On-Line Manipulation Planning for Two Robot Arms in a Dynamic Environment

Tsai-Yen Li and Jean-Claude Latombe
Robotics Laboratory, Department of Computer Science, Stanford University
Stanford, CA 94305, USA

Abstract

In a constantly changing and partially unpredictable environment, robot motion planning must be on-line. The planner receives a continuous flow of information about occurring events and generates new plans, while previously planned motions are being executed. This paper describes an on-line planner for two cooperating arms whose task is to grab parts of various types on a conveyor belt and transfer them to their respective goals while avoiding collision with obstacles. Parts arrive on the belt in random order, at any time. Both goals and obstacles may be dynamically changed. This scenario is typical of manufacturing cells serving machine-tools, assembling products, or packaging objects. The proposed approach breaks the overall planning problem into subproblems, each involving a low-dimensional configuration or configuration$\times$time space, and orchestrates very fast primitives solving these subproblems. The resulting planner has been implemented and extensively tested in a simulated environment, as well as with a real dual-arm system. Its competitiveness has been evaluated against an oracle making (almost) the best decision at any one time; the results show that the planner compares extremely well.

1 Introduction

Off-line robot motion planning is a one-shot computation prior to executing any motion. It requires all pertinent data to be available in advance. In contrast, on-line planning is an ongoing activity that receives a continuous flow of information about events occurring in the robot environment. While planned motions are being executed, new plans are generated in response to incoming events.

Off-line planning is virtually useless in dynamic environments that involve events whose occurrences in time and space are not precisely known ahead of time. On the other hand, while on-line planning can potentially deal with such environments, it raises difficult temporal issues which have not been thoroughly addressed by previous research. Indeed, timing is highly critical since motions must be both planned and executed while their goals are still relevant. Opportunities to achieve a goal may exist only during short periods of time. If the on-line planner is too slow or does not focus on the right subproblem at the right time, it will fail to achieve goals that could have been attained otherwise. The efficiency of the planner measures against an instantaneous oracle making the best decision at every time. The greater the efficiency, the better the planner. But failing to achieve some goals is acceptable. In fact, for the kinds of problems that we will consider in this paper, even a perfect oracle may not achieve all goals.

In this paper we investigate on-line motion planning for a specific, but practically interesting part-feeding scenario where two robot arms must grab parts as they arrive on a conveyor belt and transfer them to given goals without collision. This scenario is typical of robotic cells loading machines, assembling products, or packaging/palettizing objects. Today, imprecise events are eliminated by costly engineering and/or handled by enforcing time-consuming coordination rules. On-line motion planning has the potential to significantly reduce the development time and implementation cost of these cells, while increasing their throughputs. Since the timing of the operations would no longer require off-line prior analysis, the cells would also be more flexible; in particular, they could be dynamically assigned new tasks without interrupting current operations.

Our approach to on-line planning is to break the overall planning problem into a series of subproblems and orchestrate very fast primitives solving these subproblems according to the incoming flow of information. We have implemented a planner embedding this approach and have experimented with it in a simulated environment to evaluate its efficiency against quasi-optimal oracles. The results show that it is quite competitive. We have also connected the planner to a real dual-arm robot system and successfully run experiments with this integrated system.

In our scenario, the transfer of a part to its goal may require "hand-over" operations between the two arms, i.e.: an arm may have to ungrasp the part at an intermediate location (e.g., because the goal is not reachable by the arm), where it will later be regrasped by the other arm. Therefore, the planner must not only compute arm motions. It must also include grasp/ungrasp/regrasp operations between these motions. For that reason we call it a *manipulation planner*.

Section 2 relates our work to previous research and reviews motion planning concepts used in the rest of this paper. Section 3 describes the part-feeding scenario that we use to investigate on-line manipulation planning. Section 4 gives an overview of our planner and Section 5 describes in detail the techniques it uses. Section 6 presents several extensions. Section 7 describes the implementation. Section 8 provides measures of its efficiency in a simulated robot environment. Section 9 reports on the connection of the planner to a real robot system.

2 Relation to Previous Work

Motion planning has attracted a great deal of interest over the last 15 years. Most of the research, however, has focused on off-line planning in static environments. A motion plan is then computed as a geometric *path*. A major concept produced by this research is the notion of the *configuration space*, or *C-space*, of a robot [25]. Various path planning algorithms based on this concept have been proposed [22]. A number of very fast planners have been implemented for robots with few degrees of freedom (usually, 3) [3, 4, 24]. A typical technique consists of exploring a uniform grid in C-space, using a best-first search algorithm guided by a goal-oriented potential field [3]. Reasonably efficient planners have also been developed for robots with many degrees of freedom (6 or more) [3, 9, 14, 15, 21]. But these planners still take too much time to be used on-line; their time performance also lacks consistency.

Existing path planners can be used to facilitate off-line robot programming or to check the feasibility of future operations. For example, the path planner in [9] is used to compute collision-free paths of an 8-dof manipulator among cooling pipes in a nuclear plant. In [12] a planner generates paths of a 5-dof riveting machine to assemble portions of an airplane fuselage. The planner in [6] is used to check for the maintainability of aircraft engines. Today, a number of commercial CAD environments integrate off-line robot programming tools and some include automatic collision checkers and path planners. Such planners will soon be standard components of off-line robot programming systems. However, they will only provide limited help to deal with dynamic, imperfectly known environments. As such environments become more frequent in manufacturing, the need for more sophisticated planning tools increases.

Motion planning in the presence of obstacles moving along known trajectories is a step toward dealing with a dynamic environment. It has been studied in particular in [10, 11, 32, 33], where previous path planning methods have been extended to deal with the temporal aspect of this new problem. Motion plans are generated in the form of robot's *trajectories*, i.e., geometric paths indexed by time. The C-space is extended by adding a dimension, time, yielding the *configuration×time space*, or *CT-space*, of the robot. The obstacles map to a static forbidden region in this space. A trajectory is computed as a curve segment connecting the initial and goal configuration×time points and lying outside the forbidden region. This curve must be time-monotone, i.e., at any time t_0 its tangent must point into the half-space $t > t_0$. When the velocity of the robot is upper-bounded, the tangent must further point into a cone determined by the maximal velocity.

Motion planning for several robots sharing the same space has been addressed in [3, 4, 8, 16]. Two approaches have been proposed. The *centralized approach* consists of treating the various robots as if they were one single robot, by considering the Cartesian product of their individual C-spaces [3, 4]. This space is called the *composite C-space*. The forbidden region in this space is the set of all configurations where one robot intersects an obstacle or two robots intersect each other. A drawback of this approach is that it often leads to exploring a large-dimensional space, which may be too time-consuming. An alternative is the *decoupled approach*, which consists of planning for one robot at a time. In one technique, the robots whose motions have already been planned are treated as moving obstacles constraining the motions of the other robots [8]. This technique requires searching the C-space of the first robot and the CT-spaces of all other robots. Another technique, called velocity tuning, plans the path of each robot separately and then tunes the robots' velocities along their respective paths so that no two robots ever collide [16]. However, the decoupled approach is not complete, i.e., may fail to find a motion of the robots even if one exists.

Manipulation planning extends motion planning by allowing robots to move objects. It consists of interweaving transit paths, where a robot moves alone, and transfer paths, where it moves objects, separated by grasp/ungrasp/regrasp operations. These paths lie in different subspaces of the composite C-space defined as the Cartesian product of the C-spaces of all robots and movable objects. Manipulation planning has been studied for a single robot in [2, 35] and for multiple robots in [18, 19, 20, 21]. The regrasping issue with one robot has been specifically investigated in [34].

3 Scenario

The scenario considered in this paper involves two robot arms, a conveyor belt, an overhead vision system, a working table, movable parts, and obstacles. The arms must grab parts as they arrive on the belt and transfer them to specified goals on the table where, for example, they will form an assembled product.

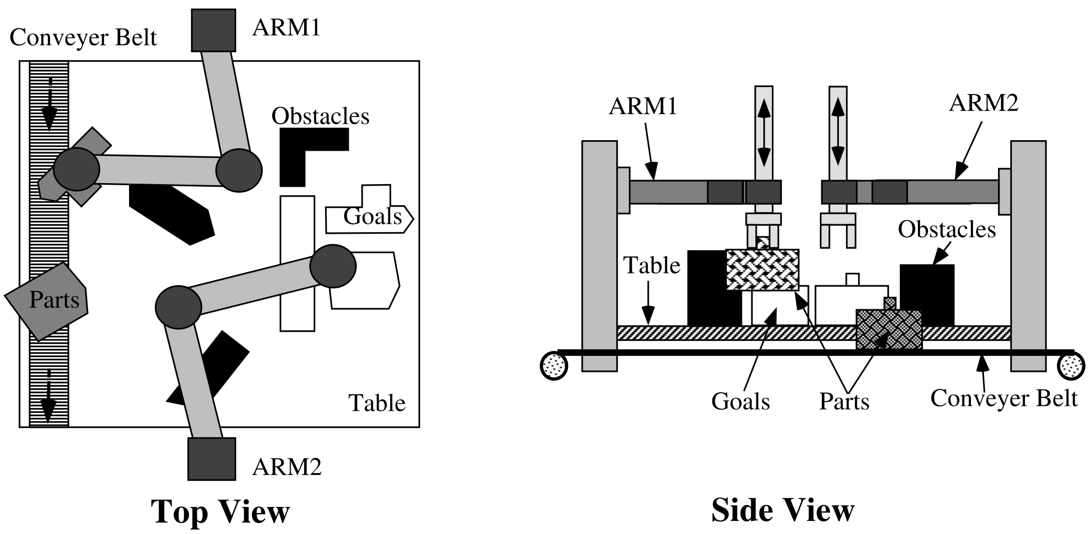

Figure 1: Two-arm robotic cell

The robot system is depicted in Fig. 1. It consists of two identical SCARA-type arms [7], each having three links and four degrees of freedom. The first two links of each arm form a horizontal linkage with two revolute joints. The third link, which carries the gripper, translates up and down. Finally, the gripper can rotate about its vertical axis. Each arm shares part of its workspace with the other arm, so that the same part may be grasped by one arm or the other; space sharing also allows for hand-over operations between the two arms. The arm that is closest to the beginning of the belt is called ARM1. The other arm is called ARM2; it can be seen as a backup for ARM1.

Parts of different types arrive on the belt at any time, in random order, and with arbitrary positions and orientations. An overhead vision system detects and identifies them, and tracks their locations while they are on the belt. The task of the arms is to grab as many parts as possible and transfer them to their goals (shown white in the figure), without collision. For each part X, the position and orientation (relative to X) where an arm's gripper can grasp X is unique and given. The goal of a part of any type is unique and within reach of at least one arm. When an arm releases a part at its goal, the part stays there until it is removed by an external mechanism. We assume that this mechanism never interferes with the arms, so that it is not taken into consideration by our planner.

Static obstacles (shown black in the figure), e.g, fixturing devices and other machines, are lying on the table. If an arm releases a part on the table, this part also becomes a static obstacle, until it is removed. All obstacles on the table lie below the horizontal volume swept out by the first two links of each arm. Similarly, an arm's gripper in its upmost position cannot collide with any obstacle. Hence, if an arm is not holding a part and its gripper is all the way up, it can only collide with the other arm. Such an arrangement is classical for SCARA-type arms, since otherwise motions would be too constrained to perform any useful task. However, when an arm holds a part, this part shares the same space as the obstacles. Hence, no part can be stacked on top of an obstacle or another part. But the belt is low enough so that when an arm holds a part with its gripper in its upmost position above the belt, the part is not hit by other arriving parts. This condition allows an arm to lift a part above the belt and stay there for a while, e.g., waiting for the next motion command.

A single-processor computing resource is dedicated to planning. The planner, which gets all its information about the arriving parts from the vision system, must use this resource to decide on-line which arm motions to execute in order to transfer as many parts as possible to their goals. Ideally, the efficiency of the planner should be measured against an instantaneous oracle that would always make the best decision. The ratio N_p/N_o, where N_p (N_o) denotes the numbers of parts successfully moved to their goals when the planner (the oracle) is in command, computed over a run, defines the efficiency of the planner for that run. The closer to 1, the better.

Example: The above scenario is illustrated in Fig. 2, 3, and 4, with a series of snapshots produced by our planner. Snapshots are indexed by time, with the run starting at time 0 and the sampling rate being 0.25sec per frame. The belt is on the left-hand side and moves downward. Each snapshot displays two configurations of the arms and moving parts; the one in darker grey is the current configuration, while the one in lighter grey is a configuration between the previous and the current snapshot. Parts of two types are fed during the run. We denote them by X_i and Y_j, where X and Y refer to the pentagonal and T-shaped parts, respectively, and i and j indicate the order of arrival. Each part disappears as soon as it is delivered to its goal.

In snapshots (2)-(4), ARM1 (the top arm) and ARM2 (the bottom arm) are simultaneously delivering X_2 and X_1 to their goals. In (6), X_1 reaches its goal and disappears. In (7)-(11) ARM1 performs the deliver motion of X_2, while ARM2 clears the way for this motion. There are two new parts, Y_1 and X_3, arriving on the belt. In (12), immediately after ARM1 has delivered X_1 to its goal, ARM2 starts executing a motion to grasp Y_1. Simultaneously, ARM1 performs a short motion to free the way for ARM2, as shown in snapshots (12)-(13). Snapshots (12)-(21) display the grasp motion of ARM2. Concurrently, in (15)-(21), ARM1 performs a motion to catch X_3. In (21), the arms grab X_3 and Y_1.

In (24)-(34) ARM2 delivers Y_1 to its goal, while ARM1 is staying still holding X_3 above the belt. In (36)-(38) ARM2 clears the way for ARM1, which starts executing the deliver motion of X_3. Note that the goal of X_i has changed between (34) and (36). This change is taken into account in ARM1's deliver motion, as shown in snapshots (40)-(51). In (40)-(42) ARM2 executes a grasp motion to catch Y_2 and starts moving Y_2 toward its goal in (51). In (53), since X_3's goal is not reachable by ARM1, ARM1 releases X_3 at an intermediate location reachable by ARM2. X_3 then becomes an additional obstacle which is taken into account by the deliver motion of ARM2 shown in (56)-(66). In (53)-(60) ARM1 first frees the way for ARM2 and then performs a grasp motion to catch Y_3. In (64)-(66) it starts transferring Y_3, while ARM2 is still delivering Y_2 to its goal. In (68) ARM2 clears the way for ARM1, which delivers Y_3 to its goal in (70)-(76). In (70)-(78) ARM2 changes posture before grasping X_3 in (80) and moving it toward its goal in (82)-(85).

In (70) part X_4 leaves the workspace ungrasped. In (78) ARM1 has finished delivering Y_3 and starts moving toward the belt to grasp Y_4. In (82)-(96), ARM1 grasps Y_4 and moves it to its goal. In (88)-(98), ARM2 moves to grasp X_5. Finally, in (98)-(100), ARM1 starts another grasp motion to catch part X_6.

In the following two sections we describe a planner that tackles the manipulation problem of the above scenario. Next, in Section 6 we will consider extensions of this problem allowing for dynamic changes in the types of parts, the goals of the parts, and the obstacles. We will also consider the case where some parts require being held by two arms simultaneously, in order to be moved from

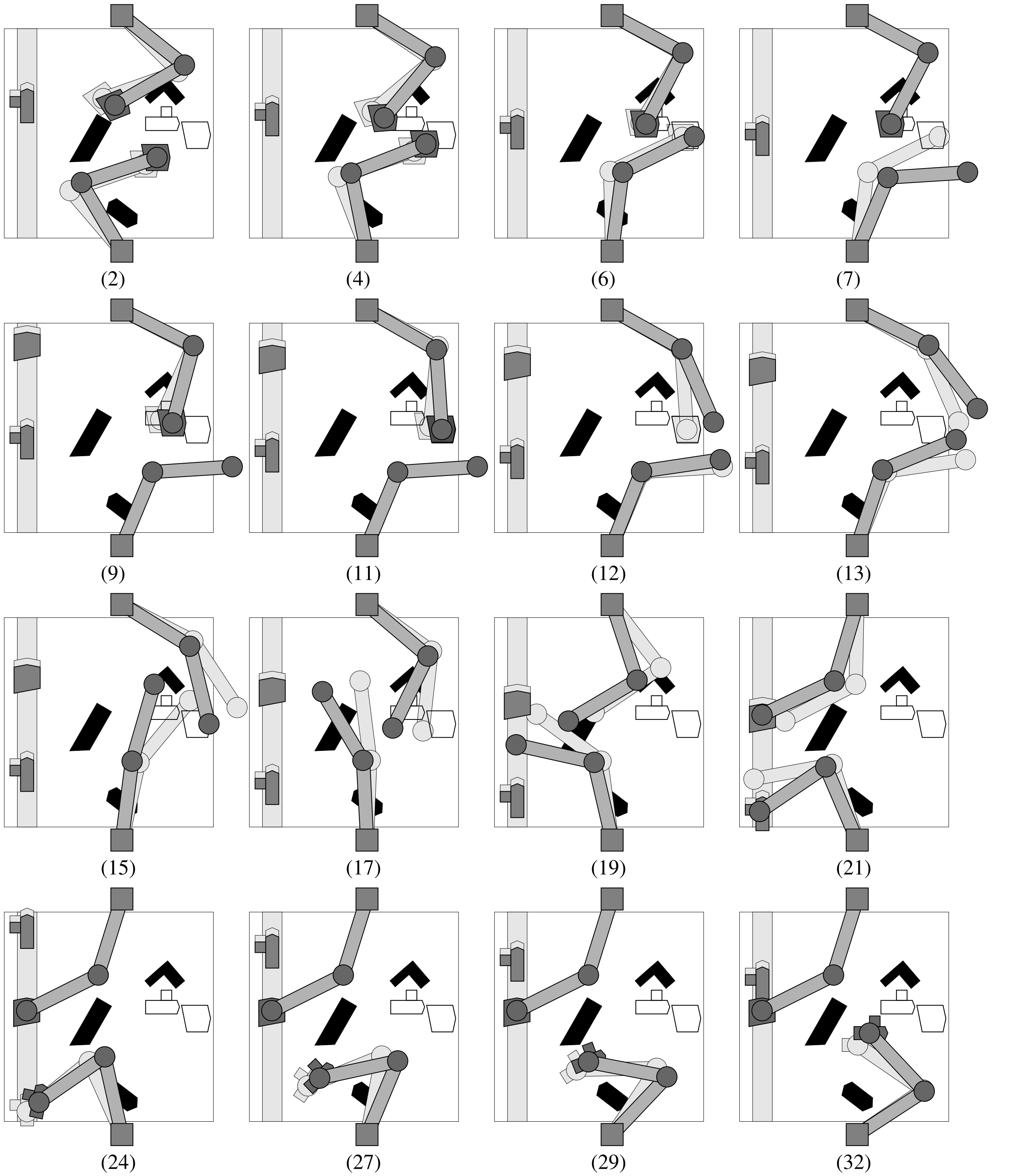

Figure 2: Example (part 1)

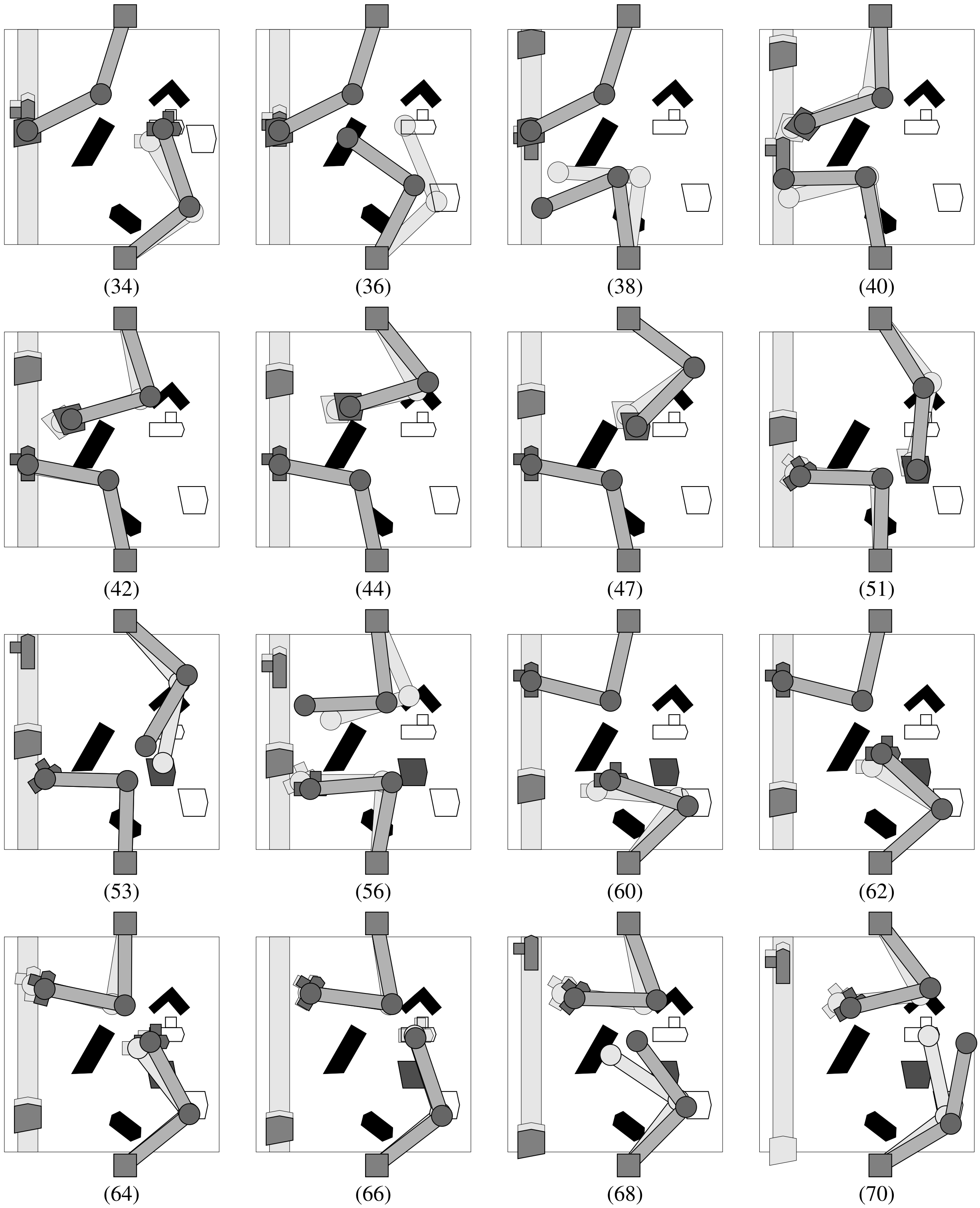

Figure 3: Example (part 2)

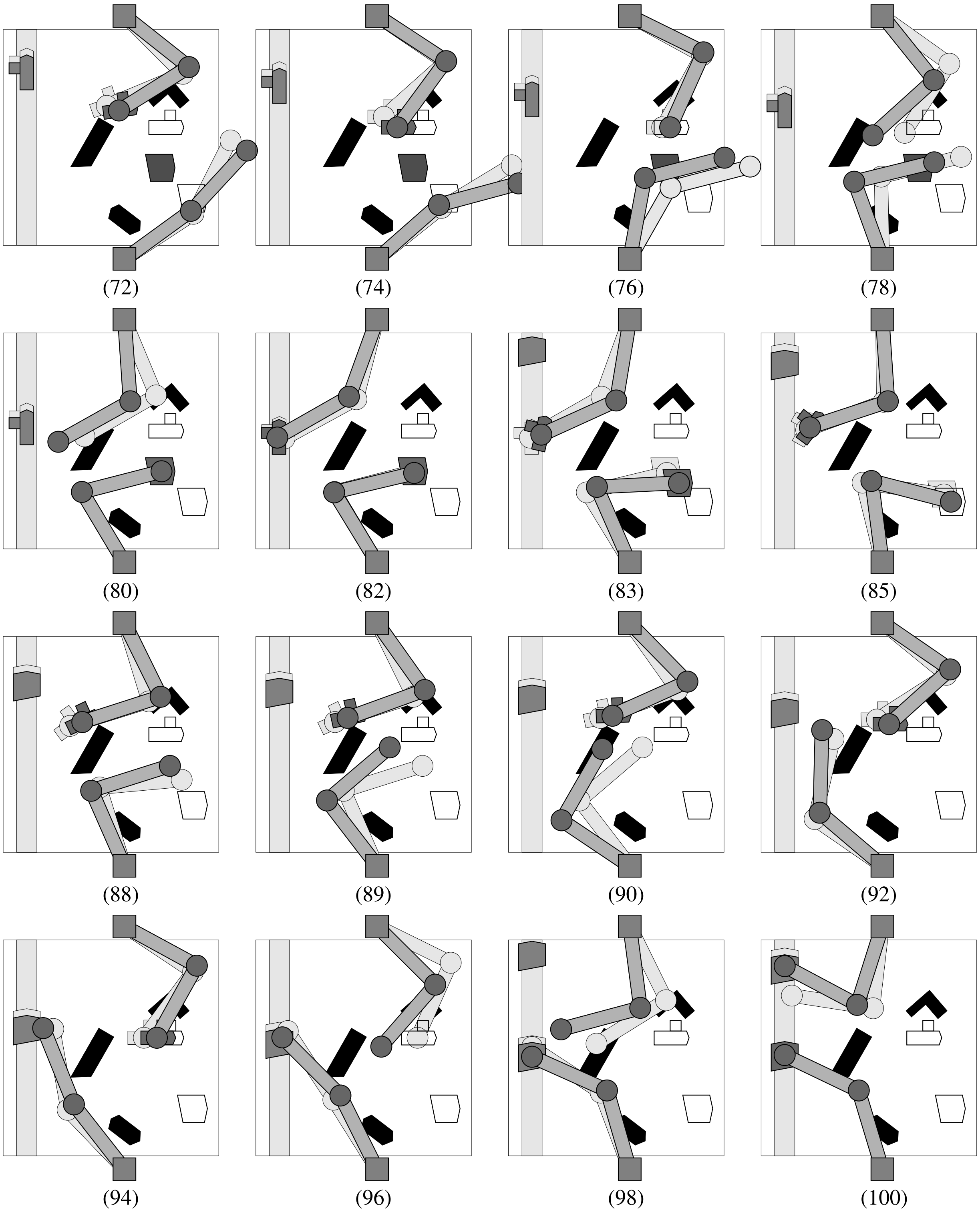

Figure 4: Example (part 3)

one configuration to another.

4 Overview

Our approach to on-line manipulation planning is to orchestrate very fast planning primitives solving a succession of subproblems selected according to the data provided by the vision system. Here, we give an overview of this approach and we state the assumptions and heuristics it relies on. We denote the arms by A_1 and A_2, with A_1 standing for either ARM1 or ARM2, and A_2 standing for the other arm.

4.1 Planning Primitives

In principle, if the arrival times of the parts on the belt were known in advance, the manipulation planning problem of our scenario could be solved off-line by searching through the high-dimensional composite C-space of the two arms and the parts. However, the fact that manipulation is constrained by time, i.e., that parts must be grabbed at the right place at the right time, would still make the problem very tricky. For instance, one can imagine sequences of incoming parts where it is preferable to let a part pass ungrasped, though it could be grasped, because this will free the arms for grabbing more parts later. Today, there exists no planning method that would do the work, without taking prohibitive time to run. So, the problem needs to be simplified.

Obviously, performing planning on-line makes simplifications even more necessary. One way to proceed is to consider several low-dimensional C-spaces, rather than a single, large-dimensional one. However, such simplification yields a planner that is no longer complete. Usually, the more simplification, the faster, but the less complete the planner.

So: How much simplification is suitable? We use the following rule-of-thumb: Planning a motion should take significantly less time than executing this motion. We justify this rule as follows: If planning is longer, the performance of the robot system degrades quickly; but if it only takes a small fraction of the time needed for execution, making it even faster has little effect on the system efficiency. This justification is actually supported by experimental evidence with our implemented planner (see Section 8). The above rule leads us to reduce the planning problem to a succession of subproblems in spaces of dimension three. Indeed, there exist techniques that plan motions in such spaces much under the second on current workstations, while planning in spaces of dimension four or higher takes one or several orders of magnitude longer [3]. Reducing planning to even smaller spaces, if possible, would yield faster, but weaker primitives. The time gain would contribute little to the total efficiency of the robot system; but the greater weakness of the primitives would likely lead to missing opportunities.

Our planner mainly searches through two types of 3D spaces: the CT-space of an arm and the C-space of a part. We briefly discuss below the assumptions and heuristics which allow us to decompose the problem and limit planning to these spaces. A more thorough description will be given in Section 5.

First, let us consider the arms. We allow the first two links of an arm to move only when its gripper is all the way up. Hence, an arm can only collide with the other arm. This allows us to represent the two arms in a 2D workspace as was shown in Fig. 2-4. Links are modeled by rectangles and joints by disks. The disk at the extremity of each second link also includes the vertical link and the gripper. With this representation, planning a collision-free motion of the two arms requires dealing with a 4D C-space. To reduce the problem further, we decouple arm planning so that we always plan for a single arm, say A_1, at a time while the other arm, A_2, can be idling or executing a previously planned trajectory. This problem can be formulated as computing a trajectory in the 3D CT-space of A_1. A part X moving on the belt maps into A_1's CT-space as the set of all tuples (q_1^g, t_g) such that if A_1 is at configuration q_1^g at time t_g, it can grasp X.

To make our presentation shorter, we will assume that translating the gripper to grasp or ungrasp a part is instantaneous, and that orienting the gripper can always be coordinated with the motions of the other two links. We will also consider that parts disappear from the table immediately after they have been delivered to their goals. These assumptions can easily be removed and are not made in the implemented planner.

Let us now consider a part X. We represent X as a 2D object by projecting it on the horizontal plane. When X is being transferred by an arm, the part both translates and rotates in the plane, hence tracing a path in its 3D C-space. Stationary obstacles map into this C-space as a forbidden region that X's path must not intersect. Our planner computes X's path between its grasp and goal configurations in the subset of its C-space that is reachable by at least one arm. Through the arm's inverse kinematics, this path then entails the path of the arm holding X. If X leaves the space reachable by this arm, the planner will command the arm to ungrasp X at an intermediate location on the table where it can be regrasped by the other arm.

One way to make it possible to plan the path of X in its 3D C-space is to require that the arms move no more than one part at a time. But such a condition may lead one arm that has picked up a part to wait for the completion of the other arm's motion before actually moving the part. On the other hand, planning for the motion of two parts X and Y simultaneously entails reasoning in their 6D composite C-space and planning for one part at a time, considering the other part as a moving obstacle, still requires searching a 4D CT-space. This leads us to use the following technique: The planner always computes the path of a part X in its 3D C-space. If another part Y is currently being moved, this motion is temporarily ignored. When X's motion has been computed, its execution is coordinated with the current motion of Y by computing the earliest time when X's motion can start without causing collision.

In addition to the simplifications made above, our planner assumes perfect sensing. It also assumes that the arms can perfectly track the planned trajectories, with each joint being able to instantaneously change velocity. Unlike previous simplifications, these assumptions yield discrepancies between the planning model and the real world. In Section 9 we will discuss how we overcome these discrepancies in order to run the planner with real robots.

4.2 Planning Processes

A crucial issue in on-line motion planning is to react to events by focusing quickly on urgent subproblems and sizing opportunities to grab parts before they vanish. *Planning processes* are the main tool used by our planner to manage its activities over time.

Whenever a new part is detected on the belt by the vision system, a message is recorded in a queue Q. When a part is grabbed or reaches the end of the belt, the corresponding message is removed from Q. At any time, Q lists all the parts currently on the belt or at intermediate locations on the table, along with their current configurations.

Assume that the planner starts with no parts on the belt and the table, and no arms moving. The arrival of a part on the belt triggers planning which consists of selecting a part X in Q (here, there is no choice, but usually there is one) and an arm A_1, and setting up a planning process whose task is (X, A_1), i.e., plan a motion of A_1 to grasp and deliver X to its goal. This process will be terminated upon the completion or the failure of its task. Although its main task is to plan for A_1, this process may also generate a motion of A_2; for instance, if A_2 is currently immobile or if its motion ends before the yet-to-be-planned motion of A_1 is over, the process may command A_2 to free the way for A_1. We call such a motion an *accommodating motion*.

A new process is created whenever a process is killed or interrupts itself to allow for the execution of an already planned motion (process interruption will be presented in the next subsection), and there exist a part $X \in Q$ and a non-moving arm A_i ($i = 1$ or 2) such that neither are currently assigned to a planning process. (The planner considers that an arm is moving as soon as it has sent a motion command to the robot controller, and that it has stopped moving when it is told so by the controller.) Since there are only two arms and each process "consumes" one arm, no more than two processes can exist simultaneously.

The robot operations in our scenario may be accomplished with different orderings of the parts and different assignments of arms to parts. Computing plans for all possible orderings/assignments and choosing the one that can grasp the largest number of parts in the shortest time would take prohibitive time to run. This would even not guarantee to produce the best solution, since this solution may also depend on parts yet to come. Instead, we assign tasks to processes according to the following heuristic rules:

1. The parts at intermediate locations on the table have higher priorities than those on the conveyer belt, since they may obstruct possible paths for these new parts.

2. The parts that are more advanced on the belt have higher priorities than those which are less advanced, since they will leave the arms' workspace earlier.

3. When both arms are not moving, ARM1 has higher priority than ARM2, since ARM2 is at the ending side of the belt and thus can be used as a backup for ARM1.

A planning process with task (X, A_1) may fail to solve its task; for example, it may not find a trajectory for A_1 to grab X from the belt on time. The process is then killed. According to the third rule above, if A_1 was ARM1, then there is still a chance that the process $(X, \text{ARM2})$ will be created. For every part in the queue Q, the planner keeps track of failures to avoid reassigning a combination part-arm to a new process, if that combination is guaranteed to fail again.

4.3 Concurrent Planning

Recall from Section 3 that a single processor is always available for planning. While a planning process is using this processor for a task (X, A_1), other parts may arrive on the belt and the other arm A_2 may be idling. The planner could wait until the current process is killed before creating a new process. However, it may be urgent to plan for A_2, in order to avoid missing parts. But the task of the first process is also urgent. This conflict leads us to break the task (X, A_1) performed by a process into two subtasks: the *grasp subtask* – plan a trajectory to grasp X – and the *deliver* *subtask* – plan a trajectory to deliver X to its goal.

A grasp subtask, in general, is more urgent than a deliver subtask since the latter's goal is time-independent. For that reason, a planning process interrupts and puts itself on hold between the two subtasks, allowing for the creation of a new process. More precisely, suppose that a process P_1 is created to plan for the task (X, A_1). P_1 first plans the motion for the grasp subtask and then puts itself on hold if a feasible motion has been found. While A_1 is executing this motion, the processor is free and can be used by other processes, say P_2, to plan for other tasks. Once A_1 has grasped X, P_1 resumes and solves for the deliver subtask. However, if P_2 is currently running, P_1 is put in the waiting state until P_2 is either interrupted or killed. While waiting for a deliver path, A_1 stays still with its gripper holding X all the way up above the belt, so that parts arriving on the belt can pass below. Processes thus take turns in using the processor.

Hence, a planning process P may traverse the following states during its lifetime:
- *Running*: P is running if it uses the computing resource to compute a plan.
- *On hold*: P is on hold while the arm assigned to it performs the grasp or deliver motion.
- *Waiting*: P is waiting if it needs to compute a delivery motion, but the computer resource is being used by another process.

At any one time, there exist 0, 1, or 2 planning processes. In theory, while one process P is on hold or waiting, several other processes can be successively created (and killed). In practice, the number of these other processes is small; indeed, as soon as one solves its grasping subtask or no new parts arrive on the belt, P will be running again.

If a process whose task is (X, A_1) fails to solve either the grasp or deliver subtask, it is immediately killed. However, in the second case, A_1 is already holding X. Then A_1 releases X on the belt as soon as there is enough distance between two incoming parts. (We will see that if X was picked up from the table, the delivery subtask cannot fail.) Part X is updated in the queue Q accordingly. There might still be a chance that the other arm can accomplish the task.

A planning process P_1 may be unable to plan the motion of an arm A_1 to deliver a part X at its goal because this goal lies outside the space reachable by A_1 or because obstacles force X's path to leave A_1's reachable space. Then P_1 does not fail. It produces a motion of A_1 that delivers X at an intermediate configuration where it can be regrasped by the other arm A_2. Q is updated, P_1 is killed, and it will require another process P_2 (with arm A_2) to move X to its goal. Heuristic rule 1 in Subsection 4.2 will give this part a higher priority than any other part on the belt. This kind of hand-over operation from one arm to the other can happen more than once for the same part.

The interruption of a planning process between the grasp and deliver subtasks relies on the assumption that the deliver subtask will rarely fail. In a realistic robot setting, the obstacles on the table should be distributed to allow incoming parts to be moved from the belt to their goals. But it may occur that, due to hand-over operations, all paths for a part are obstructed by other parts resting at intermediate location on the table and waiting to be regrasped.

This simple process coordination is free of deadlocks. No two processes communicate with each other and the computation carried out by each process before it is killed or interrupted is always finite.

5 Planning Techniques

We now present a detailed account of the activities carried out by the planner within the lifetime of a planning process solving for the task (X, A_1).

5.1 Representation of C-Spaces and CT-Spaces

Each primitive invoked by the planner searches through a low-dimensional C-space or CT-space. This subsection describes how these spaces are parameterized and represented.

As indicated in Subsection 4.1, each arm A_i, $i = 1$ or 2, is modeled as a planar two-revolute-joint linkage, hence has a 2D C-space C_i. We parameterize a configuration q_i of A_i by the arm's two joint angles, which we denote by θ_{i1} and θ_{i2}. Each angle spans an interval of amplitude less than 2π determined by the mechanical stops of the corresponding joint.

We use the following metric over C_i: Let $\omega_{i,1}$ and $\omega_{i,2}$ be the respective maximal velocities of the first and second joints of arm A_i. Let $q_i = (\theta_{i1}, \theta_{i2})$ and $q'_i = (\theta'_{i1}, \theta'_{i2})$ be two configurations of A_i. The distance $D(q_i, q'_i)$ between these two configurations is:

$$D(q_i, q'_i) = \max\{\frac{|\theta_{i1} - \theta'_{i1}|}{\omega_{i,1}}, \frac{|\theta_{i2} - \theta'_{i2}|}{\omega_{i,2}}\}.$$

This definition is consistent with our assumption that arm joints achieve their planned velocities instantaneously: $D(q_i, q'_i)$ then measures the minimal time that A_i takes to travel between q_i and q'_i. The straight-line segment joining q_i and q'_i in C_i is the shortest among all possible paths connecting these two configurations; its length is $D(q_i, q'_i)$.

The CT-space CT_i of A_i is defined as $C_i \times [0, +\infty)$, with C_i parameterized as above and the third dimension being time. At every point $(\theta_{i1}, \theta_{i2}, t)$ in CT_i, the maximum velocities $\omega_{i,1}$ and $\omega_{i,2}$ define a cone of points reachable from $(\theta_{i1}, \theta_{i2}, t)$.

Each part X arriving on the conveyor belt has a 3D C-space. A configuration of X is parameterized by the coordinates x and y of a reference point attached to X in a fixed coordinate system and the angle θ defining the orientation of X relative to this system. The pair (x, y) is called the *position* of X. While an arm is holding X, the arm's configuration determines the position of X.

All C-spaces and CT-spaces searched by our planner are represented as bitmaps. Cells containing "1"s designate the forbidden region where collision occurs. Cells containing "0"s form the free region in which paths and trajectories must lie. We will describe efficient techniques to construct these bimaps in Subsection 5.5. The set of cells in a CT-space bitmap projecting onto the same time interval is called a *time slice*.

The resolution along the two axes of an arm's C-space bitmap is chosen so that, when the arm is fully extended, moving the first joint or the second joint by one increment roughly causes the same displacement ε of the arm's endpoint. The resolution along the time axis of the CT-space bitmap is such that when one joint moves at maximal velocity during an increment of time, the arm's endpoint moves by approximately one increment in the C-space bitmap. The resolution of the C-space bitmap of a part X is the same along the two dimensions representing X's position and roughly equal to ε. The resolution along the orientation axis is such that when X rotates by one increment about its reference point, the point of X that undergoes the maximal displacement (i.e., the point in X the furthest away from the reference point) moves by approximately ε [3].

5.2 Planning a Grasp Motion for a Part on the Belt

Let us now consider the grasp subtask of the task (X, A_1). X may either be a part arriving on the belt, or a part previously ungrasped at an intermediate location on the table. The two cases receive basically the same treatment, with a few minor differences. Here we consider the first case, which is also the most frequent.

We let t_c stand for the current time. Hence, t_c is continuously changing value. We let q_1^s designate the configuration of A_1 when it starts executing the grasp motion.

For any given time t, we let $q_1^g(t)$ denote the configuration of A_1 at which it can grasp X. The map q_1^g is defined over the time interval $[t_g^{min}, t_g^{max}]$ during which X is on the belt within A_1's reach. It may yield two configurations, since the inverse kinematic equations of A_1 usually have two distinct solutions corresponding to two postures of the arm. The planner always selects the configuration which is closest to q_1^s according to the metric D. This is a reasonable choice since this grasp configuration is likely to be the quickest to reach. (An alternative would be to successively apply the planning techniques presented below to both configurations defined by the map q_1^g and select the configuration allowing for the earliest grasp. This variant would roughly take twice as much planning time, but could produce a better motion plan once in a while.)

We distinguish between two cases: In (A), the second arm A_2 is not moving; in (B), it is moving.

(A) Arm A_2 is not moving: While a grasp motion is being planned and then executed, X keeps moving on the belt. To plan A_1's motion, we must know when and where X can be grasped, which in turn depends on how much time it will take to plan and execute this motion. This difficulty leads us to iteratively determine the time t_g at which A_1 will grasp X.

We initially schedule the grasp at the latest possible time, i.e., we set $t_g = t_g^{max}$. *If A_2 is not holding a part*, the planner generates A_1's path between q_1^s and $q_1^g(t_g)$ as the straight-line segment connecting these two points in C_1. An accommodating motion of A_2, if needed, is generated in CT_2, into which A_1's path maps as a forbidden region. Since actual velocities will be computed later, we momentarily take time equal to the abscissa along A_1's path. A_2's motion is computed in the form of a time-monotone curve, i.e., a curve whose tangent at any time t_0 points into the half-space $t > t_0$. This curve joins $(q_2^s, 0)$ to some (q_2^e, L), where q_2^s denotes the current configuration of A_2, q_2^e stands for any configuration of A_2 where it does not collide with A_1 at the grasp configuration $q_1^g(t_g)$, and $L = D(q_1^s, q_1^g(t_g))$.

If the line segment joining $(q_2^s, 0)$ and (q_2^s, L) in the bitmap representing CT_2 traverses "0" cells only, no accommodating motion of A_2 is needed. Otherwise the accommodating trajectory is constructed by searching the CT_2's bitmap in a depth-first manner for a sequence of free cells connecting the cell containing $(q_2^s, 0)$ to the time slice containing L. At every iteration of the search, let c be the last cell reached (initially, c is the cell containing $(q_2^s, 0)$). The algorithm considers the seventeen cells adjacent to c in the same or next time slice, and moves to a closest free cell that is not already in the current path (the distance between two cells being measured as the distance D between the projections of their centers into C_2). This greedy algorithm tries at each step to minimize the time that will be required to execute the accommodating trajectory, but by no means does this guarantee an optimal result. If all of the seventeen cells are non-free or already part of the current path, the search backtracks and possibly fails.

If A_2 is holding a part Y, the planner could proceed as above. But it would also have to make sure that Y does not hit any obstacle on the table during the accommodating motion of A_2. This additional check would yield longer computation time. For this reason, the planner does not make A_2 accommodate to A_1's motion. Instead, it treats A_2 as a static obstacle and searches C_1 for a path of A_1 avoiding this obstacle.

At this stage, the planner knows the geometry of the coordinated paths of A_1 and A_2 (possibly, A_2's path is void). It then computes the velocity profiles of the joints to execute these paths in minimal time. This is done by discretizing the curvilinear abscissa along A_1's path into small intervals and, in each interval, letting the joint that takes the longest time at maximal velocity set the pace for the other joints. This computation yields the duration δ of the coordinated motion, hence the latest starting time $t_s = t_g - \delta$ for the motion. If t_s is smaller than the current time t_c, grasping X with A_1 is considered impossible and the planning process is killed; the pair (X, A_1) will not be reassigned to a planning process. Otherwise, the grasp scheduled at t_g is feasible. But if $t_s - t_c$ is rather large, say, more than a few times the time spent planning the motion, executing this motion will lead A_1 to wait for the arrival of X. A better motion may then be possible and the planner iterates the above procedure by scheduling an earlier grasp time t_g. At the end of every iteration, the latest starting time of the best motion computed so far is used to bound the computation time allowed to the next iteration. If one iteration exceeds the time allocated to it, it is aborted and the best motion computed so far is executed. The successive times t_g are chosen by dichotomically decomposing the interval $[t_g^{min}, t_g^{max}]$.

A motion computed as above is shown in snapshots (12)-(21) of Fig. 2. In this example, A_1 is ARM2 moving to grasp Y_1; in snapshots (12)-(13), ARM1 performs a short accommodating motion the clear the way for ARM2. (The grasp motion of ARM1 shown in snapshots (15)-(21) is generated by the technique described in Case (B) below.)

(B) Arm A_2 is moving: The motion being performed by A_2 constrains the future motion of A_1. It is mapped to a forbidden region in CT_1 and the trajectory of A_1 is computed between some (q_1^s, t_s) and some $(q_1^g(t_g), t_g)$, where $t_s > t_c$ stands for the starting time of A_1's motion and $t_g \in [t_g^{min}, t_g^{max}]$ is the time when A_1 grasps X.

Let us temporarily assume that A_2's motion is scheduled to end after time t_g^{max}. Therefore, if A_1 can grasp X, its motion will terminate before the one of A_2. As in case (A), the planner iteratively determines t_g. For every t_g such that A_1 at $q_1^g(t_g)$ does not obstruct A_2's trajectory at any time $t \geq t_g$, it searches for a trajectory joining the line $\{(q_1^s, t) | t > t_c\}$ to $(q_1^g(t_g), t_g)$, avoiding the forbidden region, satisfying the joint velocity constraints, and starting after t_c.

The planner performs the search backward, from the selected $(q_1^g(t_g), t_g)$ toward the line $\{(q_1^s, t) | t > t_c\}$, using a best-first technique. At every iteration of the search, it selects a pending node (q_1, t) of the current search tree such that $D(q_1^s, q_1)$ is minimum over all pending nodes. It computes nine potential successors of this node by successively setting the velocity of each joint to zero, its maximal value with positive sign, and its maximal value with negative sign, and integrating the corresponding motion over the duration of a time slice in the bitmap representing CT_1. If a potential successor belongs to a "0" cell c of this bitmap and c has not been visited before, then it is included in the search tree as a new pending node and c is marked 'visited'. The actual point is recorded in the search graph, so that the velocity bounds along the entire trajectory are perfectly respected. The best-first algorithm and the definition of D guarantee that the computed trajectory takes minimal time over all valid trajectories in the discretized search space. The search fails when no leaves in the search tree lie in time slices occurring after t_c.

The planner selects the successive values of t_g in increasing order between t_g^{min} and t_g^{max}, at the centers of the time slices in CT_1's bitmap. If the search fails for one value of t_g and another value is considered, the new search will discard every node's successor lying in a cell visited by a previous search. Indeed, at the bitmap resolution, this successor cannot be on a valid trajectory, otherwise the previous search would not have failed. Therefore, each new value of t_g usually yields a small amount of additional computation, and the total number of search nodes generated to compute A_1's trajectory is at most equal to the number of free cells in CT_1's bitmap. At soon as the planner succeeds in finding a trajectory for A_1, this motion is executed.

Let us now consider the case where A_2's motion is scheduled to end at time $t_2 < t_g^{max}$. Beyond t_2, A_2 may then perform another motion to clear the way for A_1. To take advantage of this possibility, the planner proceeds as follows: For every selected value of t_g greater than t_2, it generates A_1's grasp trajectory as the concatenation of two trajectories: one connects q_1^s to some q_1^i chosen as the closest configuration to $q_1^g(t_g)$, outside the forbidden region at time t_2; the other connects q_1^i to $q_1^g(t_g)$ and may require an accommodating motion of A_2. The second motion is computed first (if $q_1^i \neq q_1^g(t_g)$) using the techniques of case (A), with q_1^i substituted for q_1^s and q_2^s replaced by A_2's expected configuration when it terminates its current motion. If the latest starting time t'_s of the computed motion is less than t_2, the motion cannot be performed, and the planner tries the next value of t_g. Otherwise, a trajectory of A_1 between $\{(q_1^s, t) | t_c < t < t'_s\}$, and (q_1^i, t'_s) is generated using the above backward best-first search algorithm, with A_2 mapping into the same forbidden region in all the time slices of CT_1 between t_2 and t'_s. If this computation yields a starting time greater than t_c, the motion is executed; otherwise the next value of t_g is considered. Again, the marking of the cells in CT_1's bitmap saves considerable time.

An example of a motion computed as above is the motion of ARM1 to grasp X_3 in (15)-(21) of Fig. 2. This motion terminates approximately at the same time as the ongoing grasp motion of ARM2 and requires no accommodating motion of ARM2. Another example is the motion of ARM2 to catch Y_2 in (40)-(42) of Fig. 3; this short grasp motion ends before the ongoing transfer motion of ARM1 terminates. A third example is the motion of ARM1 to grasp Y_3 in (53)-(60). A fourth example is the motion of ARM2 to grab X_5 in (88)-(98); this motion ends after the ongoing deliver motion of ARM1, but does not require ARM1 to perform an accommodating motion. A fifth example is the motion of ARM1 to grasp Y_4 in (78)-(82) of Fig. 4. This motion required no accommodating motions of ARM2. A sixth example is the motion of ARM1 to grasp X_6 in (96)-(100).

5.3 Planning a Grasp Motion for a Part on the Table

In this case, X is not moving; hence, time is slightly less critical. We let q_1^g denote the configuration of A_1 where it can grasp X; if two such configurations are feasible, we select the one that is closest to q_1^s. Again, we distinguish between cases: (A) A_2 is not moving; (B) it is moving.

(A) Arm A_2 is not moving: This case could be treated like case (A) in the previous subsection, with one difference: there is no need for guessing successive values of the grasp time. However, we proceed in a slightly different manner. Whether A_2 is holding a part or not, we first treat it as a static obstacle and we try to generate a path of A_1 avoiding this obstacle. If such a path is found, it is executed. Otherwise, if A_2 is not holding a part, we plan a motion of A_1 along a straight-line path and we make A_2 accommodate to this motion.

The first tactic, which treats A_2 as a static obstacle, aims at avoiding an accommodating motion of A_2, since while executing such a motion, A_2 cannot grab a new part on the belt. However, this tactic tends to fail more often than the second one, which makes A_2 accommodate. Since X is not moving, we can afford to waste a short amount of computation time trying the less reliable tactic first and using the other tactic as a backup.

(B) Arm A_2 is moving: The treatment is as in case (B) of the previous section, with t_g chosen at the centers of the successive time slices beyond the current time. Since X does not move, one iteration eventually succeeds.

In snapshots (68)-(80) of Fig. 3-4, the motion of ARM2 to grasp X_3 on the table illustrates this case. This motion changes ARM2's posture, because grasping X_3 with the other posture is not feasible.

5.4 Planning a Deliver Motion

Like in the previous subsection, the goal of the motion is time-independent; but now we must plan for both X and A_1.

(A) Arm A_2 is not moving: The planner first generates a path connecting the initial and goal configurations of X by conducting a best-first search in the bitmap representing X's C-space. This search is guided by a goal-oriented potential field similar to the NF2 function described in [22] and is restricted to the configurations of X where A_1 and/or A_2 can grasp X, as proposed in [21]. An alternative, which could save time-consuming hand-over operations, is to first restrict the search to the configurations of X where A_1 can grasp X; only if this search fails, the configurations reachable by A_2 would also be considered. The generation of a path for X may fail due to parts previously placed on the table by the arms. Then A_1 puts X down on the belt or the table at its current location and the planning process is killed. The planner will not reassign the task to grasp X to any arm as long as none of the parts currently on the table has been removed.

If a path is found for X, it entails a path for A_1 through the arm's inverse kinematics. The initial posture of A_1 is the one at the end of the previous grasp path. If in this posture one joint of A_1 reaches a limit, while changing posture would allow A_1 to continue transferring X, the planner includes an ungrasp operation in A_1's path before the joint limit is attained, then a subpath to change A_1's posture, and finally a regrasp operation, before resuming tracking X's path. A_1 may change posture several times along X's path. If A_2 lies along the way of A_1's path, a motion of A_2 to clear the way for this path is generated. As much as possible, we would like to avoid a long accommodating motion of A_2, since during this motion A_2 cannot grasp new parts. The planner proceeds as follows:

- It first tries to generate a path of A_2 to clear the way for A_1. This motion is planned in C_2, into which the discrete sequence of configurations describing A_1's path (minus the subpaths changing A_1's posture) maps as a union of forbidden regions. The final configuration of A_2's is any configuration outside this union. If a path is found, A_2's final configuration is mapped to a forbidden region in C_1 and the subpaths to change A_1's posture are computed then. The path of A_2 is executed at maximal velocity. The motion of A_1, also at maximal velocity, starts as soon as it can no longer collide with A_2. To determine the starting time of A_1's motion, the planner maps A_2's trajectory to a forbidden region in CT_1. It then represent A_1's trajectory as a curve segment in CT_1 with its initial point at the time when A_2 is scheduled to terminate its motion. Finally it translates this curve toward smaller values of time. The position of the curve just before it intersects the forbidden region due to A_2 gives the starting time of A_1's motion.

- If the previous computation fails to generate paths for A_1 or A_2, the planner computes an accommodating motion of A_2 as in case (A) of Subsection 5.2. Prior to this computation, it completes the path of A_1 by inserting the subpaths changing the arm's posture. These subpaths are simply straight-line segments in C_1.

If X's path leaves A_1's workspace, the planner commands A_1 to put down X at a intermediate configuration where it can be regrasped by A_2 and the planning process is killed. The planner memorizes that A_2 will have to be assigned to the regrasp of X. Note that when A_1 releases X, there exists a path for X to its goal. But this path may later be obstructed by additional parts ungrasped on the table. For that reason, the planner computes regrasp motions for parts on the table by reversing the chronological order in which they have been ungrasped. In this way, there is no need to recompute paths for these parts.

In snapshots (24)-(34) of Fig. 2-3 the deliver motion of ARM2 to transfer Y_1 to its goal was computed as above. The path computed for Y_1 directly entailed a path of ARM2 free of collision with ARM1. Snapshots (36)-(51) illustrate the above planning techniques in a more complicated case, in which ARM1 must move X_3 to its goal. This goal was changed between (34) and (36) and is now out of reach of ARM1. The path of X_3 entails a path of ARM1 that collides with ARM2. Thus, a motion of ARM2 is planned to clear the way and is executed in (36)-(38). It is followed by the motion of ARM1 in (40)-(51). The path of X_3 is then close to leaving the space reachable by ARM1; so, in (51), ARM1 ungrasps X_3.

(B) Arm A_2 is moving: The planner generates X's path as in case (A), with one difference: If A_2 is currently transferring a part to an intermediate configuration, the part at this configuration is treated as an additional obstacle for X. The path of X entails a path of A_1. A_1's motion at maximal velocity along this path is coordinated with A_2's current motion using the same technique as above, that is, by translating A_1's trajectory toward smaller values of time. However, if A_2 holds a part Y, collision must be avoided not only between A_1 and A_2, but also between X and Y. This is done by using a separate bitmap representing the C-space of X relative to Y. This check guarantees that if X's goal lies along Y's path, X will not be moved to an obstructing location before Y has already been through that location.

There is an additional difficulty. It may happen that the final configuration of A_2 lies along A_1's path. Then let t_2 denote the time when A_2's motion is expected to terminate. A_1's motion is coordinated with A_2's motion by mapping A_2 into CT_1 until time t_2 only. Thus, A_1 will execute as much as possible of its motion prior to t_2. At t_2, the motion of A_2 ends and the motion of A_1

is also temporarily stopped. We then plan a motion of A_2 to clear the way for A_1 as in case (A). Though the planning process for A_1 is idling between the time A_1 starts moving and t_2, it is not put on hold; since A_2 is moving, this arm could not be assigned to another part anyway.

The above computation is illustrated with the motion of ARM2 in snapshots (51)-(66) of Fig. 3. This motion was planned and is executed while ARM1 is still moving. Since ARM1 is going to release X_3 at an intermediate location, this part will become an additional obstacle that is taken into account by the planner when it computes the path of Y_2 (see (60)-(64)). Here the final configuration of ARM1 lies along the way of ARM2, which requires planning a motion of ARM1 to clear the way. As indicated above, the motion of ARM2 is temporarily stopped (see (53)). The new motion of ARM1 is executed in (53)-(56); it precedes ARM1's grasp motion in (56)-(60). A second deliver motion (arm ARM1) computed as above is shown in (64)-(76). Because ARM2 obstructs the path of ARM1, a motion of ARM2 to clear the way is first planned and executed in (68). Notice that immediately after, starting in (70), ARM2 starts executing a grasp motion to grab the part X_3 lying on the table. This motion was computed after the motion of (68) was executed; the planner then realized that it was safe to execute it concurrently with the ongoing motion of ARM1. A third illustration of the above computation is the motion of ARM1 shown in (82)-(96).

5.5 Bitmap Construction

The role of a bitmap representing a C- or CT-space is twofold. It provides a discretization of a continuous space prior to searching that space. It also allows for quasi-instantaneous collision checks. Of course, we must consider the cost of generating the bitmap, but most of this computation can be done in a preprocessing phase. Furthermore, computing an entire bitmap is often not more time-consuming than performing a few explicit collision checks in the workspace. The idea of using precomputed bitmaps to accelerate collision checking is also used in [21, 23].

Part's C-space: The C-space bitmap for a part X represents the forbidden region created by the obstacles. We model both X and the obstacles as unions of convex polygons, $\{X_i\}_{i=1,2,\ldots}$ and $\{O_j\}_{j=1,2,\ldots}$, respectively. Every pair (X_i, O_j) of convex polygons yields a subset of the forbidden region in X's C-space. Any cross-section of this subset at a constant orientation of X is itself a convex polygon that is computed in time linear in the number of vertices of X_i and O_j [13, 25]. A polygon-filling function transforms this polygon into a 2D bitmap. The 3D C-space bitmap of X is constructed by stacking fixed-orientation 2D slices. Each slice is generated by computing the forbidden regions due to all pairs (X_i, O_j) and drawing them into the same 2D bitmap.

If all obstacles were fixed, the C-space bitmap for a given type of part would only be computed once. However, the obstacles include parts that have been temporarily released on the table. In the next section we will also allow dynamic changes in the obstacles. Whenever there is a change in the obstacles, the bitmap must be updated. To reduce updating costs, we precompute a bitmap representing the forbidden region corrsponding to every pair part-obstacle and part-part. The size of this bitmap is just large enough to enclose the forbidden region; hence, it is much smaller than the full C-space bitmap. The C-space bitmap is first computed by filling it with "0"s and copying the "1"s of each individual bitmap in the right cells. Whenever an object is removed from the table, the bitmap is recomputed in the same way. When a new object is placed on the table, "1"s are added to the C-space bitmap according to the individual bitmaps involving this object.

In case (B) of Subsection 5.4, we allow a part X to move while another part Y is already moving. This requires performing collision checks at various configurations of X and Y. The part-part bitmap corresponding to the types of X and Y is used there. Whenever a collision check is needed, the relative configuration of X and Y is computed; this configuration determines the bitmap cell to look into.

Arm's C-space: A technique to compute the 2D C-space bitmap of an arm A_1, given the configuration of the other arm A_2, is to enumerate all configurations in the bitmap (e.g., the centers of the cells) and, for each one, test if it is collision-free. Collision checking between arms requires considering four or three pairs of links, depending on whether the first links of the two arms can touch each other, or not (in our setting, only three pairs of links must be considered). To check if two links collide, we simply look into a link-link bitmap. This bitmap is precomputed using the technique of the previous paragraph by treating one link as a fictitious robot free to translate and rotate in the plane and the other link as an obstacle. This technique is reasonably fast, but it can be improved as follows.

In each arm, we choose the reference point of the second link at the center of rotation of the second joint. Thus, if we fix the first joint angle θ_{11} of arm A_1, the reference point of the second link is also fixed. Given the configuration of A_2, we scan all possible values of θ_{11} in C_1's bitmap. Each value determines a cell in two bitmaps, each representing the interaction between the first link of A_1 and a link of A_2. If the first link of A_1 collides with a link of A_2, the whole column in C_1's bitmap is filled with "1"s. Otherwise, the position of the reference point defined by the current value of θ_{11} determines a column in the bitmaps representing the interaction between A_1's second link and each of the two links of A_2. After shifting these two columns appropriately (to align their origins with the origin of the column of C_1's bitmap at the current θ_{11}) and removing the cells beyond the second-joint mechanical stops, we compute their boolean union and copy the result into the column of C_1's bitmap at the current value of θ_{11}. In our implementation this improvement cut the time to compute an arm C-space bitmap by almost two orders of magnitude.

Other efficient techniques to compute C-space bitmaps for articulated arms are proposed in [5, 26, 27].

Arm's CT-space: The 3D bitmap representing an arm's CT-space is computed one time slice at a time. The computation of the 2D bitmap in one time slice is done as above, only when the planner needs this time slice. A 2D bitmap is memorized at least as long as it is beyond the current time.

6 Extensions and Improvements

Changes in goals: The goal of a part can be changed at any time. If a part X arrived before its goal changed, but the deliver motion has not been computed yet, the new goal will be used by the planner when it solves for the deliver subtask. If, instead, the deliver motion has already been planned, it is executed without modification. However, immediately after X reaches its previous goal, the planner plans a new motion to transfer it to its new goal. Similarly, if X has been delivered to its goal and this goal changes prior to the removal of X by the external mechanism, the planner generates a motion to transfer X to the new goal.

One possible exploitation of this planner's ability is the following: Let the parts be used to assemble several copies of a product. Parts delivered to their goals are removed only when a product is complete. However, since parts arrive in random order, too many parts of one type may arrive during a period of time. Because the goals of these parts are occupied, the arms will let them pass ungrasped. Instead, one can define several sites for assembling the product. Initially, the goal of each part is in the first site. When this goal is filled, a new goal is set in the second site, and so on. Whenever a site contains a complete product, this product is removed and all the goals in that site become free again.

New types of parts: New types of parts can be dynamically introduced. For the user, adding a new part means describing its geometry, defining its goal, and specifying the grasp position of a gripper. For the planner, it only requires computing new bitmaps. As long as the number of obstacles and parts of different types is not too large, this computation can be carried out on-line without significantly weakening the total system performance.

Changes in obstacles: The positions and orientations of the obstacles can be changed. However, we impose that such changes happen when no deliver motion is being executed. They only require the planner to update the C-space bitmaps of the incoming parts. This modification is very fast.

Obstacles can also be added or removed. Whenever an obstacle is added, new bitmaps describing the interaction of this obstacle with the various types of parts that may be fed are computed.

Cooperative manipulation: We allow the introduction of parts that require two arms to be moved, say, because they are too heavy for a single arm, or because they have elongated shapes. Two grasp positions are defined for the grippers on each such part.

Let us assume that such a part X arrives on the belt. The planner assigns it to a planning process only if the two arms are non-moving. The two arms are also jointly assigned to this process. The planner generates the grasp motion very much like in case (A) of Subsection 5.2 by iteratively guessing a grasping time t_g. However, at each iteration it must compute the coordinated motion of the two arms: First, it generates the path of ARM1 to attain the grasp position on X (at t_g) that is closest to the beginning of the belt. This path is simply constructed as a straight-line segment in ARM1's C-space. Next, the planner generates a coordinated path for ARM2 to attain the other grasp position on X (at t_g). This motion is computed by searching through ARM2's CT-space, with time taken equal to the abscissa along ARM1's path. Planning for the deliver subtask is done like in Subsection 5.4. No hand-over operations are possible, but the planner may put down the part on the table to change an arm posture or swap grasps (see [20]).

Fig. 5 shows a series of snapshots in which the two arms perform a coordinated motion generated by the planner to grasp the same long part and transfer it to its goal. The transfer requires ungrasping the part at an intermediate location to change the posture of both arms. Note that during the grasp and deliver motions more parts arrive on the belt. Since no arm is available to grab them, they pass ungrasped.

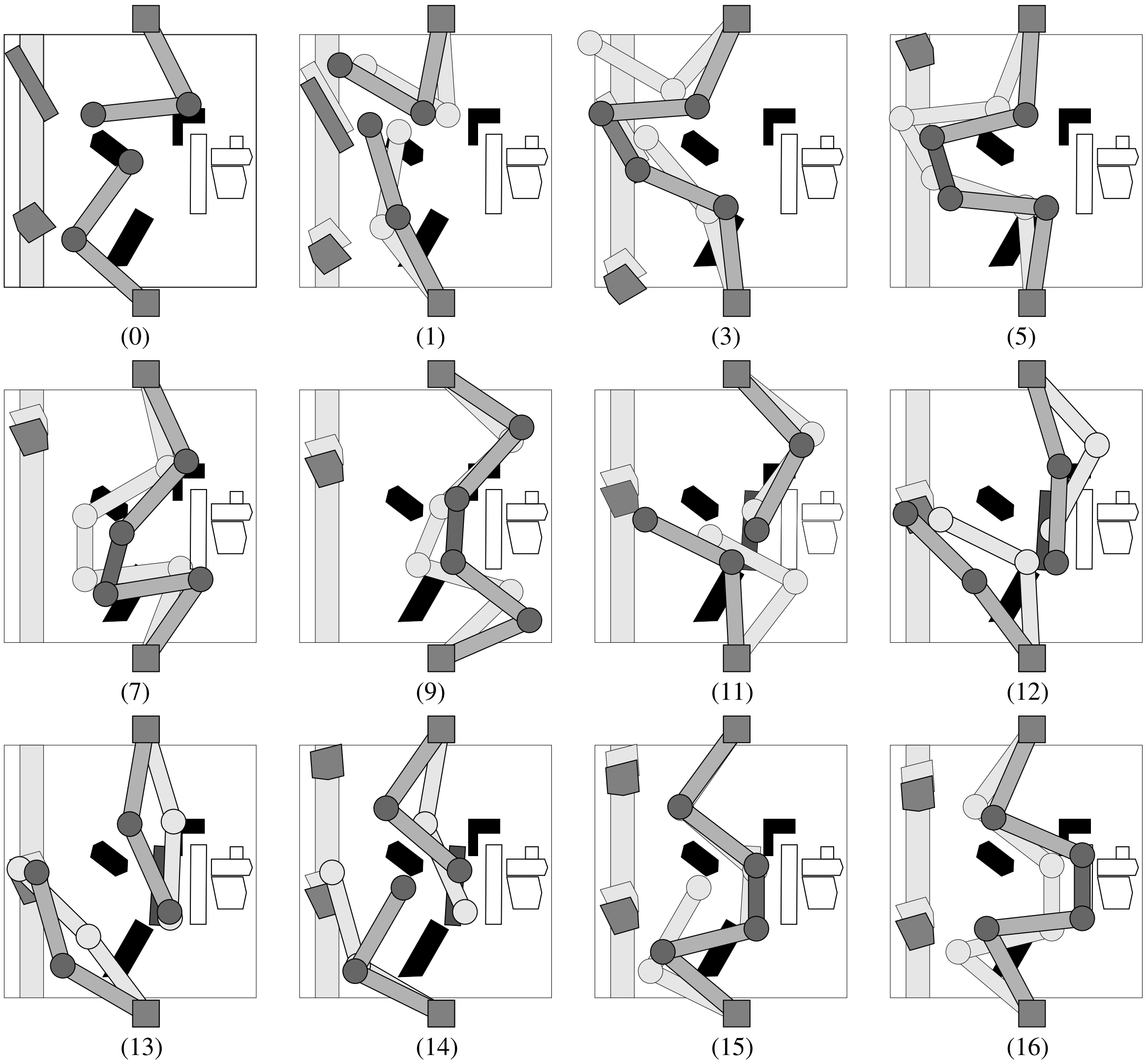

Figure 5: Cooperative manipulation of a long part

Anticipating catches: The process coordination described in Section 4 may sometimes let the planner idling. In fact, rather than idling, if an arm is not moving, the planner then generates a motion for that arm to bring it to a predefined configuration where its gripper is close to the belt. Thus, when a new part arrives on the belt, the arm will be in a better position to catch it quickly.

We could extend this idea and keep the planner always busy by making it work on what may happen beyond the next round of motions. To be really fruitful, however, this generalization requires additional study.

7 Implementation

The planner described in Sections 4, 5, and 6 has been fully implemented in C on a DEC Alpha workstation (Model Flamingo) running under DEC OSF/1. This machine is rated at 126.0 SPECfp92 and 74.3 SPECint92 on the SPECMARKS benchmark. The planner has been connected to both a robot graphic simulator and a real robotic system.

The robot system in our simulator has the same general characteristics as the real system. The lengths of the first and second links of each arm are both 24in. The first joint rotates within a 135dg interval and the second in a 285dg interval. The maximal velocities of the joints are 15.2dg/sec. The belt moves at 4in/sec. The interval of time during which a part can be grasped is approximately 15sec.

An arm C-space bitmap has size 36×76, which corresponds to increments along each axis of about 3.75dg. The size of the bitmap representing a part C-space is on the order of $128 \times 128 \times 96$; the increments along each of the two position axes are approximately 0.57in long. Having the same increments along all angular axes allows for simplifications in the planner's code; nevertheless, setting the size of the increments as suggested in Subsection 5.1 raises no particular difficulty and should be done in a new version of the planner.

We performed various tests with our software to measure the running times of some of its key components. We obtained the following average times for a representative sample of components:

- Computing a bitmap representing the interaction between two objects takes 41ms.[1]
- Computing a complete C-space bitmap for a new type of part using the precomputed part-obstacle and part-part bitmaps takes 8.3ms.
- Updating a part C-space bitmap when an object is added onto the table takes 5.6ms.
- Constructing an arm C-space bitmap using the precomputed link-link bitmaps takes 0.4ms.
- Searching a part C-space bitmap with the best-first search technique of Subsection 5.4 is done at a rate of 65,000 nodes/sec.
- Searching an arm CT-space with the best-first search technique used in case (B) of Subsection 5.2 is done at a rate on 870,000 nodes/sec.

The sequence of snapshots shown in Fig. 2-4 was produced by our planner connected to the graphic simulator.

8 Evaluation in Simulated Environment

We have generated several measures of the efficiency of the planner connected to the graphic simulator, by running it on multiple sequences of arriving parts. Each run lasts 8min, during which on the order of 100 parts are being fed. No parts require being moved by two arms simultaneously. Parts disappear immediately after they are delivered to their goals, so that no part is ungrasped because its goal is occupied. The simulator uses the same model of the physical world as the planner. We define the *missing ratio* of the planner over a run as the number of parts that go ungrasped, in percents of the total number of parts fed during this run.

[1] Our implementation uses no hardware-implemented polygon-filling function to compute such a bitmap.

Feeding Uncertainty (sec)	0.0	0.5	0.5	0.75	0.75	1.0	1.0
Slowing Down Feeding Rate	No	No	Yes	No	Yes	No	Yes
Number of Parts per Minute	13.19	13.19	11.88	13.19	11.32	13.19	10.81
Missing Ratio (Oracle)	0%	20%	0%	25%	0%	28%	0%
Missing Ratio (On-line Planner)	13%	19%	6%	17%	2%	14%	0%

Table 1: Comparison of our on-line planner to a quasi-optimal oracle

Comparison to quasi-optimal oracle: Ideally, the planner's efficiency should be evaluated relative to an instantaneous oracle always making the best decision. However, building such an oracle is not realistic, since it requires implementing an optimal off-line manipulation planner. Instead, we built a quasi-optimal oracle as follows: We let each of the two arms move at maximal velocity along a simple path connecting two configurations, one where the gripper is above the belt, the other where it is above the table away from the belt. The two arms perform these motions alternately, forward and backward, so that when one arm is above the belt the other arm is at the other end of its trajectory above the table. The trajectories are defined so that no collision occurs in the middle. Then we define a feeding sequence of parts so that when an arm reaches the end of its trajectory above the belt, a part is right there to be grasped and the goal of this part is exactly at the other end of the arm's trajectory. Finally, we distribute the obstacles on the table so that no part collides with an obstacle when it is moved by an arm. By construction, the missing ratio of this oracle for the sequence of parts defined above is 0%.

We ran the planner with the same obstacle distribution and the same sequence of parts. Table 1 compares results obtained with the oracle and the planner. In column 1, we feed the part with no uncertainty. The number of parts fed per minute is 13.19. The missing ratio of the oracle is 0%, while the missing ratio of the planner is 13%.

In columns 2 and 3 of the table, instead of feeding parts at exactly the times computed above, we let them arrive within a ± 0.5sec uncertainty interval. If the feeding rate is unchanged (column 2), the missing ratio of the oracle increases sharply to 20% (this is obtained by temporarily stopping the arms' motions whenever an arm reaches the belt prior to the arrival of the part); on the other hand, the missing ratio of the planner increases slightly to 19%. Let us slow down the feeding rate just enough so that the oracle's missing ratio becomes zero again (column 3); this requires stopping the arm motions given by the oracle, for a maximum of 1sec prior to any grasping operations. The belt now feeds 11.88 parts/min. The missing ratio of the planner is also reduced to 6%. The subsequent columns show similar results when feeding uncertainty is ± 0.75sec and ± 1sec. When the feeding rate is slowed down just enough to make the oracle's missing ratio equal to 0%, the planner's missing ratio drops to 2% and 0%, respectively. Note the stability of planner's performance throughout these experiments.

We have run several experiments similar to the above. In all cases, our planner showed the same high degree of competitiveness. Combined with the fact that it also allows for dynamic changes in the parts and the obstacles, these results suggest that in many situations an on-line planner such as ours can be more attractive than an excellent off-line planner.

CPU speed factor	S_1	S_2	S_3	S_4	S_5	average
0.1	34	33	29	28	35	31.8
0.2	26	30	23	22	27	25.6
0.5	18	15	15	13	12	14.6
1.0	9	10	9	13	8	9.8
2.0	6	6	10	9	10	8.2

Table 2: Effect of planning time on missing ratio

belt speed	S_1	S_2	S_3	S_4	S_5	average
0.5	15	10	20	20	13	15.6
0.75	12	5	9	15	11	10.4
1.0	9	11	9	13	8	10.0
1.25	7	8	8	13	12	9.6
1.5	10	11	7	11	11	10.0
2.0	6	11	12	11	12	14.4
2.5	19	16	16	20	16	17.4
3.0	20	21	20	25	19	21.0

Table 3: Effect of belt velocity on missing ratio

Effect of planning time: We also analyzed the effect of planning time by artificially changing the planner's running speed. This is done as follows: Whenever the planner solves for a grasp or deliver subtask, we interrupt the simulator and measures the planner's running time. When the computation is over, we let the simulator update the state of the environment according to the running time of the planner. For this update, we can set the running time as we desire, e.g., to twice what it actually was, or half of it, or even zero. The variations of the missing ratios for different planning speeds gives us an idea of the planner's efficiency if we used a slower or faster computer.

Table 2 gives the missing ratios of the planner for five feeding sequences $S_1, \ldots, S_5$ and five planner's speeds (0.1, 0.2, 0.5, 1.0, and 2.0 times its nominal speed). When the planner's speed is half the nominal one, the missing ratio is still reasonably small. When the speed is twice the nominal one, the planner's performance is not greatly improved. When the planner's speed is even greater (not shown in the table), planning time becomes negligible relative to execution time, and the missing ratios remain approximately constant.

These results suggest that, as computers become faster, it will be worth making the planner devote more computation than it currently does generating motion plans that are quicker to execute, e.g., by reducing the number of hand-over operations and the number of changes in arm posture, and by increasing parallelism between motions.

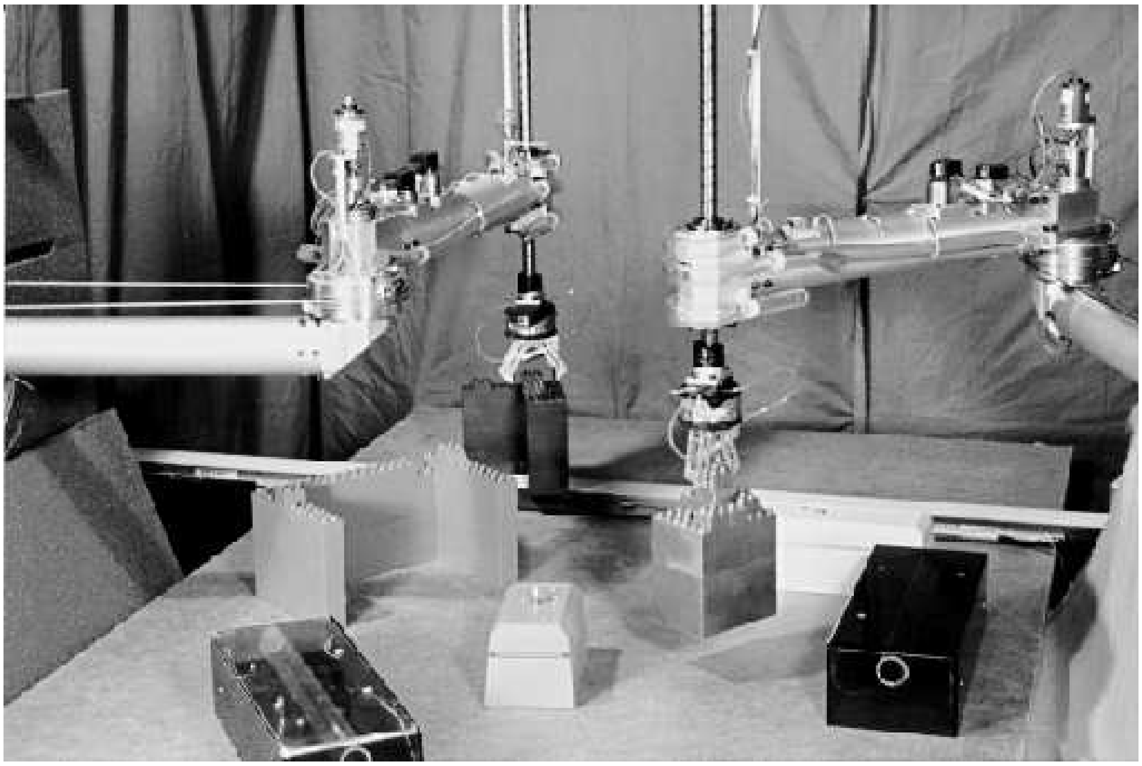

Figure 6: Dual-arm robotic system and experimental setup

Effect of belt velocity: Table 3 shows the planner's missing ratios for five different runs and eight different velocities of the belt (0.5, 0.75, 1.0, 1.25, 1.5, 2.0, 2.5, and 3.0 times the nominal velocity). The feeding rate is fixed, so that slowing down the belt results in more parts on the belt at any one time. In general, the table indicates that the planner's performance is rather insensitive to the belt speed. However, above some speed (about twice the nominal speed), it degrades more rapidly. When the belt is slowed down, one could expect an increase in performance, since parts stay longer on the belt and so can be grasped over larger intervals of time. Surprisingly, the contrary happens. This seems to be caused by the presence of more parts on the belt at the same time, leading the arms to constrain each other more severely than when the speed is higher. This suggests that our heuristics to assign parts to arms should be improved in this case; e.g., we may too frequently assign ARM1 to an arriving part whose goal is not reachable by ARM1 or hard to reach, while we could let this part advance on the belt and then assign it to ARM2. This kind of improvement could benefit the planner even when the belt's velocity is nominal.

9 Connection to Robotic System

System description: We have connected our planner with the dual-arm robotic system shown in Figure 6, which has been developed in the Aerospace Robotics Laboratory at Stanford [31]. The integrated system comprises five major modules: the user interface, the on-line manipulation planner, the dual-arm robot control system, the real-time vision system, and the graphic simula-

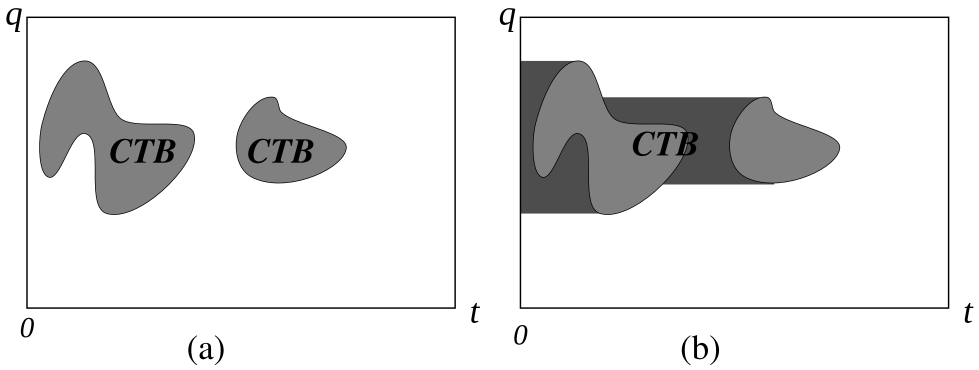

Figure 7: Extending the forbidden region in CT-space

tor [28, 29]. Characteristic arrangements of LEDs are mounted on all objects of interest (arriving parts and obstacles); the overhead vision system senses these LEDs, identifies their arrangements, and computes the positions of the objects in real time. This information leads to updating a model of the environment that is used by all other modules. For example, the planner learns that a new part arrives on the belt or that the position of an obstacle has changed by periodically accessing this model. The user interface provides commands to interactively specify and modify the goals of the parts arriving on the belt. The graphic simulator allows the user to observe graphic renderings of what the vision and the control modules believe is going on in the real world and what the planner predicts will happen soon.

This software is integrated under ControlShell, which provides object-oriented tools for combining software components developed separately [1]. The five modules are implemented on several computers and communicate through a subscription-based network data sharing system called the Network Data Delivery Service (NDDS) [30]. In the current implementation, the user interface and the planner run on two different UNIX workstations, while the control and vision modules run on several VME-based real-time processors.

We have successfully experimented with this integrated system on various examples similar to the one shown in Fig. 2-4, as well as on examples involving long objects requiring two arms for their transfer.

Interfacing the planner with the rest of the system: The planner assumes that the arm joints can change velocity instantaneously. Planned trajectories are thus impossible to execute accurately. Hence, after a trajectory has been passed by the planner to the controller, the latter recomputes its time parameterization using a realistic dynamic model of the arms. The new trajectory has the same geometry as the one produced by the planner.

Let us assume for a moment that the arms can exactly execute the recomputed trajectories. In some cases, to guarantee that the trajectories recomputed by the controller are still collision-free, the planner takes a more conservative planning approach than the one previously described. For example, consider a grasp trajectory of A_1 to be performed while A_2 is moving. The planner maps A_2's trajectory to a forbidden region in CT_1 and extends this region by its shadow along the negative time dimension, as illustrated in Fig. 7. (Note that the previous generation of A_2's motion accounted for the current configuration of A_1; therefore, this configuration lies outside the extended forbidden region.) Thus, the planned trajectory of A_1 can be arbitrarily translated toward the right (greater values of time) without causing any collision. It is then sufficient for the controller, when it recomputes A_1's trajectory, to make sure that A_1 is never ahead of time relative to A_2.

With slight modifications as the above, recomputed motions remain collision-free. But an arm may now fail to arrive in time to grab a part on the belt. This problem is handled by setting the maximal joint velocities in the planner smaller than the actual values. Some tuning is necessary: If the velocity values selected for the planner are too small, the planner will often fail to find grasp paths; but if the values are too large, the actual motions will often arrive too late to grasp the parts. On our implementation, the velocity bounds given to the planner are constants that have been estimated through preliminary experiments. Using these bounds, it may happen (though rarely) that an arm arrives too late to grasp a part. In that case, the planner may decide to make another try by generating another motion, possibly with the other arm.

In our network-distributed implementation, the planning and control modules run on different machines. Communication delays are not negligible. We model them by a duration proportional to the length of the transmitted trajectory, plus some small latency. The planner adds this delay to the computed start time of a grasp trajectory to determine if the robot reaches the grasp configuration on time.

We assumed above that the arms perfectly track the recomputed trajectories. However, control errors cannot be totally avoided. In our implementation we bounds the angular errors of the joint angles of an arm by two constants η_1 and η_2. These bounds specify that, if the arm is expected to be at configuration q at time t, it may actually be anywhere in the parallelepiped centered at q whose sides have lengths η_1 and η_2. Every configuration along a trajectory of A_1 is mapped into CT_2 by considering the region swept by A_1 between the two extreme configurations it may achieve. Appropriate link-link bitmaps are precomputed, so that this computation brings no additional cost.

Vision sensing is also imperfect. Errors in the positions of the objects on the table measured by the vision system have been bounded by preliminary experiments. These bounds are used to grow the geometric models of the objects used by the planner. Errors on the grasp position of a part are also bounded in order to generate safe C-space bitmaps for the parts.

Finally, we must consider the grasping operations on the belt. Taking conservative approaches as above would not allow an arm to reliably grasp a moving part. Hence, we proceed differently: When a gripper arrives within some distance to the part it is expected to grasp, the controller does not try to track further the planned trajectory. Instead, it tracks the part using the last position given by the vision sensor (when the gripper is almost above the part, this sensor no longer sees the part) and the measured velocity of the conveyor belt (which makes it possible to infer the motion of the part). When the gripper is above the part, both moving at the same velocity, the controller commands the grasp operation. While in this autonomous mode, the controller checks for collision between the arms. If one is going to happen, it stops both arms. It also reports the failure to the planner, which may try to generate new trajectories to catch the parts that have been missed. During a grasp operation, the vision sensor keeps updating the environment model. By accessing this model, the planner is informed of the new configurations taken by the grasping arm.

η	$\mu = -0.5$	$\mu = -0.25$	$\mu = 0.0$	$\mu = 0.25$	$\mu = 0.5$
0.0	12	13	9	17	29
0.25	11	10	10	16	31
0.5	13	11	14	18	23
average	12	11.33	11	17	27.67

Table 4: Effect of belt velocity on the missing ratio of different part sequences

Evaluation of planner: The way we deal with discrepancies between the world model used in planning and the real world affects the efficiency of the total system. Thus, the following question arise: Could it be preferable to use a more realistic model of the real world at planning time? Using such a model would certainly increase planning times, but on the other hand it would also reduce delays in the interface between the planner and the robot controller. We do not have a definite answer to this question, but we have conducted experiments that shed some light on it.

Our experiments consisted of running the planner connected to a modified version of the simulator used in the previous section. Consider a motion generated by the planner. Let δ be the duration of the motion according to the planner's model. The modified simulator executes this motion in a time randomly selected in the interval $[\delta \times (1.0 + \mu - \eta), \delta \times (1.0 + \mu + \eta)]$, where μ and η are two parameters. Table 4 gives the planner's missing ratio for the first feeding sequence used in Table 2, for several values of μ and η. The third entry of the first row in Table 4 is the missing ratio (9%) when there are no discrepancy between the planner and simulator models.

Note that the missing ratio increases more rapidly when $\mu > 0$ (i.e., when the planner over-estimates the performance of the arms) than when $\mu < 0$ (the planner under-estimates the performance of the arms). When the planner over-estimates performance, the motions arrive late to grasp the parts; this requires the arms to spend more time tracking the parts on the belt; the chances to miss parts also increase. When the planner is conservative, it may fail to generate grasp motions that would actually be feasible; but whenever a grasp motion is found, this motion is usually executed with success; moreover, less time is wasted tracking the parts to be grasped. Table 4 also shows that the planner is rather insensitive to the variations of η. By comparing Table 4 and the first column of Table 2, we notice that the effect of $\mu = 0.25$ has about the same magnitude as doubling planning time, while setting $\mu = 0.5$ has about the same effect as increasing planning time by a factor of 4.

These results combined with those of Section 8 (Effect of planning time) suggest that future increase in computer speed could usefully be exploited by making use of a more realistic arm model at planning time.

10 Conclusion

This paper has described an on-line manipulation planner for a dual-arm robot system whose task is to grab parts arriving on a conveyor belt and deliver them at specified goals. Parts arrive at any time, in random order. The planner uses information provided by a vision system to break the overall planning problem into a stream of rather simple subproblems and orchestrate fast planning primitives solving these subproblems. Experiments conducted with this planner in a simulated robot environment show that it compares very well to quasi-optimal oracles. Experiments with a real dual-arm robot system have demonstrated the viability of on-line planning in the real world. Since the planner also allows dynamic changes in obstacles, goals, and tasks, this result suggests that on-line planning may rapidly become more attractive than off-line planning (whose efficiency is also very sensitive to feeding accuracy). In fact, we believe that our on-line planner enables low-cost, flexible, and efficient part feeding.

Evaluation of the planner shows that, as computers become faster, future research should focus on spending more planning computation to produce motion plans that are quicker to execute (e.g., avoiding hand-over operations and changes in arm posture) and on using more realistic arm models to reduce delays in the planner/controller interface. Additional research could also be done to keep the planner always busy, by making it anticipate future motions whenever there is time available for that. Other interesting research topics include dealing with more than two arms and with more than one computing resource (possibly shared with other activities, like sensing and control).

Acknowledgments: This research was funded by ARPA grant N00014-92-J-1809. The authors thank Yoshihito Koga and Gerardo Pardo-Castellote for their constructive comments. They also gratefully acknowledge the use of the dual-arm system made available to them by Prof. R.H. Cannon in the Aerospace Robotics Laboratory. The experiments with this system were conducted with Gerardo Pardo-Castellote and Stan Schneider.

References

[1] ——, *ControlShell: Object Oriented Framework for Real-Time Software – User's Manual*, Real-Time Innovations Inc., Sunnyvale, CA, 4.2 edition, August 1993.

[2] R. Alami, T. Siméon, and J.P. Laumond, A Geometrical Approach to Planning Manipulation Tasks: The Case of Discrete Placements and Grasps, *Robotics Research 5*, MIT Press, Cambridge, MA, 1990, 453-459.

[3] J. Barraquand and J.C. Latombe, Robot Motion Planning: A Distributed Representation Approach, *Int. J. of Rob. Res.*, 10(6), Dec. 1991, 628-649.

[4] J. Barraquand, B. Langlois, and J.C. Latombe, Numerical Potential Field Techniques for Robot Path Planning," *IEEE Tr. on Sys., Man, and Cyb.*, 22(2), 1992, 224–241.

[5] M.S. Branicky and W.S. Newman, Rapid Computation of Configuration Space Obstacles, *Proc. IEEE Int. Conf. Rob. and Aut.*, 304-310, 1990.

[6] H.S. Chang and T.Y. Li, Assembly Maintainability Study with Motion Planning, Manuscript, 1994.

[7] J.J. Craig, *Introduction to Robotics. Mechanics and Control*, Addison-Wesley, Reading, MA, 1986.

[8] M. Erdmann and T. Lozano-Pérez, On Multiple Moving Objects, *Algorithmica*, 2(4), 1987, 477-521.

[9] B. Faverjon and P. Tournassoud, A Practical Approach to Motion-Planning for Manipulators with Many Degrees of Freedom, *Robotics Research 5*, MIT Press, Cambridge, MA, 1990, 425-433.

[10] K. Fujimura, *Motion Planning in Dynamic Environments*, Springer-Verlag, New York, NY, 1991.

[11] K. Fujimura, Motion Planning Amid Transient Obstacles, *Int. J. of Rob. Res.*, 13(5), 1994, 395-407.

[12] L. Graux, P. Millies, P.L. Kociemba, and B. Langlois, Integration of a Path Generation Algorithm into Off-Line Programming of AIRBUS Panels, *Aerospace Automated Fastening Conf. and Exp.*, SAE Tech. Paper 922404, Oct. 1992.

[13] L. Guibas, L. Ramshaw, and J. Stolfi, A Kinetic Framework for Computational Geometry, *Proc. FOCS*, 1983, 100-111.

[14] K.K. Gupta and Z. Guo, Motion Planning for Many Degrees of Freedom: Sequential Search with Backtracking, to appear in *IEEE Tr. on Rob. and Aut.*.

[15] K.K. Gupta and X. Zhu, Practical Motion Planning for Many Degrees of Freedom: A Novel Approach Within Sequential Framework, *Proc. IEEE Int. Conf. Rob. and Aut.*, San Diego, CA, 1994, 2038-2043.

[16] K.K. Gupta and S.W Zucker, Toward Efficient Trajectory Planning: Path Velocity Decomposition, *Int. J. of Rob. Res.*, 5, 1986, 72-89.

[17] L. Kavraki and J.C. Latombe, Randomized Preprocessing of Configuration Space for Fast Path Planning, *Proc. IEEE Int. Conf. Rob. and Aut.*, San Diego, CA, 1994, 2138-2145.

[18] Y. Koga, K. Kondo, J. Kuffner, and J.C. Latombe, Planning Motions with Intentions, *Proc. SIGGRAPH'94*, 1994, 395-408.

[19] Y. Koga, T. Lastennet, T.Y. Li, and J.C. Latombe, Multi-Arm Manipulation Planning, *9th Int. Symp. on Automation and Robotics in Construction*, Tokyo, 1992, 281-288.

[20] Y. Koga and J.C. Latombe, Experiments in Dual-Arm Manipulation Planning, *Proc. IEEE Int. Conf. Rob. and Aut.*, Nice, France, 1992, 2238-2245.

[21] Y. Koga and J.C. Latombe, On Multi-Arm Manipulation Planning, *Proc. IEEE Int. Conf. Rob. and Aut.*, San Diego, CA, 1994, 945-952.

[22] J.C. Latombe, *Robot Motion Planning*, Kluwer, Boston, MA, 1991.

[23] J.C. Latombe, A Fast Path Planner for a Car-Like Indoor Mobile Robot, *Proc. 9th Nat. Conf. Artif. Int.*, AAAI, Anaheim, CA, 1991, 659-665.

[24] J. Lengyel, M. Reichert, B.R. Donald, and P. Greenberg, Real-Time Robot Motion Planning Using Rasterizing Computer Graphics Hardware, *Proc. SIGGRAPH'90*, Dallas, TX, 1990.

[25] T. Lozano-Pérez, Spatial Planning: A Configuration Space Approach, *IEEE Tr. on Comp.*, 32(2), 1983, 108-120.

[26] T. Lozano-Pérez, Parallel Robot Motion Planning, *Proc. IEEE Int. Conf. Rob. and Aut.*, Sacramento, CA, 1991, 1000-1007.

[27] A.A. Maciejewski and J.J. Fox, Path Planning and the Topology of Configuration Space, *IEEE Tr. on Rob. and Aut.*, 9(4), 1993, 444–456.

[28] G. Pardo-Castellote, *Experimental Integration of Planning and Control in a Distribited Robotic System*, PhD thesis, Dept. Electrical Eng., Stanford Univ., CA, 1994.

[29] G. Pardo-Castellote, T.Y. Li, Y. Koga, R.H. Cannon, J.C. Latombe, and S.A. Schneider, Experimental Integration of Planning in a Distributed Control System, *Proc. Int. Symp. on Experimental Rob.*, Kyoto, Oct. 1993, 217-222.

[30] G. Pardo-Castellote and S. Schneider, The Network Data Delivery Service: Real-Time Data Connectivity for Distributed Control Applications, *Proc. IEEE Int. Conf. Rob. and Aut.*, San Diego, 1994, 2870-2876.

[31] L.E. Pfeffer, *The Design and Control of a Two-Armed, Cooperating, Flexible-Drivetrain Robot System*, PhD thesis, Stanford Univ., Stanford, CA, 1993.

[32] J.H. Reif and M. Sharir, Motion Planning in the Presence of Moving Obstacles, *Proc. FOCS*, 1985, 144-154.

[33] C. Shih, T. Lee, and W.A. Gruver, A Unified Approach for Robot Motion Planning With Moving Polyhedral Obstacles, *IEEE Tr. Sys., Man, and Cyb.*, 20, 1990, 903-315.

[34] P. Tournassoud, T. Lozano-Pérez, and E. Mazer, Regrasping, *Proc. IEEE Int. Conf. Rob. and Aut.*, Raleigh, NC, 1987, 1924-1928.

[35] G. Wilfong, Motion Planning in the Presence of Movable Obstacles, *Proc. ACM Symp. on Comp. Geometry*, 1988, 279-288.