100	73	0.08	2.74	10.06	5.14	6.24	6.71	656.71
7.35	100	84	0.16	3.09	10.08	5.38	6.79	7.35	584.01

Table 6.1: Results for DADAT Iterative Refinement with Random Action Discretization

Algorithm 25 DADAT Depth-First Search with Strong Pruning, Node Ordering, Upper Bound, and Random Discretization

```
DADAT-DFS-SPNOUB-RANDOM(node, delay, depthLimit, sampleVector)
▷ Input: search node,
         simulation delay,
         depth of search below node, and
         vector of samples for each possible action parameter region
if (isGoal(node)) then
    globalGoalFound ← true
    globalBestNode ← node
    return
if (depthLimit = 0 or legalMoveRegions(node) = null) then
    if (f(node) < f(globalBestNode)) then
        globalBestNode ← node
    return
childCount ← 0
foreach move region r[i] of legalMoveRegions(node) do
    for i ← 1 to sampleVector[r[i].index] do
        childCount ← childCount + 1
        child[childCount] ← wait(makeMove(clone(node), randomMove(r[i])), delay)
Sort child[i] in increasing order of f(child[i])
i ← 1
done ← false
while (not done and not globalGoalFound) do
    ▷ Do not expand a node with f-value exceeding the global upper bound
    if (f(child[i]) > globalUpperBound) then
        done ← true
    else
        DADAT-DFS-SPNOUB-Random(child[i], delay, depthLimit − 1)
    i ← i + 1
    if (i > childCount) then
        done ← true
return
```

Second, most solutions found by SADAT searches tend to run due east along the top bank, varying speed as necessary to time passing between patrolling ships just as a person walks through an automatic revolving door. In previous experimentation, optimal trajectories often contained segments where the submarine was heading due east at full speed. In randomly generating headings and speeds, the search will not always be presented with a similar action, and thus will not find solutions as optimal or as often.

It would be desirable to see how much the decrease in performance of these results is due to not having the SADAT discretization's full-speed actions versus not having the SADAT discretization's due-east actions. One way would be to randomly rotate the SADAT discretization and see the resulting performance. Another way would be to add an additional linear move region consisting of different speeds with a due-east heading. Allotting samples to a second move region would amount to providing additional domain knowledge for search. In keeping with a desire for maximum generality, we will use the former means rather than the latter.

The results of using SADAT Iterative Refinement with Strong Pruning, Node Ordering, and Upper Bound with random rotations of the original action discretization are shown in Table 6.2. From these results, it is immediately apparent that the orientation of our original discretization was very significant. Neither approach is better for all chosen time horizons. While random discretization is clearly dominated by the original discretization, it is roughly comparable to the randomly rotated discretization. The random discretization success rate for finding solutions peaks at a greater time horizon than that of the randomly rotated discretization. With random discretization, the average action speed will be less than that of the rotated discretization, necessitating a greater time horizon on average for solutions.

Time Horizon	Results	% Goal	Time to Goal			Cost to Goal			Nodes/Sec
			Min	Avg	Max	Min	Avg	Max	
4.20	100	0	N/A	N/A	N/A	N/A	N/A	N/A	1,120.86
4.83	100	26	0.12	4.63	10.04	4.43	4.74	4.83	923.82
5.46	100	61	0.24	3.35	10.04	4.71	5.26	5.45	740.12
6.09	100	73	0.23	3.71	9.87	4.44	5.67	6.08	599.26
6.72	100	67	0.09	3.94	9.94	4.91	6.11	6.72	538.88
7.35	100	49	0.17	4.42	10.04	4.88	6.50	7.33	513.83

Table 6.2: Results for SADAT Iterative Refinement with Strong Pruning, Node Ordering, Upper Bound, and Randomly Rotated Action Discretization

6.4 DADAT Iterative Refinement with Information-Based Action Discretization

In this section, we take a different approach to the selection of actions for search. Rather than selecting them randomly, we apply information-based optimization. The pseudocode is given in Algorithms 26–29.

When applied to the DADAT Submarine Channel Problem, this algorithm was not able to solve any of the 100 problem instances with any of the 6 different time horizons. In the DASAT work of Chapter 4, we saw the benefit of applying Information-Based Optimization to the choice of actions in alpha-beta search. In the Magnetic Levitation Problem, we were interested in offline design where a single one-dimensional action region defined possible control actions, i.e. possible solenoid current settings. In one dimension, information-based optimization allows for direct calculation of the next best point to evaluate.

In this DASAT Submarine Channel Problem, the action space is two-dimensional. Thus we must use the candidate-sampling multidimensional version of Information-Based Optimization which selects random candidate points and performs calculations with respect to every previously evaluated point to check for shadowing and slope to a goal value at the candidate point. To review details of the algorithm, see Section 2.7.

Algorithm 26 DADAT Iterative Refinement with Strong Pruning, Node Ordering, Upper Bound, and Information-Based Discretization

DADAT-IR-SPNOUB-IB(*rootNode*, *initialDelay*, *refinementLimit*, *upperBound*, *sampleVector*)

▷ **Input:** root node,
initial list of branching times,
limit on number of refinement iterations,
upper bound on solution cost,
vector of samples for each possible action parameter region

▷ **Output:** goal node with cost beneath upper bound if found,
best leaf node found otherwise

globalUpperBound ← *upperBound*
globalGoalFound ← **false**
globalBestNode ← **null**
refinement ← 1
while (not *globalGoalFound* and not *refinement* > *refinementLimit*) **do**
  DADAT-DFS-SPNOUB-IB(*rootNode*, *initialDelay*/*refinement*, *refinement*, *sampleVector*)
  refinement ← *refinement* + 1
return *globalBestNode*

Algorithm 27 DADAT Depth-First Search with Strong Pruning, Node Ordering, Upper Bound, and Information-Based Discretization

DADAT-DFS-SPNOUB-IB(*node*, *delay*, *depthLimit*, *sampleVector*)

▷ **Input:** search node,
simulation delay,
depth of search below node, and
vector of samples for each possible action parameter region

▷ **Output:** exact or lower bound value through node

if (isGoal(*node*)) **then**
  globalGoalFound ← **true**
  globalBestNode ← *node*
  return f(*node*)
if (*depthLimit* = 0 or legalMoveRegions(*node*) = **null**) **then**
  if (f(*node*) < f(*globalBestNode*)) **then**
    globalBestNode ← *node*
  return f(*node*)
foreach move region *r[i]* of legalMoveRegions(*node*) **do**
  init-IB-Optimizer(*optimizer[i]*, *r[i]*, *sampleVector[r[i].index]*, *globalTargetValue*)
  {*moveChoice[i]*, *child[i]*} ← IB-NextChild(*node*, *optimizer[i]*, *delay*)
return DADAT-DFS-SPNOUB-IB-expand(*node*, *optimizer*, *moveChoice*, *child*, *delay*)

Algorithm 28 IB-NextChild Procedure for Algorithms 27 and 29

IB-NEXTCHILD(*node*, *optimizer*, *delay*)
▷ **Input:** parent node,
information-based optimizer for move region, and
simulation delay
▷ **Output:** chosen move parameters, and
best next child node to expand accord to info-based optimization
▷ nextChoice returns null when optimizer sample limit is reached
moveChoice ← nextChoice(*optimizer[i]*)
if (not *moveChoice* = **null**) **then**
    move ← createMove(*optimizer.region.index*, *moveChoice*)
    child ← wait(makeMove(clone(*node*), *move*), *delay*)
else
    child ← **null**
return {*moveChoice*, *child*}

Algorithm 29 Child Expansion Procedure for Algorithm 27

```
DADAT-DFS-SPNOUB-IB-EXPAND(node, optimizer, moveChoice, child, delay)
▷ Input: current node,
         information-based optimizers for move regions,
         candidate move choices for move regions,
         associated child choices for move regions, and
         simulation delay
▷ Output: goal node with cost beneath upper bound if found,
          best leaf node found otherwise
nextBestF ← ∞
childNum ← -1
foreach child[i] do
    if (not child[i] = null and f(child[i]) < nextBestF) then
        nextBestF ← f(child[i])
        childNum ← i
done ← childNum = -1
fMin ← ∞
while (not done and not globalGoalFound) do
    ▷ Do not expand a node with f-value exceeding the global upper bound
    if (nextBestF > globalUpperBound) then
        ▷ If pruned, use f-value as return value
        returnValue ← nextBestF
    else
        returnValue ← DADAT-DFS-SPNOUB-IB(child[childNum], delay,
                                          depthLimit − 1)
    if (returnValue < fMin) then
        fMin ← returnValue
    addData(optimizer[childNum], moveChoice[i], returnValue)
    {moveChoice[childNum], child[childNum]}
        ← IB-NextChild(node, optimizer[childNum], delay)
    nextBestF ← ∞
    childNum ← -1
    foreach child[i] do
        if (not child[i] = null and f(child[i]) < nextBestF) then
            nextBestF ← f(child[i])
            childNum ← i
    done ← childNum = -1
return fMin
```

Multidimensional information-based optimization has greater computational complexity than that of the one dimensional case because of the check for shadowing. The high computational overhead expended in the intelligent selection of actions for search outweighed the benefit of the intelligent selection for our real-time problem. However, this algorithm may prove useful in problem domains with smaller branching factors where intelligent sampling has a high payoff in search efficiency or solution quality.

6.5 DADAT Iterative Refinement with Dispersed Action Discretization

We have seen that random sampling is computationally inexpensive, yet the sampling is inferior to the given action discretization for the SADAT Submarine Channel Problem. We have also seen that information-based optimization makes intelligent choices, yet the computational complexity of information-based optimization makes it unsuitable for this real-time problem domain. We are presented with a tradeoff between computational efficiency and the utility of such computation. One would desire a compromise between the strengths of random and information-based discretization which would echo the intuition behind the choice of the SADAT discretization without incurring such computational cost for each node expansion.

In seeking a compromise, we note that information-based minimization of a finite-valued function with a target value of $-\infty$ will yield a set of points, each of which is as far as possible from the previous points. See Figure 6.1. If one were to perform such an optimization for a circular area with the first point on the edge of the circle, the second point would be directly across the circle. The third and fourth points would be directly across from each other rotated 90 degrees from the first and second points. The fifth point would be farthest from the previous four in the center. The following four points would be chosen in positions rotated 45 degrees from the first four. The following eight would be chosen at centers of circles circumscribing triangles formed by the center point and closest pairs of edge points.

Given a starting point on the edge of the circular move region, the first 17 points

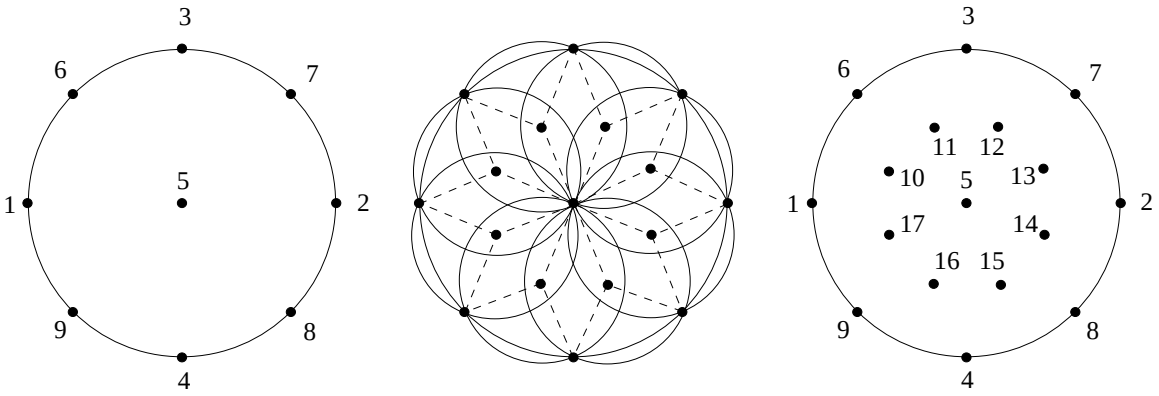

Figure 6.1: Information-Based Optimization point choices for a finite values and an infinite target, confined to a circular region

of information based minimization with a target of $-\infty$ look remarkably like the SADAT action discretization. One needs only to slightly increase the speeds of the half-speed moves and rotate their headings 22.5 degrees. The point here is that the intuitive choice of the SADAT action discretization echoes a mathematically well-founded choice of information-based optimization with an infinite target.

If we could have our algorithm dynamically and efficiently compute a discretization with points as far away from each other as possible, we would expect much improvement. While a detailed investigation of such techniques is beyond the scope of this dissertation, we have implemented a simple point dispersion technique based on simulating repulsive electrical forces.

The basic idea of "dispersed" discretization is to take a number of randomly sampled points from the action region and simulate them as if they were point charges mutually repelling each other with force proportional to the inverse square of their distance. The point dispersion algorithm pseudocode is given in Algorithm 30. We use a repulsion factor of 0.008 and a repulsion factor decay of 0.93 for 20 iterations. These values were chosen empirically based on a small number of trials with the submarine action region. In future work, we would desire these dispersion parameters to be rapidly self-adapting to the size of the region and the number of sampled points.

In pseudocode Algorithms 31–32, we present a variation on SADAT Iterative

Algorithm 30 Dispersed Discretization

```
DISPERSE-POINTS(region, samples, weight, decay, iterations)
▷ Input: move parameter region,
         number of points to sample,
         weight of change for first iteration,
         decay of change for following iterations,
         number of iterations
▷ Output: an array of dispersed points within the region
for i ← 1 to samples do
    x[i] ← randomPoint(region)
for i ← 1 to iterations do
    for j ← 1 to samples do
        dx[j] ← 0
        for k ← 1 to j do
            difference ← x[k] − x[j]
            distance ← sqrt(x[j]^2 + x[k]^2)
            dx[j] ← dx[j] − difference/(distance^3)
            dx[k] ← dx[j] + difference/(distance^3)
    for j ← 1 to samples do
        dx[j] ← weight ∗ dx[j]
        x[j] ← x[j] + dx[j]
        if (not inRegion(x[j], region)) then
            ▷ Reassign to closest point on region border
            containInRegion(x[j], region)
    weight ← weight ∗ decay
return x
```

Refinement with Strong Pruning, Node Ordering, and Upper Bound (§ 5.5) where we lazily compute dispersed discretization for move regions. That is, as a move discretization is needed, we look to a list of discretizations indexed by region. If a discretization has not yet been computed, we compute it, otherwise we use the precomputed global discretization for that move region.

Using this dispersed discretization, we obtain excellent results for the 10-Ship DADAT Submarine Channel Problem as shown in Table 6.3. As before, we note that good performance requires the time horizon parameter to be sufficiently high. Particularly surprising is the fact that the results are better than those with the given SADAT discretization.

Looking over a number of dispersed discretizations, one quickly notices that more

Algorithm 31 DADAT Iterative Refinement with Strong Pruning, Node Ordering, Upper Bound, and Random Discretization

DADAT-IR-SPNOUB-DISPERSED(*rootNode*, *initialDelay*, *refinementLimit*, *upperBound*, *sampleVector*, *dispersionWeight*, *dispersionDecay*, *dispersionIterations*)

▷ **Input:** root node,
initial list of branching times,
limit on number of refinement iterations,
upper bound on solution cost,
vector of samples for each possible action parameter region,
weight of change for first dispersion iteration,
decay of change for following dispersion iterations,
number of dispersion iterations

▷ **Output:** goal node with cost beneath upper bound if found,
best leaf node found otherwise

globalUpperBound ← *upperBound*
globalGoalFound ← **false**
globalBestNode ← **null**
refinement ← 1
while (not *globalGoalFound* and not *refinement* > *refinementLimit*) **do**
    DADAT-DFS-SPNOUB-Dispersed(*rootNode*, *initialDelay*/*refinement*, *refinement*, *sampleVector*, *dispersionWeight*, *dispersionDecay*, *dispersionIterations*)
    refinement ← *refinement* + 1
return *globalBestNode*

Algorithm 32 DADAT Depth-First Search with Strong Pruning, Node Ordering, Upper Bound, and Dispersed Discretization

```
DADAT-DFS-SPNOUB-DISPERSED(node, delay, depthLimit, sampleVector,
                           dispWeight, dispDecay, dispIterations)
▷ Input: search node, simulation delay, depth of search below node,
         vector of samples for each possible action parameter region,
         weight of change for first dispersion iteration,
         decay of change for following dispersion iterations,
         number of dispersion iterations
if (isGoal(node)) then
    globalGoalFound ← true
    globalBestNode ← node
    return
if (depthLimit = 0 or legalMoveRegions(node) = null) then
    if (f(node) < f(globalBestNode)) then
        globalBestNode ← node
    return
childCount ← 0
foreach move region r[i] of legalMoveRegions(node) do
    index ← r[i].index
    if (dispersedMoves[index] = null) then
        dispersedPoints
            ← disperse-points(r[i], sampleVector[index], dispWeight,
                              dispDecay, dispIterations)
        for j ← 1 to sampleVector[index] do
            dispersedMove[index][j] ← createMove(index, dispersedPoint[j])
    for j ← 1 to sampleVector[index] do
        childCount ← childCount + 1
        child[childCount]
            ← wait(makeMove(clone(node), dispersedMove[index][j]),
                   delay)
Sort child[i] in increasing order of f(child[i])
i ← 1
done ← false
while (not done and not globalGoalFound) do
    ▷ Do not expand a node with f-value exceeding the global upper bound
    if (f(child[i]) > globalUpperBound) then
        done ← true
    else
        DADAT-DFS-SPNOUB-Dispersed(child[i], delay, depthLimit − 1)
    i ← i + 1
    if (i > childCount) then
        done ← true
return
```

points are repelled to the edge than in the given SADAT discretization. Although not a probable configuration, any number of points placed at even intervals around the edge would be in equilibrium. With repulsion parameters given above, it was typical to see 12 or more points along the edge of the circle with 5 or fewer points dispersed internally. As noted in the previous discussion, the extreme parameters represented by the edge of the circular action region are more likely to appear in optimal solutions. We hypothesize that having extra edge action choices aids in finding better approximations to optimal solutions.

Furthermore, in this problem domain, searches of faster submarine trajectories (i.e. with discretizations having more maximal velocities) will have lesser search depths to solutions if such speedy solution trajectories exist. Since search depth affects search time complexity exponentially, we likely benefit from a discretization with more maximal velocity values.

Time			Time to Goal			Cost to Goal			
Horizon	Results	% Goal	Min	Avg	Max	Min	Avg	Max	Nodes/Sec
4.20	100	0	N/A	N/A	N/A	N/A	N/A	N/A	911.98
4.83	100	92	0.04	1.32	10.07	4.32	4.69	4.83	1,107.76
5.46	100	97	0.04	0.57	10.06	4.27	5.09	5.46	829.24
6.09	100	98	0.05	0.78	9.94	4.27	5.52	6.09	694.12
6.72	100	98	0.06	0.68	4.04	4.30	5.94	6.72	591.83
7.35	100	100	0.03	1.33	10.06	4.20	6.48	7.35	539.47

Table 6.3: Results for DADAT Iterative Refinement with Dispersed Action Discretization

6.6 DADAT Iterative Refinement with Dispersed ϵ-RBFS

In this section, we apply dispersed discretization to SADAT Iterative Refinement with ϵ-RBFS to create another DADAT search algorithm we call DADAT Iterative

Refinement with Dispersed ϵ-RBFS. The algorithm is given in pseudocode in Algorithms 33–34.

Algorithm 33 DADAT Iterative Refinement with ϵ - Recursive Best-First Search and Dispersed Discretization

DADAT-IR-eRBFS-DISPERSED(*rootNode*, *bound*, *initialDelay*, *espilon*, *refinementLimit*, *sampleVector*, *dispWeight*, *dispDecay*, *dispIterations*)

▷ **Input:** root node,
upper bound on solution cost,
initial simulation delay,
epsilon minimum bound increment,
limit on number of refinement iterations,
vector of samples for each possible action parameter region,
weight of change for first dispersion iteration,
decay of change for following dispersion iterations,
number of dispersion iterations

▷ **Output:** goal node if solution found, null if not

goalNode ← **null**
refinement ← 1
while (*goalNode* = **null** and not *refinement* > *refinementLimit*) **do**
  DADAT-eRBFS-dispersed(*rootNode*, f(*rootNode*), *bound*, *initialDelay*/*refinement*, *espilon*, *sampleVector*, *dispWeight*, *dispDecay*, *dispIterations*)
  refinement ← *refinement* + 1
return *goalNode*

The quality of the results for the 10-Ship DADAT Submarine Channel Problem are good, but not so good as DADAT Iterative Refinement with Dispersed Action Discretization, Strong Pruning, Node Ordering, and Upper Bound. However, this algorithm commends itself for use where f' is not monotonic, or where a good time horizon is not known. Consider the broad range of initial delay parameters over which we have good results in Table 6.4. The parameters for dispersed discretization were as follows: $dispWeight = 0.008$, $dispDecay = 0.93$, $dispIterations = 20$,

To again see how the dispersed discretization is an improvement over the randomly rotated given discretization of the SADAT version of the problem, consider the results of Table 6.5. For the same problems, the dispersed discretization increases the number of solutions found by about 33%.

Initial Delay	Results	% Goal	Time to Goal Min	Time to Goal Avg	Time to Goal Max	Cost to Goal Min	Cost to Goal Avg	Cost to Goal Max	Nodes/Sec
1.00	100	76	0.01	1.89	9.50	4.32	7.71	9.53	491.39
2.00	100	71	0.02	1.61	9.37	5.02	7.77	9.65	490.55
3.00	100	74	0.03	1.89	8.10	4.31	7.71	9.63	502.99
4.00	100	69	0.04	1.89	8.91	5.70	7.94	9.77	491.30
4.20	100	72	0.01	1.82	8.77	4.46	8.16	10.00	454.78
4.83	100	77	0.03	3.10	9.56	4.70	8.07	9.84	471.74
5.46	100	70	0.03	2.80	9.92	4.29	8.15	9.98	461.75
6.09	100	69	0.05	2.45	9.86	4.31	7.98	9.98	465.63
6.72	100	73	0.02	3.01	10.04	4.20	8.07	9.98	448.34
7.35	100	68	0.04	2.93	9.21	4.89	8.46	9.94	453.53

Table 6.4: Results for DADAT Iterative Refinement with Dispersed ϵ-RBFS

Initial Delay	Results	% Goal	Time to Goal Min	Time to Goal Avg	Time to Goal Max	Cost to Goal Min	Cost to Goal Avg	Cost to Goal Max	Nodes/Sec
4.20	100	47	0.06	2.76	8.78	5.95	8.18	9.80	477.43
4.83	100	34	0.04	3.67	9.51	6.17	8.20	9.96	460.03
5.46	100	39	0.17	2.69	7.57	5.58	8.26	10.00	464.32
6.09	100	38	0.22	5.31	10.02	5.53	8.11	9.99	456.96
6.72	100	33	0.03	2.97	9.53	6.00	8.25	9.82	452.57
7.35	100	40	0.17	4.94	9.99	5.95	8.16	9.95	448.07

Table 6.5: Results for SADAT Iterative Refinement with ϵ-RBFS and Randomly Rotated Action Discretization

Algorithm 34 DADAT ϵ - Recursive Best-First Search with Dispersed Discretization

```
DADAT-eRBFS-DISPERSED(node, nodeF, bound, delay, espilon, sampleVector,
                      dispWeight, dispDecay, dispIterations)
▷ Input: node, calling stored search value of node, local cost upper bound,
         simulation delay, epsilon minimum bound increment,
         vector of samples for each possible action parameter region,
         weight of change for first dispersion iteration,
         decay of change for following dispersion iterations,
         number of dispersion iterations
▷ Output: return stored search value of node
if (f(node) > bound) then
    return f(node)
if (isGoal(node)) then
    goalNode ← node
    exit algorithm
if (numOfChildren(node) = 0) then
    return ∞
foreach move region r[i] of legalMoveRegions(node) do
    index ← r[i].index
    if (dispersedMoves[index] = null) then
        dispersedPoints ← disperse-points(r[i], sampleVector[index], dispWeight,
                                          dispDecay, dispIterations)
        for j ← 1 to sampleVector[index] do
            dispersedMove[index][j] ← createMove(index, dispersedPoint[j])
    for j ← 1 to sampleVector[index] do
        childCount ← childCount + 1
        c ← wait(makeMove(clone(node), dispersedMove[index][j]), delay)
        if (f(node) < nodeF) then
            cF ← max(nodeF, f(c))
        else
            cF ← f(c)
        insert(heap, c, cF)
{c, cF} ← extractMin(heap)
while (cF ≤ bound and cF < ∞) do
    ▷ The new local upper bound must increase by at least epsilon.
    if (childCount > 1) then
        cF ← max(DADAT-eRBFS-dispersed(c, cF, min(bound, minValue(heap))),
                 cF + epsilon)
    else
        cF ← max(DADAT-eRBFS-dispersed(c, cF, bound), cF + epsilon)
    insert(heap, c, cF)
    {c, cF} ← extractMin(heap)
return cF
```

Dispersed discretization parameters were tuned across several runs. While the chosen dispersed discretization parameters were reasonably well chosen for the submarine action parameter region, they would obviously not be generally suited for all regions one might encounter. In future work, it would be good to have such parameters be adaptively tuned much as step size is tuned in local optimization. If one could reliably get convergence to a good dispersion, then dispersion parameters could be removed from these algorithms and their use would be simplified.

6.7 Conclusions

In this chapter, we gave formal definitions of DADAT Hybrid System Games and DADAT Hybrid System Search Problems. We defined the DADAT Submarine Channel Problem as the SADAT Submarine Channel problem without a given action discretization. The submarine instead is allowed any heading and any speed up to its maximum speed.

We then investigated means of augmenting SADAT search techniques of the previous chapter such that action discretizations are performed dynamically. We observed that the percentage of solutions found for random discretization is comparable to those achieved with SADAT action discretization when headings are uniformly rotated by a random angle. However, cost to goal of such solutions is increased. This is due to the fact that optimal submarine path solutions often involve extreme values, especially full speed. The random discretization will, on average, have considerably fewer actions near full speed than the SADAT discretization.

We next observed the unsuccessful application of information-based optimization to action discretization. While making good decisions in principle, the overhead of performing a multidimensional information-based optimization at each node is too burdensome for this real-time task. Thus the computational benefit of more intelligent node expansion is outweighed by the computational cost of computing such choices.

Between random discretization and information-based optimization based on sound mathematical principles, we wished to find a compromise: a discretization which would reflect informed choices while being very simple to compute. We observed that

an extreme case of information-based optimization, where the function is finite-valued and the target is infinite, yields a discretization where each point is as far away as possible from preceding points. In fact, one such information-based optimization yields a discretization remarkably similar to the SADAT discretization we were given.

Based on the extreme case of information-based optimization, and imitating the natural phenomenon of electrostatic repulsion of "point" charges, we developed a dispersion algorithm which yielded discretizations with considerably better goal finding performance than was achieved with the given SADAT action discretization with headings uniformly rotated by a random angle.

It should be noted that a good representation of the problem is necessary to the success of search applications. Two specific characteristics are of special note. First, one should keep the representation as simple as possible. Complex behaviors need not have complex underlying decisions, and keeping the dimensionality of action parameter regions low is important given the limited sampling one can perform.

Second, one should represent the action parameter regions in such a way as to uniformly distribute parameters according to likelihood of utility of such actions. For example, one could represent possible submarine actions as a rectangle with sides bounding possible headings and speeds. Compared to uniform sampling of the circular representation, uniform sampling of the rectangular representation gives greater importance to moves with slower speeds. Of course, this issue could also be avoided at the action parameter representation level if we specialize our discretization methods to vary importance of sampling over action parameter regions.

The main point is that at some level, one encodes a notion of sampling importance over possible action parameters. Choosing low dimensional action parameter region representations which uniformly distribute the likely importance of parameters is important in representing a problem for successful use with these techniques.

In summary, if a good time horizon is known and the heuristic evaluation function f' is known to be monotonic, then among our algorithms, DADAT Iterative Refinement with Strong Pruning, Node Ordering, Upper Bound, and Dispersed Discretization is preferred. Otherwise, if one can provide a decent heuristic evaluation

function, then DADAT Iterative Refinement with ϵ-RBFS and Dispersed Discretization is preferred.

Thus, we have introduced a collection of algorithms which perform dynamic discretization of action and action timing in search. There is much yet to be done in this area, yet we hope that these first steps will bring Artificial Intelligence and Control researchers closer to fruitful common work.

Bibliography

[1] Tamar Başar and Geert Jan Olsder. *Dynamic Noncooperative Game Theory, 2nd Ed.* Academic Press, London, 1995.

[2] E. Bizzi, F.A. Mussa-Ivaldi, and S. Giszter. Computations underlying the execution of movement: A biological perspective. *Science*, 253:287–291, 1991.

[3] C. Guus E. Boender and H. Edwin Romeijn. Stochastic methods. In Horst and Pardalos [19], pages 829–869.

[4] E. Bradley. Autonomous exploration and control of chaotic systems. *Cybernetics and Systems*, 26(5):499–519, 1995.

[5] Michael S. Branicky. *Studies in Hybrid Systems: modeling, analysis, and control.* PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, USA, 1995.

[6] F.H. Branin. Widely convergent method for finding multiple solutions of simultaneous nonlinear equations. *I.B.M. J. R&D.*, Sept 1972.

[7] Arthur E. Bryson, Jr. and Yu-Chi Ho. *Applied Optimal Control: optimization, estimation, and control.* Hemisphere Publishing Corporation, New York, 1975.

[8] A. Corana, M. Marchesi, C. Martini, and S. Ridella. Minimizing multimodal functions of continuous variables with the "simulated annealing" algorithm. *ACM Trans. Mathl. Software*, 13(3):262–279, 1987.

[9] R. Dechter and J. Pearl. Generalized best-first search strategies and the optimality of a^*. *Journal of the Association of Computing Machinery*, 32(3):505–536, 1985.

[10] Rutvik Desai and Rajendra Patil. SALO: combining simulated annealing and local optimization for efficient global optimization. In J.H. Stewman, editor, *Proceedings of the 9th Florida AI Research Symposium (FLAIRS-'96)*, pages 233–237, St. Petersburg, FL, USA, 1996. Eckerd Coll.

[11] Gene F. Franklin, J. David Powell, and Abbas Emami-Naeini. *Feedback Control of Dynamic Systems – 3rd Ed.* Addison-Wesley, Menlo Park, California, USA, 1994.

[12] Gene F. Franklin, J. David Powell, and Michael Workman. *Digital Control of Dynamic Systems – 3rd Ed.* Addison-Wesley, Menlo Park, California, USA, 1998.

[13] Andrew Grace. *Optimization Toolbox*. The Mathworks Inc., 24 Prime Park Way, Natick, MA 01760-1500 USA.

[14] T.P. Hart and D.J. Edwards. The tree prune (tp) algorithm. memo 30, M.I.T. Artificial Intelligence Project, Cambridge, Massachusetts, December 1961.

[15] Thomas A. Henzinger, Pei-Hsin Ho, and Howard Wong-Toi. Hytech: the next generation. In *Proceedings of the 16th Annual IEEE Real-time Systems Symposium (RTSS 1995)*, pages 56–65. IEEE Computer Society Press, 1995.

[16] Thomas A. Henzinger, Pei-Hsin Ho, and Howard Wong-Toi. Hytech: a model checker for hybrid systems. *Software Tools for Technology Transfer*, 1(1):110–122, 1997.

[17] Thomas A. Henzinger, Pei-Hsin Ho, and Howard Wong-Toi. Algorithmic analysis of nonlinear hybrid systems. *IEEE Transactions on Automatic Control*, 43:540–554, 1998.

[18] Thomas A. Henzinger and Shankar Sastry, editors. *LNCS 1386: Hybrid Systems: computation and control, First International Workshop, HSCC'98, Proceedings.* Springer, Berlin, 1998.

[19] Reiner Horst and Ranos M. Pardalos, editors. *Handbook of Global Optimization.* Kluwer Academic, Dordrecht, Netherlands, 1995.

[20] C.S. Hsu. *Cell-to-Cell Mapping; A Method of Global Analysis for Nonlinear Systems*. Springer-Verlag, 1987. Series: Applied Mathematical Science v. 64.

[21] Lester Ingber. Adaptive simulated annealing (ASA): Lessons learned. *Control and Cybernetics*, 25(1):33–54, 1996.

[22] S. Kirkpatrick, C.D. Gelatt, and M.P. Vecchi. Optimization by simulated annealing. *Science*, 220:671–680, 1983.

[23] Donald E. Knuth and R.E. Moore. An analysis of alpha-beta pruning. *Artificial Intelligence*, 6:293–326, 1975.

[24] Richard E. Korf. Multi-player alpha-beta pruning. *Artificial Intelligence*, 48:99–111, 1991.

[25] Richard E. Korf. Linear-space best-first search. *Artificial Intelligence*, 62:41–78, 1993.

[26] Albert C. Leenhouts. *Step Motor System Design Handbook*. Litchfield Engineering, Kingman, Arizona, USA, 1991.

[27] C.A. Luckhardt and K.B. Irani. An algorithmic solution of n-person games. In *Proceedings AAAI-86*, pages 158–162, Philadelphia, Pennsylvania, USA, 1986.

[28] Jeff A. May and Feng Zhao. Verification of control laws using phase-space geometric modeling of dynamical systems. In *Proceedings of the IFAC Symposium on Artificial Intelligence in Real-Time Control, October 5–8, 1998, Grand Canyon National Park, Arizona, USA*, Oxford, UK, 1998. Elsevier Science.

[29] N. Metropolis, A.W. Rosenbluth, M.N. Rosenbluth, A.H. Teller, and E. Teller. Equation of state calculations by fast computing machines. *J. Chem. Phys.*, 21(6):1087–1092, 1953.

[30] Jonas Mockus. *Bayesian Approach to Global Optimization: theory and applications*. Kluwer Academic, Dordrecht, The Netherlands, 1989.

[31] Jonas Mockus. Application of bayesian approach to numerical methods of global and stochastic optimization. *J. Global Optimization*, 4:347–365, 1994.

[32] A.W. Moore, G.T. Atkeson, and Schaal S.A. Memory-based learning for control. Technical Report CMU-RI-TR-95-18, The Robotics Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania, USA, April 1995.

[33] A. Newell, J. Shaw, and H. Simon. Chess playing programs and the problem of complexity. *IBM J. Res. and Develop.*, 2:39–70, October 1958.

[34] Nils Nilsson. *Problem Solving Methods in Artificial Intelligence.* McGraw-Hill Book Company, New York, 1971.

[35] Judea Pearl. *Heuristic - intelligent search strategies for computer problem solving.* Addison-Wesley, Reading, Massachusetts, USA, 1984.

[36] János D. Pintér. *Global Optimization in Action - continuous and Lipschitz optimization: algorithms, implementations, and applications.* Kluwer Academic, Dordrecht, Netherlands, 1996.

[37] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes Example Book (C) - 2nd Ed.* Cambridge University Press, Cambridge, 1992.

[38] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes in C: the art of scientific computing - 2nd Ed.* Cambridge University Press, Cambridge, 1992.

[39] A.H.G. Rinnooy Kan and G.T. Timmer. Stochastic global optimization methods; part I: multi level methods. *Mathematical Programming*, 39:27–56, 1987.

[40] A.H.G. Rinnooy Kan and G.T. Timmer. Stochastic global optimization methods; part II: clustering methods. *Mathematical Programming*, 39:57–78, 1987.

[41] Stuart Russell and Peter Norvig. *Artificial Intelligence: a modern approach.* Prentice Hall, Upper Saddle River, NJ, USA, 1995.

[42] Stuart Russell and Eric Wefald. *Do the Right Thing: studies in limited rationality*. MIT Press, Cambridge, MA, USA, 1991.

[43] A. Samuel. Some studies in machine learning using the game of checkers ii. recent progress. *IBM J. Res. and Develop.*, 11(6):601–617, November 1967.

[44] Yaroslav D. Sergeyev. An information global optimization algorithm with local tuning. *SIAM J. Optimization*, 5(4):858–870, 1995.

[45] J.R. Slagle and J.K. Dixon. Experiments with some programs that search game trees. *Journal of the Association of Computing Machinery*, 16(2):189–207, 1969.

[46] Thomas C. Smith, Peter W. Jacobus, and David P. Watson. Preparing to do tomorrow's job today - automated tactical and mission planning assistance for the information age submarine. Submarine Technology Symposium 1998, Johns Hoskins University Applied Physics Laboratory, 1998.

[47] Roman G. Strongin. *Numerical Methods in Multiextremal Problems*. Nauka, Moscow, 1978. (In Russian).

[48] Roman G. Strongin. Deriving computing schemes for multiextremal problems. In *The Contemporary State of the Operations Research*. Nauka, Moscow, 1979. (In Russian).

[49] Roman G. Strongin. The information approach to multiextremal optimization problems. *Stochastics and Stochastics Reports*, 27:65–82, 1989.

[50] Andrew Stuart and A.R. Humphries. *Dynamical Systems and Numerical Analysis*. Cambridge University Press, Cambridge, 1996.

[51] A.G. Sukharev. *Optimal Search of Extremum*. Moscow University Press, Moscow, 1975.

[52] Claire Tomlin, John Lygeros, and Shankar Sastry. Synthesizing controllers for nonlinear hybrid systems. In Henzinger and Sastry [18], pages 360–373.

[53] P.K.C. Wang. A method for approximating dynamical processes by finite-state systems. *Int. J. Control*, 8(3):285–296, 1968.

[54] Dinez Yuret. From genetic algorithms to efficient optimization. Master's thesis, Massachusetts Institute of Technology, May 1994.

[55] Feng Zhao, Shiou C. Loh, and Jeff A. May. Phase-space nonlinear control toolbox: The maglev experience. In M. Lemmon, editor, *Proceedings of Hybrid Systems V (HS '97)*, South Bend, IN, USA, 1997. Center for Continuing Education, University of Notre Dame.