Planning by Search Through Simulations

David Miller

YALEU/CSD/RR #423

October 1985

This work was supported by the Advanced Research Projects Agency of the Department of Defense and monitored under the Office of Naval Research under contract N00014-85-K-0301.

Planning by
Search Through Simulations

A Dissertation
Presented to the Faculty of the Graduate School
of
Yale University
in Candidacy for the Degree of
Doctor of Philosophy

by
David Paul Miller
December 1985

Abstract

Planning by Search Through Simulations

David Paul Miller

Yale University
1985

This research is directed towards the creation of automatic planning systems capable of directing the actions of robots in solving realistic tasks. Realistic domains often involve constraints on quantitative issues such as time and cost. In order to produce successful plans, interactions and conflicts between plan steps that involve these quantities must be detected and resolved. The central assumption of this work is that the ordering of tasks, and the sequencing of the plan steps created for their solution, are major and necessary parts of the planning process. The order in which a plan's actions are to be carried out can greatly affect the efficiency and outcome of executing the plan. The actual design of the plan can and should be affected by the ordering of its steps.

Least-commitment robot planners such as Noah produce final plans that are in the form of a partially ordered set of steps for the robot to carry out. This research shows that plans generated by least-commitment planners are inadequate in many domains because the partial order can hide the quantitative interactions between the plan steps. In particular, plan interactions that involve looping plans, efficiency concerns, deadlines, and the allocation of continuous and renewable resources are beyond the scope of least-commitment planners.

This thesis presents a new planning paradigm in which a more detailed analysis is performed on the effects of certain ordering decisions. Some of the total orderings of the plan steps are constructed in order to detect interactions invisible in partial orders. The construction of total orders allows the production of plans that include steps which are repeated indefinitely. This allows the the automated planning of problems in domains that were beyond the abilities of previous problem solving systems. Problems in the domain of real-time mobile-robot navigation are used to demonstrate the expanded abilities of this planning system.

©Copyright by David Paul Miller, 1985
ALL RIGHTS RESERVED

Acknowledgements

There are many people who have made my stay here at Yale an enjoyable and productive one; I would like to take this opportunity to thank some of them. First, I would like to thank my adviser and friend, Drew McDermott. Drew provided crucial comments and suggestions on both the implementation and write-up of this work. He, along with Judi and Noel McDermott, provided me with much needed enthusiastic distraction during those occasional days of thesis depression.

I would also like to thank the other members of my reading committee: Christopher Riesbeck and Dan Koditchek. Both of them spent significant amounts of their time discussing with me the various aspects of my work. I greatly appreciate their comments, insights, and fast turn-around time on earlier drafts of this dissertation.

I first got excited about AI in courses I took from Roger Schank. He, along with Elliot Soloway, rounded out the AI faculty at Yale, and helped to provide a very stimulating environment in which to work.

The other members of the Forbin Project, Tom Dean and Jim Firby, were a great help to me in the completion of my dissertation. Both helped to proof this thesis, and both took part in extended discussions on planning which helped me to solidify many of my ideas. Additionally, Jim wrote the Draw program which I used to create the many figures in this document. I am very grateful for their help.

Yoav Shoham, Brad Alpert, Steve Hanks, Jim Spohrer, Alex Kass, and Charles Martin read earlier drafts of parts of this dissertation and provided many useful comments. They along with Kris Hammond, Gregg Collins, Stan Letovsky, David Atkinson, Ernie Davis, Dan Gusfield, and Dana Angluin have given me countless hours of fruitful discussion. Many of the ideas in this thesis had their origin in these talks.

A computer science dissertation is difficult to create without access to computers and printers. Whenever I had a problem with any of the machines I was always able to count on David Teodosio, John Philbin, Rob Carey, Bob Parker, and Richard Guillemette. Their quick and helpful responses eliminated much of the frustration of dealing with itinerant machines.

Some of the other people in the department who have made my time at Yale more interesting and enjoyable include: Norm Adams, Beth Adelson, Bill Bain, Doug Baldwin, Larry Birnbaum, Mark Burstein, Natalie Dehn, Denys Duchier, Margot Flowers, Andrew Gelsey, Eric Gold, Abraham Gutman, Eduard Hovy, Larry Hunter, Lewis Johnson, Richard Kelsey, David Leake, Kai Li, David Littleboy, Steve Lytinen, Chris Owens, Jim Philbin, Colleen Seifert, Steven Slade, and Eric Urdang. I would especially like to thank Chris Owens and Ken Olsen for providing me with a comfortable place to stay when I needed it.

Finally, I would like to thank my wife, Cathryne. Being married to a graduate student is not always an easy task, but she has carried it out admirably. Her enthusiasm and love have made my task of creating this dissertation much easier.

Contents

1 Introduction **1**

1.1 Planning: A Historical Perspective 1

1.1.1 Situation-Based Planners 1

1.1.2 Least-Commitment Planners 5

1.2 The Planning Problem 9

1.2.1 Deadlines and Travel Time 10

1.2.2 Resource Allocation 13

1.2.3 Domains Containing Loops 15

1.2.4 Mobile Robot Feedback Tasks 16

1.3 Planning: A New Paradigm 21

1.3.1 Exploring Total Orderings 22

1.3.2 Solving Mobile Robot Problems 26

1.4 Overview of This Dissertation 30

1.5 Review of Related Works 31

1.5.1 Planning Systems 31

1.5.2 Temporal Relationships and Representations 35

1.5.3 The Scheduling Problem 37

1.5.4 AI Planners That Do Scheduling 41

2 Space, Time, and Representation: What a Robot Needs to Know **45**

2.1 Introduction 45

2.2 The Mobile Robot 46

2.2.1 The Robot's Effectors and Sources of Error 47

2.2.2 The Ultrasonic Sensor 48

2.3 The Spatial Representation 50

2.4 Robot Positioning 54

2.5 The Route Planner 57

2.5.1 Finding the Easiest Route 57

2.6 Spatial Information and Planning 60

2.6.1 Choosing an Observation Strategy 61

2.7 The Task Specification Language 66

2.7.1 Introduction 66

2.7.2 The Task Specification Index and Name 67

2.7.3 Parameters and Declared Variables 68

2.7.4 Monitor Functions 69

2.7.5 Resource Specification 69

2.7.6 Declaration of Scheduling Relations 70

2.7.7 The Plan Schema 72

2.7.8 Specifying Constraints 74

2.7.9 The Utility Function 78

2.7.10 Example Task Specification 79

2.8 How TSL Compares to Other Planning Languages 82

3 Resources, Physics, and Scheduling **85**

3.1 Specifying the Physics 85

3.1.1 The Syntax of Resource Declarations 85

3.1.2 Scheduler Variables 87
3.2 Resources and State Changes 88
3.2.1 State Delays 91
3.2.2 State Restrictions 94

4 The Scheduling Algorithm 101

4.1 Introduction 101
4.2 The Scheduling Process 103
4.2.1 The Searcher 104
4.2.2 The Schedule Dependency Array 106
4.2.3 Determining if a Feasible Schedule Can be found 109
4.3 Task Orderings and Prerequisites 110
4.4 Execution Windows 112
4.5 Current Status Objects 114
4.6 Scheduling Windows 116
4.7 Schedule Rating Heuristics 117
4.7.1 Rating the Schedule So Far 117
4.7.2 Rating Possible Prefix Expansions 118
4.7.3 Final Scoring, Re-Scoring, and Critics 120

5 Scheduling Loops 125

5.1 Introduction 125
5.2 Scheduling Repeated Tasks 127
5.3 Condensing Loops 134
5.4 Loop Termination 141

6 Experiments 145

6.1 Scheduler Implementation 145

6.2 Randomly Generated Scheduling Tests 147

6.3 Additional Scheduling Tests 153

6.3.1 Unusual State Changes: Vehicles 153

6.3.2 Designing Multi-Agent Plans in the Baseball Domain 155

6.4 Task Selection for the Forbin Planner 157

7 Summary and Conclusions 163

7.1 Summary of Scheduling and Loop Scheduling Algorithms 163

7.2 Limitations and Possible Extensions to Task Scheduling 165

7.2.1 End-Coordinated Planning Problems 165

7.2.2 Scheduling Resource-Intensive Tasks 167

7.2.3 Using the Task Scheduler for Planning 168

7.2.4 Finding Out Why the Scheduler Chooses the Schedule it Does . . . 170

8 References 173

A Example Task-Scheduling Run 179

List of Figures

1.1 Three Stacked Blocks . . . 3
1.2 The Blocks World Problem of Creative Destruction . . . 5
1.3 Part of the Task Network for Planning Car Repair . . . 7
1.4 The Difficulties of Stacking Three Blocks . . . 8
1.5 Workstations to be Visited in a Limited Time . . . 12
1.6 Planning a Double-Play . . . 14
1.7 A Robot Following a Hallway Wall . . . 18
1.8 A Robot Looking in the Wrong Direction at the Wrong Time . . . 19
1.9 A Robot Looking for the Second Left Turn . . . 19
1.10 Plans for Navigating Mobile Robot . . . 20
1.11 Plan Prefixes and Total Orderings . . . 23
1.12 The Architecture of the Bumpers System . . . 27
1.13 A Robot Looking for the Second Left Turn . . . 28
1.14 Plan for Navigating Robot to Second Left . . . 29

2.1 Sonar Dispersion & Definition of D_{max} . . . 49
2.2 Why Objects in the Map Must be at least of Height H . . . 50
2.3 The Importance of the Outer Hull Being at the Sensor Height . . . 51
2.4 Dividing a Map into Regions . . . 53
2.5 Typing the Regions in a Map . . . 54

2.6 The Tree of Observations and Edge Matches 55
2.7 Voronoi Diagram of the Example Map 56
2.8 A Quick Check of Orientation and Distance to the Nearest Wall 58
2.9 Navigating Across a Gymnasium with a Table in the Center 59
2.10 V-Regions, The Combined Voronoi Diagram and Regionalized Map 60
2.11 The Easiest to Navigate Route 61
2.12 Sensor Direction for a Robot Following a Wall 63
2.13 Sighting a Landmark Off to the Left 64
2.14 Tracking a Landmark to Follow an Arbitrary Course 65
2.15 Task Specification for Stopping Near a Wall 67
2.16 Function for Calculating Maximum Lateral Drift 71
2.17 The Syntax of the Prog Statement 72
2.18 The Syntax of the Repeat Statement 73
2.19 Using the MAKE-EVAPORATE Statement 77
2.20 Constraints for Executing a Parachute Jump 78
2.21 Plan Schema for Hammering a Nail and Reading a Book 78
2.22 Map for the Example Task 81
2.23 The Complete Specification for Traversing a 2-F Region 84

3.1 The WITH-RELATIONS Statement for Calculating Lateral Drift 86
3.2 Syntax of the Physics Declaration 86
3.3 Resource Declaration for the Robot's Position 89
3.4 The Fields of a State Object 90
3.5 Transition Function Side Effects 92
3.6 Three Rooms to be Painted 96
3.7 The Syntax of the Restriction Statement 96
3.8 Many Paths; Few that are Acceptable 98

3.9 A Restriction Against Traveling Through Toxic Waste 98

4.1 A Scheduling Tree for Four Tasks: A,B,C, & D 103

4.2 How Dependencies Are Organized in the SDA 108

4.3 Reducing a Three Task SDA for Schedule Validation 110

4.4 An Itinerary of a Typical Day 113

4.5 Calculating Three RTTs for Three Ready-Tasks 121

4.6 Why RTTs Are Used Instead of Transition Delays 121

5.1 The Widget Painting Factory 126

5.2 The Partial Order for Painting Ten Widgets 127

5.3 The SDA for the Widget Painting Factory 130

5.4 The SDA With the Robot Holding a Can of Paint 131

5.5 The SDA After *Fill Painter* has Been Done Once 132

5.6 SDA When the Painter has All the Paint 133

5.7 The Role of the *Cycle Task* in Unordered Loops 133

5.8 Schedule for Painting Six Widgets 135

5.9 Schedule for Painting Six Widgets 136

5.10 Three Unordered Workstations to be Visited 137

5.11 Two Iterations of Loop1 Containing Fragments of Loops 2 & 3 138

5.12 Loop Extraction Triggered by Cycle-Task of Loop1 139

5.13 Map of Corridors for Mobile Robot to Navigate 143

6.1 The Task Scheduler's Heuristic Rating Function 147

6.2 A Typical Randomly Generated Workstation Problem 149

6.3 Problem Size vs Scheduling-Tree Nodes Searched 152

6.4 Distance Map for Car Repair Problem 154

6.5 Flow of Control in the FORBIN System 159

6.6 Layout of the Forbin Factory 160

7.1 Constraints for Coordinating Dinner 166

List of Tables

3.1 Motor Speeds: Alone and in Conjunction 93

3.2 Arranging Resources for Efficient State Changes 94

4.1 Scoring Changes Due to Scheduling Dead Ends 122

5.1 SO and PO Delays During Loop Extraction 140

6.1 The Size of the Task Scheduler Routines 148

6.2 Initial Values for Rating Factors and Their Decay Rates 150

6.3 Solution to Errand Running Problem Involving Vehicles 155

6.4 Travel Times in the Factory 161

6.5 Forbin's Solution to the Example Problem 162

Chapter 1

Introduction

The research described in this dissertation is directed towards the creation of an automatic planning system. Planning, as the term is used throughout this work, is the process of combining the solutions from several simple problems, having well understood strategies, into solutions for larger, more complex tasks. Many attempts have been made at creating automatic planning systems. Several different strategies have been attempted, each meeting with varied degrees of success. This work presents a new method of planning which attacks a class of problems that has, until this research, been largely ignored by the planning literature.

In order to fully appreciate how this research furthers the cause of automated planning, it is first necessary to know how the planning problem has been attacked in the past. Presented next is a short review of the major planning work that has preceded this research. It is followed by details on the objectives of this dissertation and the methods used to achieve them.

1.1 Planning: A Historical Perspective

1.1.1 Situation-Based Planners

The first major AI planning system was GPS [Ernst 69], otherwise known as *The General Problem Solver.* GPS was a system based on search, and it used ***means-ends analysis*** to help guide the search. Means-end analysis is the search through the space of possible world

situations guided by the difference between the state of the world and the desired state. A problem was presented to GPS as:

- An initial world state
- A goal state
- A function for measuring differences between states
- A set of *operators*
- An *operator-difference* table

The job of GPS was to find a sequence of operators that would transform the world from the initial state of the world to that of the goal state. The system started out by using the difference function to find out what the differences were between the initial state and the goal state. If there were no differences then the problem was solved. If there were differences, GPS would look the differences up in the operator-difference table. In the table, the operators were indexed under the differences for which they were applicable. The indexed operators were then applied to the initial state in order to form a new state of the world. Occasionally operators had preconditions that had to be met before the operators could be applied. A state of the world that had those preconditions satisfied would be set up as a goal state — a state that had to be met before further work on the original goal could continue. Such recursion happened as necessary. Differences between the resulting state and the goal state were measured and reduced. New operators were applied until no differences remained.

The main difficulty with GPS was in putting problems into the search formalism. Difference functions were hard to produce for many problems. Also, problems where the world state could become complex required huge operator difference tables. Since the table and difference function had to be supplied by the user, a lot of work was needed to prepare a problem for GPS.

Strips [Fikes 71], a direct descendent of GPS, was designed to get around these difficulties. Like GPS, Strips' algorithm was based on a search through world situations in order to find a set of operators that would achieve the goal state. Unlike GPS, Strips had one specific way in which to represent world states — a way that made problem preparation and solution easier and more efficient.

Strips represented the world as a conjunction of facts in the first-order predicate calculus. For example, the goal state of the blocksworld problem shown in Figure 1.1 would be stated in Strips as the conjunction of

`(on A B) (on B C) (on C TABLE)`

The goal state was considered achieved when the three facts above were in the data base.

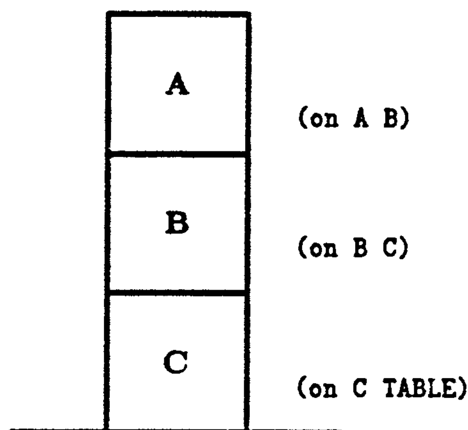


Figure 1.1: Three Stacked Blocks

With this representation of the state of the world the difference function became trivial; facts that were in the statement of the goal state but were not in the "current" state of the world were differences. Operators were standardized so that an operator could have two effects on the state of the world:

1. The operator could add facts to the data base
2. The operator could delete facts from the data base

The facts that the operator added or deleted were contained in its ***add-lists*** and ***delete-lists***. When an operator was applied to a world state the facts in its add-list were put into the data-base while any facts in the data-base that were contained in the delete-list were removed. The resulting data-base of facts captured, as far as Strips was concerned, the complete state of the world.

The use of add-lists made the operator-difference table unnecessary. Operators were indexed by the contents of their add-lists in the Strips system. The add-lists contained all of the differences that an operator could affect.

Preconditions for operators were also set up as predicate-calculus assertions. When an operator was chosen for application, but the current state was missing the preconditions

for that operator, the missing preconditions were set up as differences to be eliminated. Other operators were then applied until the world state reached a point where the original operator could be used.

Essentially, Strips and GPS searched through the space of all possible world states looking for a sequence of operators that would eventually lead to the desired world state. The operators that the system found made up the *plan* for achieving that goal. If the planner's model of the world was correct then the plan it created could be *executed*, in order to actually achieve the goal. Execution, in this sense, is the application, in sequence, of the operators to the world. If each operator successfully performs the actions it is supposed to, and the planner has taken everything into account, then the world should be in a state in which the next operator may be successfully applied.

In such a system, it is possible that applying an operator may add more differences than it removes — undoing the work of previous operators. It is even possible that in some situations an operator can add a difference which no existing operator can remove. Fortunately, planners are not really actors (at least not at the same time). Strips does not really perform the operation associated with an operator; instead it simulates the application of the operators by adding and deleting assertions from the data base. Strips generates and examines each world state. A state where the system has made an error can usually be noticed by the upsurge in differences with the goal state.

However, sometimes the only path to the goal state is one that first moves away from the goal state. Figure 1.2 shows an example from the blocksworld where it is necessary to move away from the goal in order to eventually achieve it. In the blocksworld the robot can only move one block at a time. It is therefore necessary to remove block B from on top of block C, so that C may be placed onto block D. If the system chose to first put block A onto B it would be forced to later undo its work (i.e., plan to remove block A from on top of block B), or to *backtrack* and *replan* (i.e., forget it had ever put A on B).

Backtracking in situation-based planners occurs when either the wrong operator is chosen for the elimination of a difference, or when the differences are eliminated in the wrong order. To backtrack, the system finds a point in its planning process where it had a choice of more than one operator to apply, and this time applies a different operator. Such backtracking can be combinatorically explosive. Backtracking became a major issue in developing planners. To get around the problem it was necessary to develop a completely different idea of what planning was.

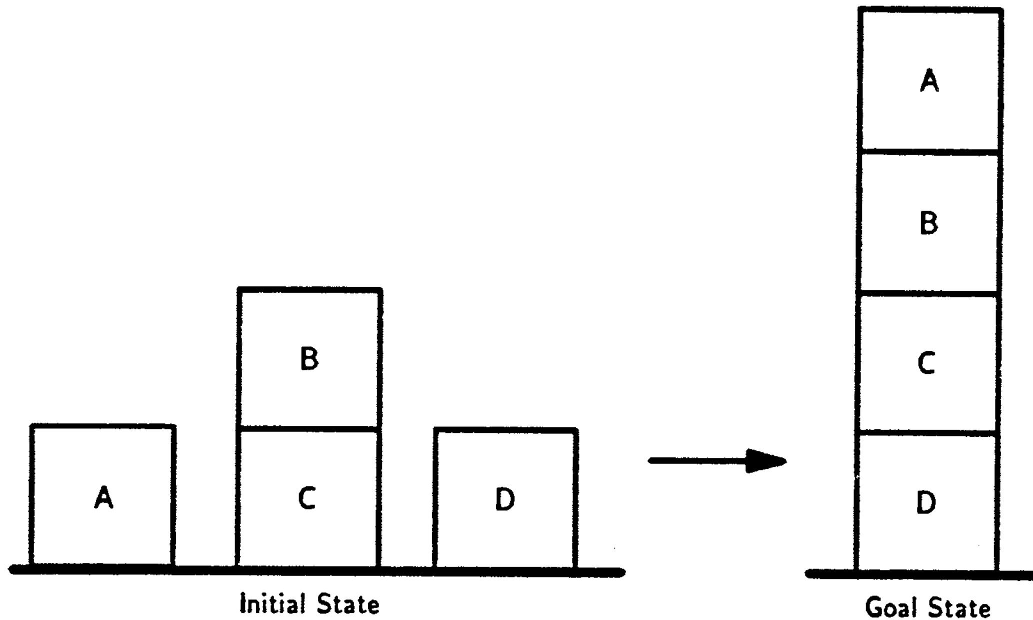


Figure 1.2: The Blocks World Problem of Creative Destruction

1.1.2 Least-Commitment Planners

One cause of backtracking in a Strips-like system is that attempts are made by the system to eliminate relatively unimportant differences at the same time and in the same way as important high-level differences. For example, this would be like deciding exactly what speed you would drive your car at before deciding on a route to take. If you decided on 30mph and then found that the only route was a highway, you would have to backtrack and decide on a different speed, hoping that it will turn out to be more compatible with the rest of your plan.

Sacerdoti tried to solve this problem by arranging the Strips operators and differences into a hierarchy so that the most important aspects of a problem would be decided upon first. His program, Abstrips [Sacerdoti 74], did much less backtracking than Strips on some problems.

Much of the backtracking that remained in Abstrips was caused by the very nature of the search performed by situation-based planners. Systems such as Strips search sequentially through the space of possible situations. This means that they start at one discrete state

of the world, assume some operation and jump to the resulting state of the world, evaluate the situation and assume another operation. Strips has no way of knowing what the next operator to be applied will turn out to be; Strips knows only what operators have been used so far. With this knowledge it is possible for the system to make mistakes and choose an operator that, though it looks good, will not lead to a solution. The search performed by Abstrips is very similar, but the sequences of operators are developed at different levels of abstraction.

In his next system, Noah, Sacerdoti decided to search a different space: the space of partially ordered plans. A plan in Noah's planning paradigm is a partially ordered set of tasks to be accomplished by the system. Since the plan is a preformed group of tasks, it can be assumed that it is internally consistent. In other words, if executed under the standard conditions the plan should work successfully without any need of backtracking. For each task the planner can look in a library of plans and find a plan that is applicable to that task. The plan found for the task contains tasks of its own, referred to as subtasks. The planner searches the library for subplans with which to solve the subtasks. For very low-level tasks the library contains primitive actions — actions directly executable by the system and needing no further *expansion*.

One way this planning paradigm reduces the need for backtracking is that the hierarchical expansion of plans allows a detailed plan to be formed without commiting the steps of the plan to a particular order. The only ordering conditions imposed are those that exist as part of the plan in the plan library.

For example: suppose the planner is given the task fix the car. Using the task as an index, the planner searches through its library of plans for an appropriate solution: take the car to "Ed's Garage," tell him what's wrong. wait. and pay. When a plan is found it must be expanded in order to become usable. Plan expansion is accomplished through the hierarchical decomposition of the plan's subtasks e.g., take the car to "Ed's Garage" is one of the subtasks in our example plan. The plan library is searched for a plan for accomplishing each subtask. The process then continues recursively until a *final plan* is achieved — a plan where every step is a primitive action.

The plan library used by a planner such as Noah is in the form of a *task network*. The task network is a graph where the nodes are tasks, subtasks, and primitive robot actions. There are several types of links that connect the nodes of the task network (see Figure 1.3). Connections that encode the hierarchical structure of the network are called *plan links*. These links go from the top level task to the tops of the various plans for achieving that

task. For example, the link from fix the car to fix the car at "Ed's Garage" is a plan link.

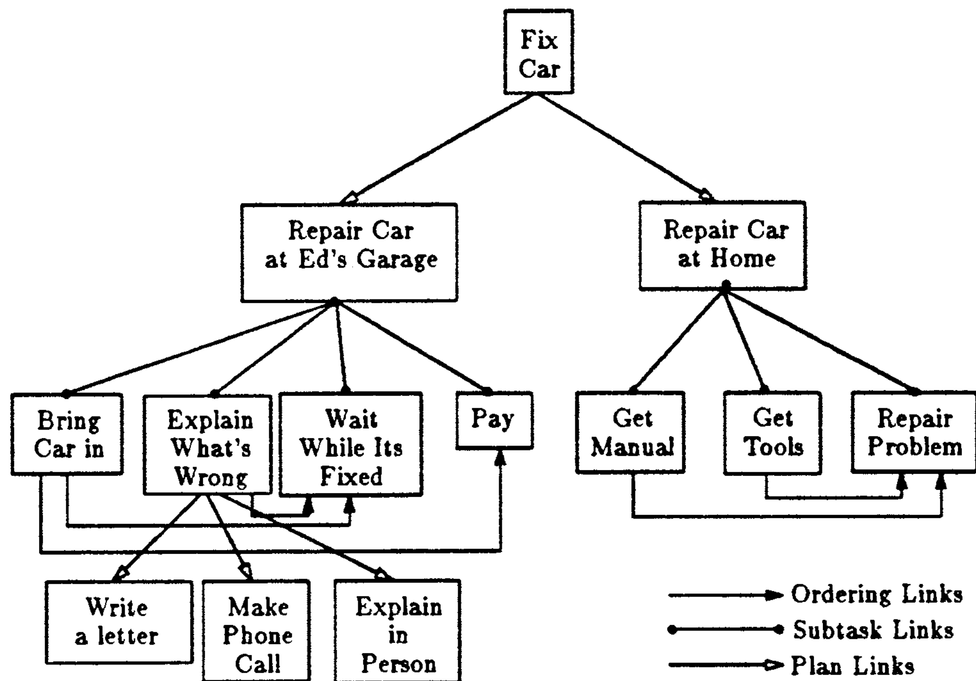


Figure 1.3: Part of the Task Network for Planning Car Repair

Each plan contains a set of subtasks that must be achieved in order for the plan to be accomplished. The subtasks are joined, in the task network, to the plan by a set of ***subtask*** links. When the completion of one subtask in a plan is a prerequisite for another subtask then an *ordering* link is placed into the task network. In our example, it is necessary for the car to be brought to "Ed's Garage" before the subtask of wait for the car to be fixed can be successfully accomplished.

On the other hand, some steps in the plan do not necessarily have to be ordered — Ed could be informed about what was wrong with the car before or after the car is brought in. There are advantages to not ordering the subtasks in a plan where such an ordering is not absolutely necessary. The lack of commitment to particular ordering keeps the plan more flexible; there can be more ways in which to expand the subtasks of the plan.

The phrase *policy of least commitment* is used to summarize this style of planning. It is a policy most closely associated with the Noah planner [Sacerdoti 77], and its immediate

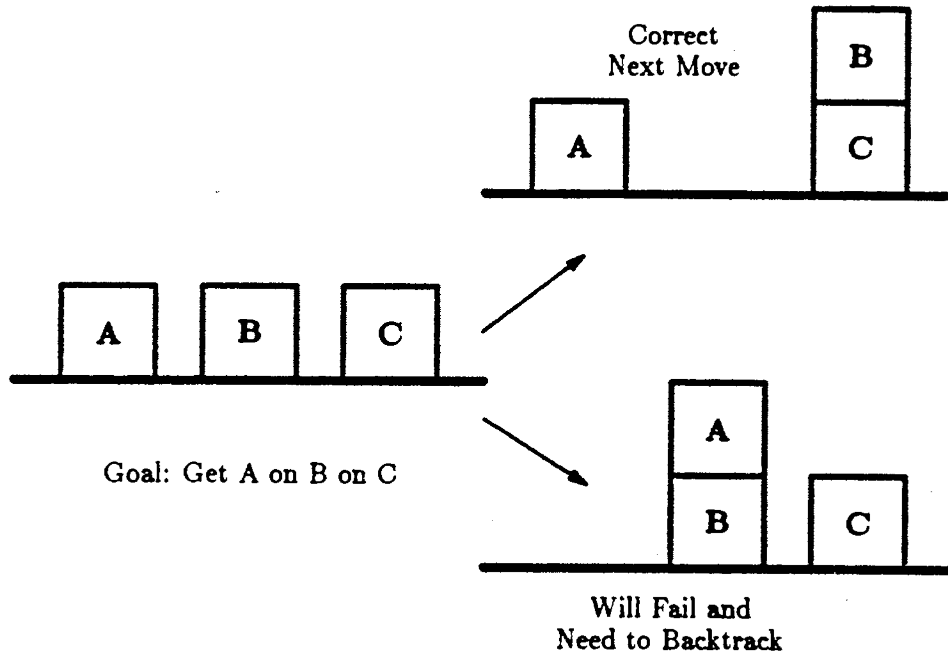


Figure 1.4: The Difficulties of Stacking Three Blocks

successors such as Nonlin [Tate 77]. This policy expresses the belief that by making ordering decisions only when obviously necessary, a planner is more likely to be able to find a workable solution to its given problem. Figure 1.4 shows a situation where an ordering decisions is necessary. There are two possible first moves in attempting to stack three blocks in order. If A is first put on B then the system will be stuck and have to undo its work. The system will realize that it is in trouble because, like Strips, Noah keeps track of the facts that are added and deleted by its actions. Since Noah was designed to avoid backtracking and since it is searching through the space of partially ordered plans, rather than the space of situations, it has more information to go on than Strips. Noah knows some of the actions it is going to do. It stores the facts that are added and deleted by its actions in a Table Of Multiple Effects, or TOME; the TOME can be manipulated to find potential conflicts and alert the planner accordingly. Noah can therefore spot that "A on B" as its first move will fail. It therefore inserts an ordering link forcing the move to put "B on C" to come before stacking A.

Planning, in the least commitment paradigm, is the search through the task network

for a set of non-interfering primitive actions that will allow the achievement of the top-level task. Since the network contains links to guide the search towards the correct primitive actions for achieving any given task, the major difficulty in this form of planning is making sure that those actions do not undo one another. In the domains where least commitment planners have had great success (such as the blocksworld) protection violations can be spotted by noticing where the effects of one task undoes a precondition of another task.

1.2 The Planning Problem

Planning is the process of creating solutions for complex tasks out of the solutions known for simple tasks. Depending on how "simple" the solutions to the simple tasks are, and the exact situation in which they are going to be used, some editing of the plans may be necessary in order to fit all the parts together without conflicts. This editing is a major part of what makes planning a difficult problem.

Situation-based planners create plans by finding solutions for simple problems (i.e., operators) and stringing them together in various ways until a sequence is discovered that solves the problem. The least-commitment planners have taken the opposite approach. They break the complex task into simpler and simple tasks until finally the tasks are so simple that they can be solved by primitive actions.

Least-commitment planners have caused the planning problem to be redefined. In situation-based planners a plan was created from scratch to solve each new problem. Sometimes during the creation of a plan, two or more of the operators would interact badly with one another, e.g., undoing one another's preconditions. These conflicts were solved by Strips-like systems by the exact same means as were used in building the plan in the first place. The method for correcting such a problem was to backtrack to a point before the problem existed, and plan again from there.

In Noah, a problem either has a corresponding plan in the plan library, or it does not. If there is not a plan then there is nothing the system can do. If a plan does exist then the system has to fill in some variable slots and eliminate any bad interactions between the subplans chosen for the subtasks.

For a least-commitment planner the job of planning is detecting and eliminating adverse interactions among subplans. Yet, in this area, current systems have done little more than scratch the surface. Least-commitment planners are really only designed to catch one type of interaction: A might undo B's precondition unless B is done first, also known as Goal

clobbers brother goal [Sussman 75]. In fact, a Noah-like system can only recognize this type of conflict if:

1. Both A and B occur at the same level of hierarchical expansion.
2. A's effect and B's precondition can be expressed as a fact declaratively stated in the system data base.

The first of these problems was solved in part by Tate's Nonlin planner [Tate 76], one of Noah's immediate successors. Nonlin updated the task network with the preconditions and effects of the plan, as each piece of the plan was expanded. When an expansion occurred, all of the preconditions and effects for the task being expanded were passed onto the subtasks in the expansion. The network kept track of conflicts that arose — no matter what level in the hierarchy the conflicting tasks originated from. However, the type of conflicts detectable by Nonlin were limited (as were those detectable by Noah) to one situation: a task's effect undoing another task's precondition. In cases where a simple ordering restriction was insufficient to resolve the conflict, some backtracking was necessary; such situations could arise in domains as simple as the blocksworld.

The second obstacle to conflict recognition is a representational issue. The least-commitment planners grew directly out of the situation-based systems, and most of them share the basic structure of their data base with that used by such systems as Strips. Noah and Nonlin can only recognize task conflicts of the type effect undoing precondition because the preconditions are stated as a conjunction of facts where one or more of the facts is explicitly contradicted by an effect placed in the data base. If A is a precondition then $\neg A$ must appear in the data base for a contradiction to be noticed. This representation is insufficient for recognizing conflicts that appear in domains dealing with time and continuous function — domains that are described in the remainder of this section.

1.2.1 Deadlines and Travel Time

In a problem of the sort get the car fixed. and go see a movie. and eat dinner at Bob's, it is conceivable that a least-commitment planner could find a solution. If the problem statement is further elaborated by: the repair shop closes at 6pm. dinner is served at 6pm. the movie is at 8pm — then the problem goes beyond the abilities of a Noah-like system.

Deadlines turn a problem that formerly could be solved by any "solution" into a problem that requires an efficient solution. Efficient solutions are commonly required in domains

that deal with factors such as time and travel. Processes with temporal constraints do not fit in well with the declarative state information used by Strips-like planners. It is possible to attach some procedural knowledge to Strips (see [Hendrix 73] for an example of this) but this adds new difficulties since the space of possible situations is not finite when continuous processes are allowed. If continuous processes are allowed, it is necessary to select the critical situations (discrete states where operator choices should be made), but this is beyond the abilities of current situation-based systems. Such representation issues do not necessarily eliminate least-commitment systems from these domains. Searching through the space of partially ordered plans is not made any more difficult by the addition of efficiency constraints and continuous processes. However, there are problems that arise in the detection and elimination of adverse plan interactions.

If a task has a deadline associated with it, the plan chosen to solve the task must be efficient enough to execute in the time available. In many cases the order in which the subtasks of a plan are executed can make all the difference between an efficient plan and one that is unworkable. Imagine a situation where a robot must run several errands in a limited amount of time (see Figure 1.5). The amount of traveling that the robot must do for each errand depends on where it is coming from, which is a function of the errand just before. The order in which the errands are performed will affect whether or not they are all completed in time. Yet there is no interacting effect that can be stored in the task network that would allow the planner to realize that an ordering decision must be made. There is no way that a Noah-type planner could even know that a deadline constraint might be violated from just observing a partial order of the errands; the partial order does not provide enough information to calculate the total travel time. Yet the tasks remain only partially ordered because there is no conflict to enforce an ordering decision; an ordering must be first enforced to create a conflict. In such situations a listing of the general requirements and effects associated with a plan does not provide sufficient information to realize that there is an interaction between the tasks, let alone provide the knowledge necessary to deal with the interaction.

The travel task in the errand domain can be thought of as performing a *state transition*. The agent performing the errands must be in the proper state (i.e., location) in order to accomplish each errand. The act of moving from the location of one errand to the location needed for the next is the transition that must be performed. The *state-transition delay* is the amount of time necessary to execute the transition — the amount of time necessary to move the agent from its present location to that location necessary for performing the next errand. But couching an efficiency problem (such as executing a set of errands in limited

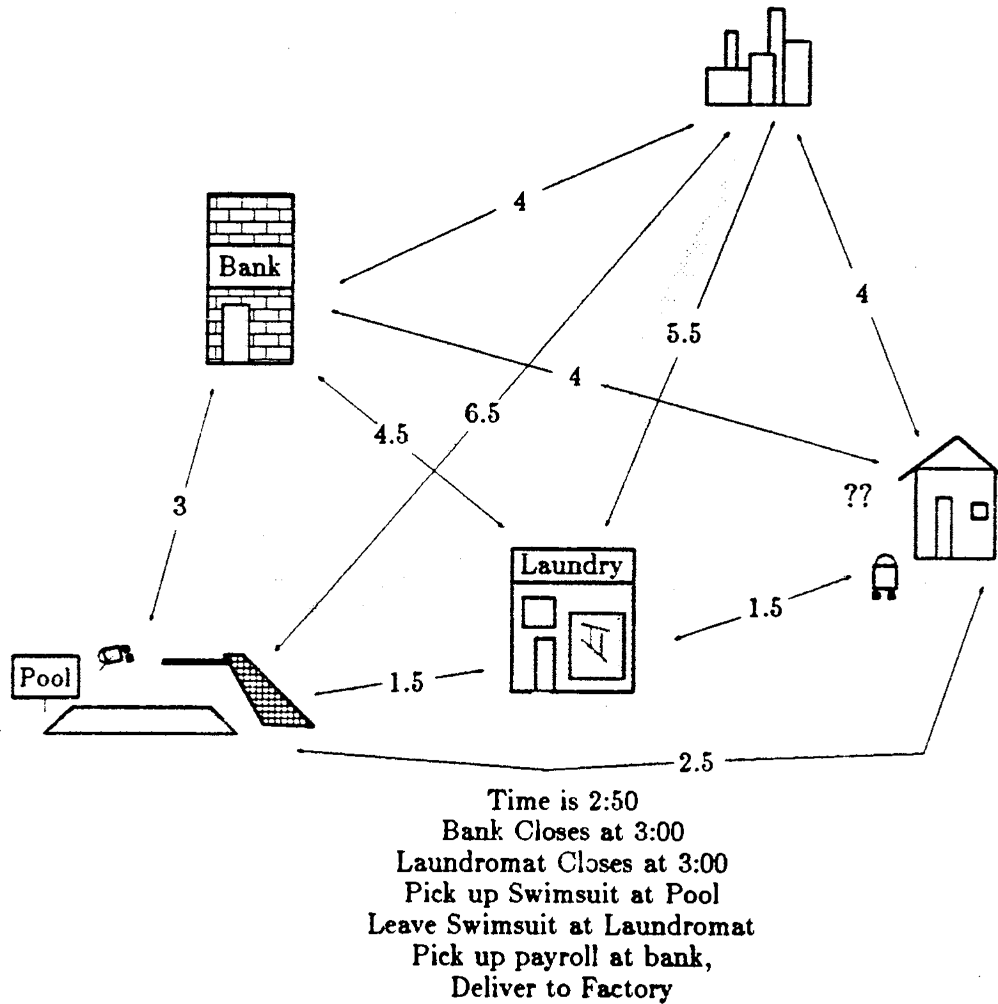


Figure 1.5: Workstations to be Visited in a Limited Time

amount of time) as a set of state changes does not convert the problem into one solvable by a situation-based system. A new Strips-like system could be created where operators updated a clock and goals expressed temporal concerns such as `time < 3pm`. However, the Strips matcher (that part of the planner that selects the next operator to be applied) could not tell which sequence of operators would be most promising in satisfying these new goals. The planner would be able to notice when a plan it has built will cause a deadline violation, but it would be incapable of determining the sequence of operators that would

be likely to avoid such a violation, without first trying them out — a strategy that is, in general, intractable. Augmenting the matcher to provide it with the necessary predictive abilities is not a simple task. In fact the creation of such a system is a large part of what this dissertation is about.

1.2.2 Resource Allocation

Renewable resources can cause problems for Noah-style planners in much the same way as deadlines and other efficiency concerns. A renewable resource is some resource that is required in some amount by some tasks and is produced in various amounts by other tasks. Money is a good example of a renewable resource; some robot tasks will use up the supply of the resource (e.g., spending some money) while others increase the robot's supply (e.g., earning money). Suppose a robot is starting out with $20, has five errands to perform (each of which will cost $10), and will make a trip to the bank to withdraw $30. To a least-commitment planner the order of the tasks that the robot is to make will appear inconsequential; the robot starts out with enough money to perform each of the errands and spends in total no more than it takes in. It is not until an attempt is made to order the tasks that it will be noticed that many of the possible orderings can not be legally executed. In a least-commitment planner such an ordering will never be created; the conflicts with therefore never be detected.

Resource and efficiency constraints can combine to produce problem domains in which it is difficult for traditional systems to function. Imagine the construction of a defensive plan for executing a double-play in baseball. The task for the planner is to assign the duties for each player in the field should a ball be hit to a particular part of the field. A typical plan for when the ball is hit near the first-base line (see Figure 1.6) would be to have the first-baseman retrieve the ball, have the short-stop cover second and the pitcher take first. If the pitcher went after the ball then the rest of the plan would have to be changed.

The plan for every step of the play is dependent on the plan that was used in the step before. The subplan that has the ball retrieved the fastest may not be the best play; the person retrieving the ball may have to hold onto it for several seconds while another player moves over to cover the appropriate base. Dozens of possible plans for executing the play can be developed; all of them operate under very tight temporal, spatial, and resource constraints. Only a few plans will succeed, but the information necessary to distinguish those plans from the ones that will fail cannot be easily contained in the task network. The task network cannot have restrictions on one plan linked to another plan unless those

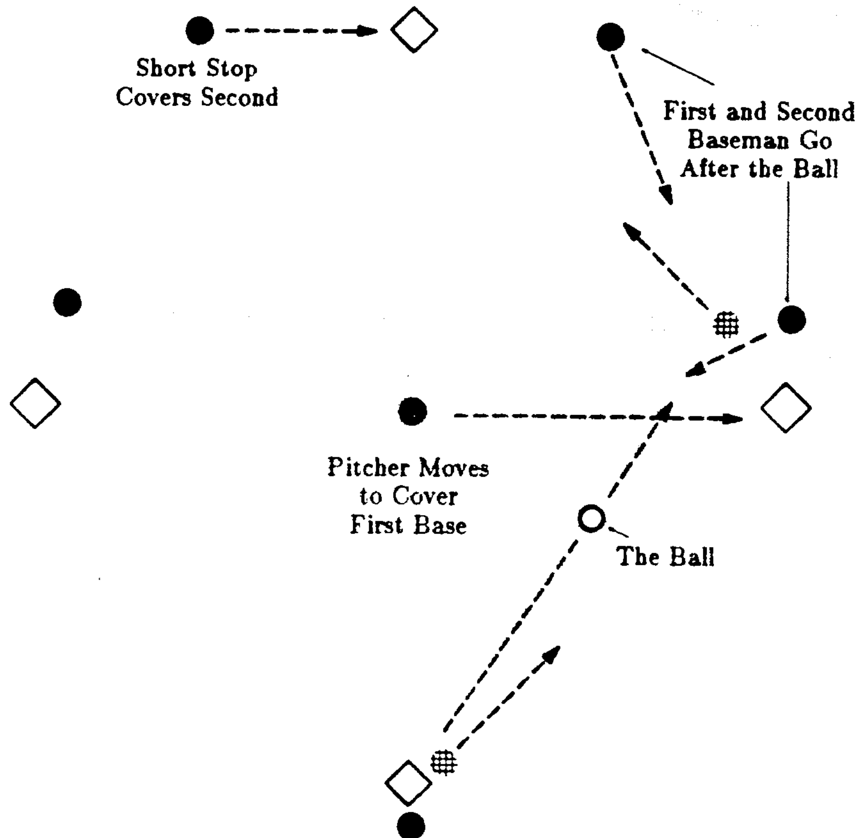


Figure 1.6: Planning a Double-Play

two plans are somehow linked, perhaps by an ordering link. However, if ordering links were made explicit for the baseball problem then every play that could possibly work would have to be explicitly stored in the task network. What *should* be in the task network is a plan such as:

1. retrieve ball
2. throw out runner and batter

This plan cannot exist in the task network because there is no way to expand the second step without knowing how the first step was expanded. Instead, a Noah-like system would

have to have each of the plans it might want to use worked out in detail and stated explicitly in the network.

1.2.3 Domains Containing Loops

Many problems in computer programming and real life involve repeated tasks or loops. A repeated task is one where the subtasks must be done over and over again some number of times. Noah was capable of handling some repeated tasks through the use of a *replicate* node. The replicate node was a normal node in the task network with the additions that it contained a counter and the ability to copy itself. If ten screws needed to be screwed into a machine, the plan for inserting one screw would be developed and then placed into a replicate node with a counter set to ten. During execution, when a screw needed to be inserted the node would be copied, instantiated with the proper screw data, and have its counter decremented. When the counter was down to the last iteration the replicate node itself would be used, rather than a copy being made.

Drummond [Drummond 85] reworked the task networks used by Noah into a *plan net*, a structure that integrates the basic features of the task network with the iterative abilities achieved by Noah's replicate nodes. However, both the plan net and the replicate nodes are incapable of representing repeated tasks where either the exact number of iterations is unknown or a single iteration leaves the world in a procedurally defined state.

For example, suppose the task is insert ten screws. This would be expanded to a replicate node with a counter set at ten and a pointer to a plan for screwing in a screw. The plan for screwing in a screw is something like: repeat as needed(insert screwdriver. turn clockwise one-half turn). Screwing in a screw is a repeated task. The exact number of iterations is not known in advance, though if the physics of the screw is known (i.e., how many threads over the length of the screw) then the number of iterations could be calculated. After each iteration the screw would be screwed further into the hole, yet Noah would need to have its representation considerably supplemented to say anything much beyond "the screw is screwed in N one-half clockwise turns."

What would be desirable would be to have a planner that could say that "each screw will need 12 ± 2 turns in order to be screwed in," After each iteration the system should also be able to make some estimate of the progress towards the desired goal such as "the screw is three-quarters of the way." Finally, it is necessary for many planning tasks that steps in repeated tasks be carefully coordinated with steps in other repeated tasks, or in

linear tasks. If the insert screw task involved screwing in the screw and then tightening it, it may be necessary for the planner to coordinate things so that all ten screws are screwed in before any are tightened. This is impossible to do in the systems designed by Sacerdoti and Drummond unless the tightening task is specified at the top level of the task network (i.e., the task would have to be specified as screw in and tighten ten screws rather than insert ten screws where the plan for a screw insert is screw in screw then tighten).

The whole least-commitment philosophy is designed for domains where interactions dictate ordering decisions — not the other way around. When insufficient commitment is made during the planning process, either in plan choice or in plan ordering, the planner can spend a considerable amount of time exploring unprofitable or infeasible plan avenues — plan possibilities that would be obviously unwise if only it had made and stuck to an earlier decision.

Domains that involve efficiency constraints, loops, and continuous resources can have plan interactions that are dependent on how the pieces of the plan are ordered. Least commitment planners are designed to make ordering decisions only when adverse interactions are detected. They therefore have difficulty in the domains described above because no interactions appear until an ordering is enforced and the planners will not enforce an ordering until an interaction appears.

1.2.4 Mobile Robot Feedback Tasks

One of the goals of automatic planning is to be able to plan the actions of an autonomous mobile robot — one capable of moving about and dealing with the world on its own. One of the first steps towards producing such a robot is to create the software for having the robot navigate about in its environment. Since no machinery is completely accurate, and since the environment is never completely predictable, an autonomous robot will have to use feedback in order to operate and correct itself. Unfortunately, feedback tasks usually involve tight temporal constraints, renewable resources, and loops — the three problems that are beyond the abilities of traditional least-commitment planning systems. The remainder of this section gives some details about the robot feedback domain, the difficulties it entails, and the planning problems it produces.

For a person programming a robot, it is relatively easy to build routines for having a robot follow a wall, locate a landmark, and change its direction; building a program to do all of these at once turns out to be more difficult. Every situation seems to require a unique

program. You might think that programming for a simple task such as go down the hall and turn left at the second doorway would require only the selection of the proper routines and the substitution of some parameters; unfortunately this situation is not often the case. The routines have intricate interaction with one another that sometimes require delicate manipulation of motor speeds in order to run efficiently. Sometimes, a massive reordering or merging of the steps in the routines is needed for them to function as a workable whole. Using an automatic planner to produce robot programs means that the planner has to be able to temporally coordinate the various robot feedback-sampling routines; a task that is not only beyond the abilities, but one that also goes against the grain of any Noah-like planner's policy of least commitment.

Feedback is the process of sensing the environment around you and planning actions based on the sensory data. For a robot, feedback can be used for avoiding obstacles, maintaining a desired course, picking up objects, and a variety of other actions where the robot might interact with the world. The most general feedback schema is for the robot to:

1. perceive
2. compute
3. act

Usually this schema is executed in an indefinitely repeated loop; the robot using its sensors to guide the actions of its motors.

A robot feedback loop might be used in a situation where a robot is trying to walk down a long hallway. To avoid bumping into the walls as it travels, the robot might decide to keep a constant distance from the right-hand wall, say five feet. A plan to accomplish this is for the robot to point its distance sensor at the right wall and measure the distance to that wall every few feet. If the robot records a distance of less than five feet it turns its steering wheel slightly away from the wall; if the distance measured is greater than five feet the robot turns slightly towards the wall. The distance traveled between observations depends on the robot's estimated accuracy; a highly accurate robot could travel large distances between observations, an inaccurate machine would have to check its position more often. With a feedback strategy of this sort, the robot will follow a path similar to that shown in Figure 1.7.

Such a program is difficult for a Noah-type problem solver to create because the plan involves a loop and because the subplans have specific spatial constraints on when they are

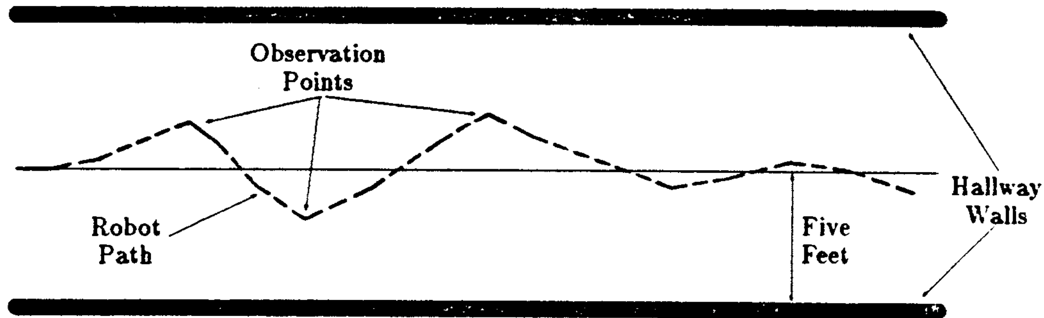


Figure 1.7: A Robot Following a Hallway Wall

to be executed (i.e. when to take an observation of the hallway wall is dependent on how far the robot has traveled since the last observation). However, the real difficulties of this domain do not appear until the robot is given several feedback-oriented tasks to perform simultaneously. Two feedback plans can interfere with one another. If a robot is using a single sensor to look for landmarks and to avoid obstacles it must carefully coordinate its observations with its motor commands in order to ensure that it does not look at the wrong place at the wrong time, as in Figure 1.8.

As another example, Figure 1.9 shows a situation where our robot must follow the right-hand hallway wall but also is looking for the second left-branching corridor. In this situation, the robot's plan is much more complex than in the case of simply following a hallway wall. One strategy for finding a left-branching corridor is for the robot to maintain a straight course along the right wall, and watch for a sudden increase in the distance between the robot and the left hallway wall. If the robot is equipped with only a single distance sensor then it must turn that sensor from the right (where it is keeping the robot on a straight course) to the left (where the robot will use it to spot the branching corridors).

Fortunately, neither of the tasks that the robot is trying to accomplish (travel in a relatively straight line, locate the second left-branching corridor) requires continuous observation. The robot only needs to look for the branching corridor every time it has traveled a distance equal to the width of one of the branching corridors. Observations spaced in this manner assure that one observation will be made of each corridor. Similarly, observations of the right hand wall need only be made whenever the robot has traveled far enough that it may have drifted substantially to the left or right.

Conceivably there may be plenty of time for the robot to accomplish both tasks without

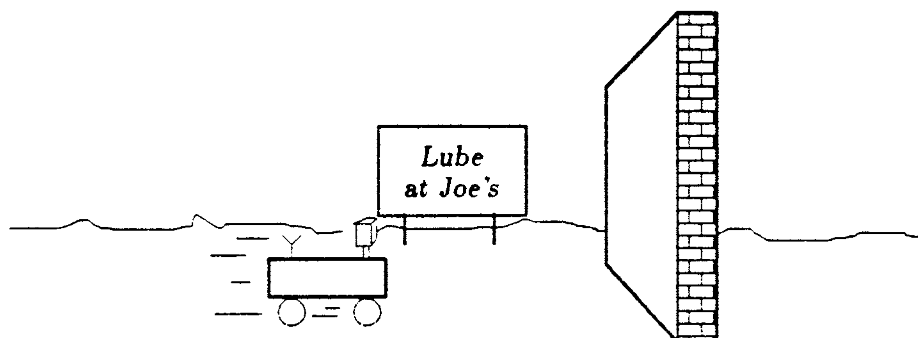


Figure 1.8: A Robot Looking in the Wrong Direction at the Wrong Time

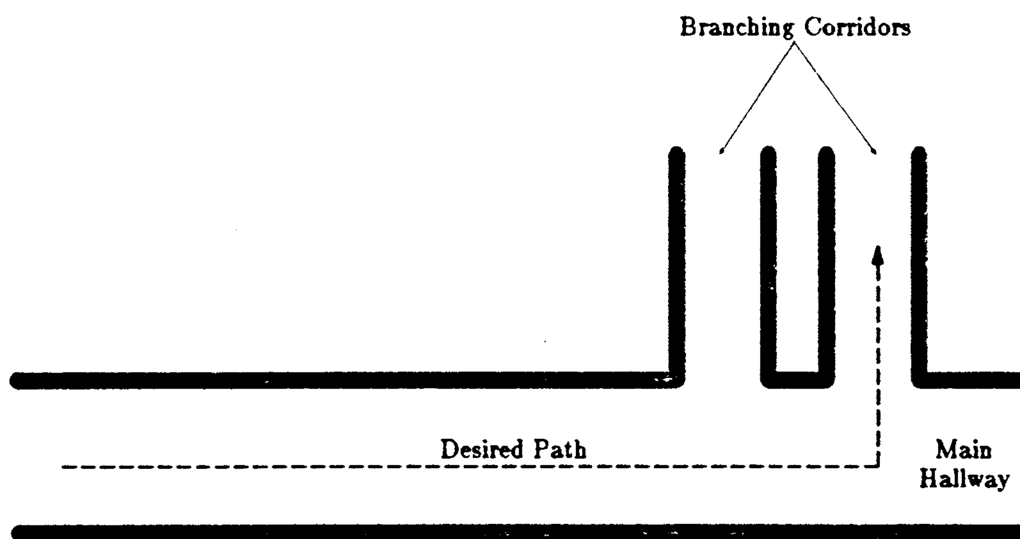


Figure 1.9: A Robot Looking for the Second Left Turn

running into a resource conflict. The factors that will decide this one way or the other are:

- The accuracy of the robot
- The width of the branching corridors
- The speed at which the robot is moving
- The speed at which the robot can reorient its distance sensor.

While the first two items are beyond the robot's control, the last two can be manipulated by the robot. By slowing the robot's speed and by moving the robot's sensor at its maximum velocity, a workable plan can almost certainly be created. If the robot has any deadlines to meet, then the robot's speed may have to be controlled more carefully as part of the overall plan: sped up whenever possible, slowed down whenever more time is needed to make observations.

```
(plan-follow-right-wall
    (precondition (not (at corridor)))
    (loop every-4-feet
        (subtask look-right)
        (subtask measure-distance)
        (subtask make-steering-correction)))

(plan-spot-corridor
    (loop every-12-feet
        (subtask look-left)
        (subtask (if (at corridor) stop))))

(plan-turn-left
    (precondition (at corridor))
    (subtask steer left)
    (subtask drive on))
```

Figure 1.10: Plans for Navigating Mobile Robot

The three tasks that the robot must carry out — follow the right-hand wall, look for the second left corridor, turn left at the corridor — are all shown in Figure 1.10. Aside from the looping structures, these plans could fit well into a Noah-style task network. However, the looping structures contain all of the important information about the plans — information critical to producing a plan and eventually an actual program for the robot.

Aside from the preconditions explicitly stated in the plans, there are a number of conditions that are implicitly stated as part of the loop. For the wall-following and corridor-finding plans there is an implicit precondition on every iteration of their loops that the subtasks of those loops can be completed before the robot moves the specified distance. These preconditions cannot be tested on the plans in isolation, but must be checked as the plans are expanded and combined. Least-commitment planners check preconditions only once at each level of expansion and expand in a breadth-first manner. Yet the expansion of each plan level, for each of these plans, is in part dependent on the expansions used for the plans with which they will be merged. For example, details on how fast the robot should

travel down the hallway cannot be decided until it is known how fast the robot will be turning its head from the right to the left, and how often and when it must move its head.

In summary, the robot feedback domain has proven to be a difficult domain in which to do automatic planning because the plan steps must be carefully coordinated with one another. Plan coordination is not a task that least-commitment planners are well suited for. In order to coordinate time-dependent plan fragments (such as those that arise in the robot feedback domain) the effects that one piece of the plan might have on others must be taken into account. These effects are often dependent on the order in which the pieces of the plan are to be executed. Because of the importance of ordering, these effects are often invisible to least-commitment planners.

1.3 Planning: A New Paradigm

After the advent of least-commitment planners, the planning problem became defined as the successive refinement of a plan while detecting and eliminating adverse interactions among the plan steps. The first two sections have shown that least-commitment planners have solved this problem for only one case of plan-step interactions — interactions where a context-independent precondition of one plan step is contradicted by the context-independent effect of another plan step.

Many real-life problems involve other types of interactions — interactions that involve efficiency issues and resource consumption — interactions that use cumulative quantities, instead of context-independent facts, as their representation. Problems containing quantitative interactions can be exacerbated by the particular plan chosen to accomplish a task and by the ways in which the plans for multiple tasks are coordinated with respect to one another.

The research described in this dissertation expands on previous automatic planning techniques in an effort to create a new planning paradigm where quantitative issues in planning, such as temporal expenditures and deadlines, can be reasoned about and planned for in detail. The central thesis of this work is that quantitative planning issues can be reasoned about through the exploration of the total orderings of the various plan-actions and events. This thesis is backed up by an implemented planning module that uses empirically derived heuristics to explore the space of totally ordered plans efficiently. The rest of this section, and the remainder of this dissertation concentrate on examples, algorithms, and

the implementation of a planning module for dealing with planning problems containing quantitative temporal interactions. This module is called the ***task scheduler***.

1.3.1 Exploring Total Orderings

In the planning paradigm being explored in this dissertation the planner's job is to create a sequence of primitive actions that, when executed by the robot, will accomplish the desired task. The plan thus produced must

- Be free of adverse interactions among the plan steps
- Have all the necessary resources explicitly assigned to each task
- Have the plan steps temporally coordinated with one another
- Explicitly include state-transition tasks
- Contain estimates of how often loops will have to be executed.

The planning problem as seen this way bears some resemblance to the basic scheduling problem explored in computer science and operations research. Like many of the scheduling problems in those fields, the solutions to the planning problem all appear to be exponential in their computational complexity. However, task scheduling is not quite the same "scheduling" problem as has been explored in these other fields. Task scheduling is accomplished, in this research, through the incremental extension of selected schedule *prefixes* until a complete schedule is found. A schedule prefix of length n is a partial schedule that starts with the first task to be done, and continues sequentially for $n - 1$ additional tasks. Figure 1.11 shows the schedule tree and some prefixes for a simple problem consisting of four tasks.

There are many different possible orderings for a given set of plan steps. Figure 1.11 shows all the different ways in which four plan steps can be arranged. An ordering of plan steps that starts at the root of the tree and continues on towards a leaf is a schedule prefix. In the Figure AC is a prefix that can be expanded upon in two different ways — either with B or D. The task scheduler's job is simulate various plan prefixes, and search through the simulations to find one that leads to a good plan. Only by searching through the simulations of the plan prefixes can a plan be found that has the context-dependent effects taken into account.

In the figure there are 24 or 4! possible schedules that could be found for the four tasks. There are more than twice that number of nodes in the search tree. Like traditional

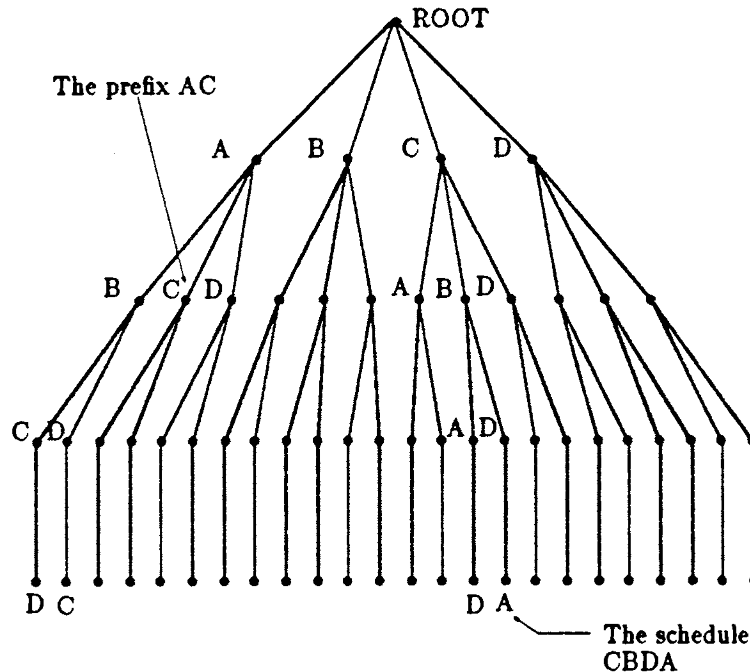


Figure 1.11: Plan Prefixes and Total Orderings

scheduling problems, many task-scheduling problems are exponential in their computational complexity. However, in the planning problems in the mobile-robot domain, tasks can sometimes be repeated an indefinite number of times. This allows there to exist an indefinite, if not infinite, number of schedules that could be formed for these problems.

An additional difficulty is that, like many other scheduling problems, planning often deals with specific time constraints. While some of the scheduling algorithms that have been developed to handle various forms of temporal constraints, all of those algorithms break down when the constraints can change depending on the ordering chosen by the scheduler. In planning, even in the limited robot-feedback domain, task ordering can greatly influence inter-task constraints. For example, suppose that for a robot to roll down a hallway, without running into walls, it must check its distance from the walls every ten feet that it travels. If after traveling 49 feet the robot is scheduled to determine its absolute position in the world as a subtask of some high-level goal, a good planner should not need to schedule a check of the robot's distance from the hallway walls when the robot reaches a distance of 50 feet from its starting point. Checking the distance from the wall is meant to occur

every time the robot travels ten feet from the last time it knew its distance from the walls. Therefore, if the robot has just calculated its exact position in some other way it should not do another wall check until it has traveled an additional ten feet.

The flexibility of the scheduling constraints, the need to balance time, space, and other resources against one another, and the need to schedule poorly specified loops combine to make the task-scheduling problem beyond the abilities of all but heuristically-based scheduling algorithms.

Search is necessary to find a good total ordering of the tasks given to the task scheduler. As with all search problems, the quality of the search algorithm depends on its abilities to prune the search-space. The task scheduler described in this work searches through the space of totally ordered plans. This space can be thought of as the different paths that can be traveled through the task network described in section 1.1.2. The task networks used by Noah-like systems contained a wide variety of constraints that can be used for guiding the search. Minor modifications to the network, in order to allow loops and temporal information, allow the search to be further guided. Much of the *legal* search-space (i.e., those schedules that do not violate explicit ordering constraints) may be pruned away through the use of heuristics based on the constraints of the problem. These constraints, temporal, spatial, and resource-based, can be combined with ordering constraints to eliminate large parts of the possible search-space. Unfortunately, the part of the search-space that remains might still be prohibitively large.

One way of searching through the remainder of the possible search-space is to rate partial schedules on their probable efficiency, and then to pursue the most efficient looking schedules. In this way the majority of the possible schedules for a set of tasks are left undeveloped. It is only the most promising of schedules that are expanded at all, and of those only a tiny amount will be completed.

Limiting the search-space by using efficiency as a criterion for expansion is the method used by the system described in this work. It uses total expected execution time as the metric by which to judge efficiency.[1] In other words, this system tries to produce a plan that is not only functional, but fast.

The task scheduler can be thought of as simulating the execution of alternative schedulings of the tasks that have been passed to it. It then selects and returns what it considers the most efficient schedule of those that meet all of the problem constraints. Included in the

[1] In domains where time is not explicitly represented, this metric becomes identical to looking for the plan with the fewest steps.

output of the task scheduler are the state-transition tasks necessary to execute the chosen ordering.

A transition task is a task that is not included as a subtask in any of the plan tasks that are passed to the scheduler, but that performs a necessary precondition of one or more of the tasks in the schedule. An example of a transition task is the job of pointing the robot's sensor in the proper direction for gathering the desired data. The actual sensing is the task that is passed to the scheduler; the pointing of the sensor and any other initialization procedures are the transition tasks added by the scheduler.

It is the transition tasks, and their relationship to the temporal constraints in a problem, that often are the major factors in determining a schedule's efficiency. If the amount of time necessary to accomplish all of the transition tasks is considerably less than the amount of time before the main task may be started (due to some other constraints) then the intermediate time is probably being wasted. In such a case it is usually better to do some other task, one that has a better match between its start time and the amount of time necessary for its transition tasks. In other situations, one task may require almost no time to accomplish its transition tasks at a certain point in the schedule. It is probably most efficient to do that task at that point.

Heuristics like those above are used by the task scheduler when building up a schedule. Other relationships between transition tasks, resources, and deadlines can be used to predict orderings that are doomed to failure — without actually simulating that particular ordering. Unfortunately, different scheduling problems are affected in slightly different ways by the factors which these heuristics measure. To overcome this problem, the scheduler analyzes situations where it is forced to backtrack, that is, situations where the orderings it tried lead to infeasible schedules. The analysis is used to adapt the scheduler's heuristic rating functions by dynamically adjusting the weights on the heuristics it uses. For example, if the robot has several tasks at various workstations to perform and the schedule prefix the scheduler was developing lead to a deadline violation for a task at a distant workstation, the scheduler would analyze the situation and decrease the importance of the minimize-travel heuristics and increase the importance of doing tasks with approaching deadlines.

The scheduling process is made more efficient through the application of scheduling *critics*. The critics are called into action whenever certain probable scheduling difficulties arise, such as when the transition delay necessary for initializing one task would cause another task to go over its deadline. The appropriate critic is selected and passed to the search procedures which eliminate the affected areas of the search space. The critics are

used for dynamically rescoring and retailoring parts of the scheduling tree in accordance with newly discovered properties of the problem (e.g., that the transition delays of task A interfere with the deadline of task B). The critics are created by the scheduler, for a particular problem, in order to supplement the heuristics in the standard rating functions used by the system. The critics, rating functions, and search heuristics all make annotations to the schedule. The final schedule contains the reasons why particular tasks were chosen for their place in the schedule.

The task scheduler searches the space of totally ordered plans. This search is necessary for the detection of quantitative task interactions such as the potential deadline violations that occur in efficiency problems. Unfortunately, like most scheduling problems, the space being searched is combinatorically large. It is therefore necessary to perform a heuristic search over the space of totally ordered plans. The heuristics, both domain-dependent and independent, combined with the adaptive scoring with which the heuristics are applied are sufficient to keep the performance of the system quite acceptable.

1.3.2 Solving Mobile Robot Problems

Section 1.2.4 showed that the programming of mobile robots was a difficult planning problem that entailed dealing with quantitative interactions and the coordination of loops. To demonstrate the scheduler's plan-coordination abilities, the task scheduler described in this work has been integrated into a planner, called Bumpers, designed to create feedback-driven navigation plans for a simple mobile robot. These plans involve the integration and coordination of two or more robot-feedback loops.

Bumpers is not a general-purpose planner. It contains several special modules designed specifically for handling robot navigation tasks. The way in which these modules are arranged also differs from most typical planners. Bumpers is not a hierarchical planner. Instead, a task works its way through the system only once before a final plan is created. The elimination of hierarchical plan expansion allows the process of creating plans for repeated tasks (a process referred to as *loop expansion*) to be done in a more straightforward manner.

Bumpers is a *single expansion* planner. This means that there is but one way for the system to plan for any single task, and that plan contains a series of subtasks that can be considered primitive actions; no hierarchical expansion is involved. While there is only a single way to expand any one task there is a potentially infinite variety of ways of combining the plans for two or more tasks. It is in the coordination of the solutions for multiple tasks that Bumpers uses its scheduler, and gets its power.

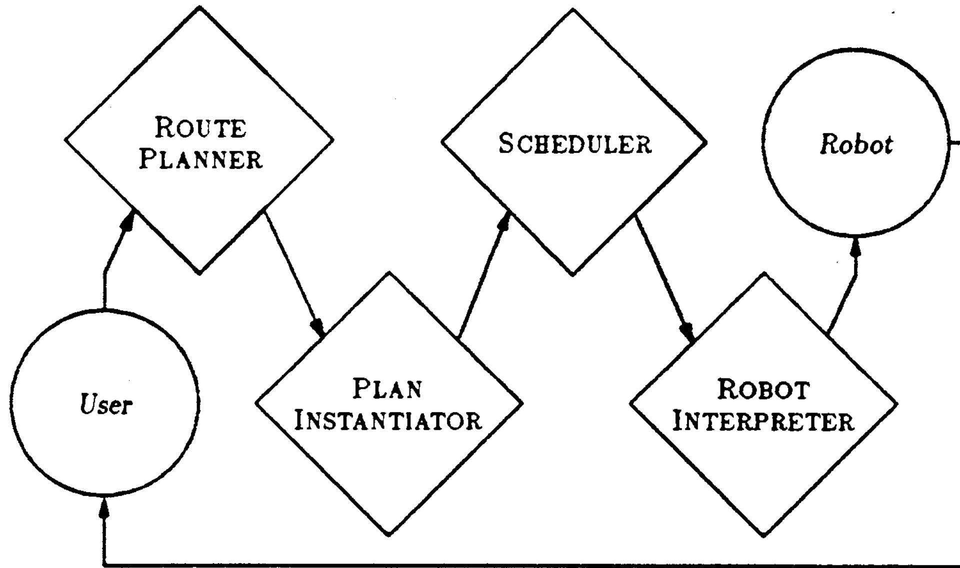


Figure 1.12: The Architecture of the Bumpers System

The Bumpers system is composed of four basic modules:

- *Route Planner*
- *Plan Instantiator*
- *Scheduler*
- *Robot Interpreter.*

Figure 1.12 shows the basic communications paths between the modules. The system is designed to generate the robot instructions necessary to move a simple robot about in the world, given its starting and destination points as input along with a map of the robot's world. The output of the system are the motor and sensor instructions necessary to maneuver the robot (a machine with very limited dead reckoning abilities) to its destination point. The instructions generated must be sufficient for the robot to monitor its progress through the plan and successfully complete its task, barring an unexpected amount of mechanical deviation.

The input to the system first goes to the Route Planner, which generates a safe path for the robot to traverse from its starting to destination point. This route is then broken

into various sections depending on the type of feedback that is required to guide the robot through them. These route segments are then passed off to the Plan Instantiator.

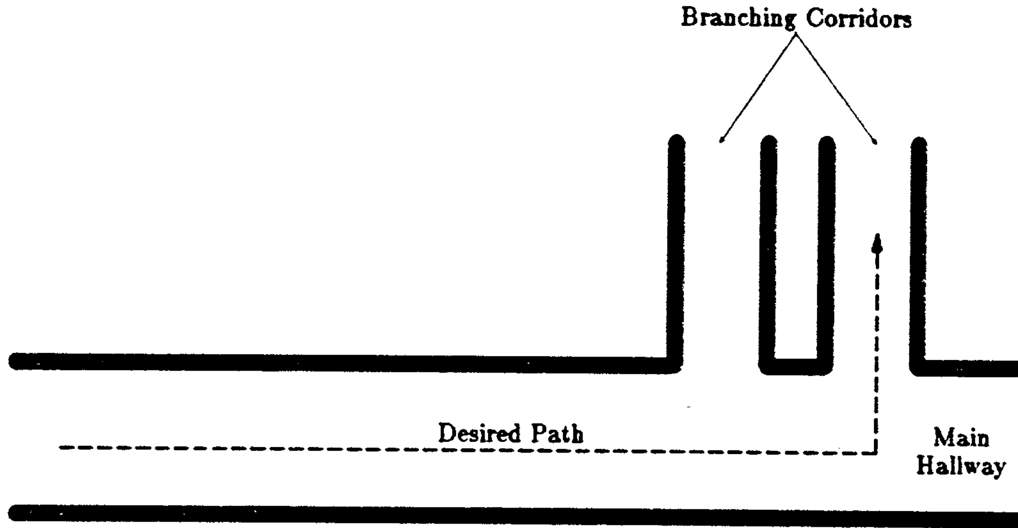


Figure 1.13: A Robot Looking for the Second Left Turn

The Plan Instantiator takes each segment of the planned route and searches the plan library data-base for an appropriate schema by which to accomplish that segment's traversal. The route needed to traverse the path shown in Figure 1.13 uses schemas like those given in Figure 1.10 (section 1.2.4 gave a full description of this problem). When the proper schema is found, its *plan variables* (those parameters in the schema that vary depending on which map is being used) are then calculated and incorporated into the schema — producing a plan for the traversal of the route. The information in the original map given to Bumpers may contain various uncertainties. Long hallways in particular have some uncertainty to their exact lengths. These uncertainties are passed on to the scheduler through the values instantiated into plan variables.

The Scheduler takes all of the plans presented it by the plan instantiator and breaks each plan down into its component subtasks, many of which will involve feedback loops. The constraints on each of the subtasks are extracted from the plan schemas. These constraints contain timing and distance bounds between executions of the robot's data gathering tasks. Some bounds on the size of the hallways are used to set up initial estimates on how many iterations of the data-gathering motor-correction loops will be needed. The scheduler then

```
(set head-speed 18deg/sec)
(set robot-head +90)
(set robot-speed 1.0fps)
(set drive on)
(measure-distance)
(make-steering-correction)
(LOOP
    (wait 4 seconds)             ;;; while the robot moves four feet
    (measure-distance)
    (make-steering-correction)
    (wait 4 seconds)
    (measure-distance)
    (make-steering-correction)
    (set robot-speed 0.4fps)
    (set robot-head -90)
    (wait 10 seconds)            ;;; while the robot's head turns around
                                 ;;; and the robot advances four feet
  UNTIL (at corridor)
    (set robot-head +90)
    (set robot-speed 0.0fps)
    (wait 10 seconds)            ;;; while the head moves again
    (measure-distance)
    (make-steering-correction)
    (set robot-speed 1.0fps))
(set robot-steering left)
```

Figure 1.14: Plan for Navigating Robot to Second Left

starts performing a simulations to determine the best ways to integrate the subtasks into a coherent plan. The simulation may involve creating several copies of the subtasks that are contained in loops (this process is called *loop unrolling*). When the loops have been sufficiently unrolled in order to determine a feasible schedule the repetitive parts of the schedule are *condensed* into a loop and more estimates are made on the number iterations that will be needed (see Figure 1.14). While doing the ordering, the scheduler inserts the proper timing and speed constraints to keep the feedback loops of the navigation routines in balance. Additionally, the scheduler adds the proper motor commands to move the robot's sensors and effectors between jobs, in cases where a single sensor is being used simultaneously for multiple subtasks. Because there is some uncertainty in the robot feedback domain, the part of the plan that guides the robot from one segment of the route to the next can become quite complicated. It is in these areas that the most difficult scheduling goes on.

The complete plan provided by the Scheduler is then passed to the Robot Interpreter.

This module assembles the final code for the robot from the instructions provided by the Scheduler's plan.

Bumpers overcomes many planning difficulties in solving problems in the robot feedback domain. Primary among these is the coordination of temporally and resource-restricted subtasks into a coherent plan. The fact that these tasks usually take the form of loops, which must be coordinated, only makes the job more difficult. Loops with a poorly specified number of iterations can hide the actual amount of time and resources needed to fully execute them. When several loops must be integrated the final time may bear little relationship to the time required for the individual loops. The task scheduler and its abilities to unwind loops, detect their interactions with one another, and propagate the uncertainties of resource and time usage are critical to the successful planning in domains such as robot feedback.

1.4 Overview of This Dissertation

There are a myriad of planning problems, a large portion of which would benefit from a planning strategy that included task scheduling. It is not possible to explore each of these problems in detail. Rather than cover just the general problems and strategies, this dissertation focuses on the solving of some robot feedback-sampling problems in the domain of navigation, a domain tackled by the Bumpers planning system. Experiments from other domains will also be presented at the end of this work.

At the end of this chapter is a review of some of the relevant literature on robot planning, scheduling, and the representation of time.

Quality planning requires some information about the domain in which the plans are to function. Chapter Two presents the necessary information, along with representation schemes for that information, for the mobile robot feedback domain. Not surprisingly, this information is somewhat more detailed than has been the case for previous planning systems. This is due to the more detailed quantitative reasoning done by the Bumpers system than has been attempted by Noah-like systems.

The third chapter concentrates on resources and their role in planning decisions. The ways in which resources are used in the mobile robot domain is discussed in Chapter Two; Chapter Three elaborates on the role resources, and their interactions in plans, affect the planning process. How resources are represented and manipulated by the task scheduler is also discussed.

Chapters Four and Five give the algorithms used by the task scheduler. Chapter Four presents the basic algorithm, its special representations, and the heuristics that are used to make it efficient. The fifth chapter shows how the algorithm works for the scheduling of loops. Minor additions to the scheduler's internal representation of tasks must be made in order to handle loops well. These modifications are also presented in Chapter Five.

A variety of experiments have been performed with this system. The task scheduler has been used as a stand alone system and as part of larger planning programs. Several different domains have been tested, and abstract problems have been created to provide benchmarks by which to judge different parts of the program. The experiments and their results are detailed in Chapter Six. The final chapter attempts to draw everything together. Performance of the program is discussed as well as what the performance says about a task scheduler's usefulness in the world of reductionist planning systems.

1.5 Review of Related Works

The central thesis of this work is that quantitative planning issues can be reasoned about through the exploration of the total orderings of the plan-actions and events. This is basically a scheduling approach to planning. Both Automated planning and scheduling have been the subjects of large amounts of research in recent years. Additionally, in the past few years, several temporal representation systems have been developed. All of these areas play some role in this research, and are reviewed in this section.

1.5.1 Planning Systems

Many of the current robot planning systems are direct descendents of GPS [Ernst 69] and Strips [Fikes 71]. These systems were described in detail in section 1.1.1. In order to get the system to concentrate on the "important" parts of the problem first, Sacerdoti created Abstrips [Sacerdoti 74], which organized the Strips operators hierarchically. Soon afterwards, Noah [Sacerdoti 77], the first least-commitment planner, was created (see section 1.1.2).

Noah used the least-commitment strategy to try and avoid backtracking. Another strategy for avoiding backtracking was used in Hacker [Sussman 75] and Interplan [Tate 75]. These systems searched through the space of plans, much as Noah did. However, they chose an arbitrary total ordering of the plan steps, and had less of a hierarchical structure

to their plan networks. Knowing that the ordering of the plan steps was arbitrary, the systems would then proceed to *debug* the plans — searching for adverse interactions among the steps, and reordering or replacing certain steps in order to alleviate the interactions. In general, this strategy of debugging proved more expensive and difficult than the least-commitment strategy. Almost all planning systems since Noah have been grounded in the least-commitment paradigm.

The next major steps in least-commitment planning were done by Tate [Tate 77], [Tate 76]. His Nonlin program corrects several of the deficiencies found in Noah. In particular, Nonlin keeps a record of all the alternative orderings and expansions that it does not use. If a point in the planning process is ever reached where an unresolvable conflict arises between different parts of the plan, then Nonlin can backtrack to one of the earlier decisions it has made and choose one of the alternatives.

Nonlin is also able to detect a larger set of plan-step interactions than Noah. Like Noah, Nonlin detects occasions when one plan-step has an effect that contradicts a precondition for another step at the same expansion level. Additionally, Nonlin is able to detect conflicts that occur between plan steps at different levels in the hierarchy. This is done by keeping and updating a partially-ordered network of preconditions and effects. When a plan step is expanded the expansion is substituted into the network for the original step. All of the dependencies and effects from the original step are then attached to the expansion. The new effects and preconditions from the expansion are propagated through the network so that any potential conflicts, detectable by the more detailed information of the expansion, are seen. When a potential conflict is detected an ordering link is added to the network in order to assure that the conflict will not actually arise.

Ordering links that would cause a failure of one of the problem goals are never added to the network because of the GoSt, or goal structure. The GoSt is a data structure that keeps track of which effects are being used to achieve the various preconditions in the plan. By referencing the GoSt whenever an ordering link is being added to resolve a conflict, Nonlin is able to avoid creating new problems for itself. The GoSt can also be used for analyzing why the system makes some of the decisions that it does. Nonlin uses the GoSt to help direct backtracking. Potentially, the GoSt could be used during plan execution in order to discriminate between significant, and insignificant, departures from the plan.

One of the drawbacks of least-commitment planning, is that it is only useful in domains where events are very predictable [Dean 85]. One subclass of unpredictable events is where the events are uncertain, but the odds and possibilities of the uncertainty are known. In

such cases decision theory may be used to guide the planning process. Early research in robot planning assumed that the only uncertainty that would appear in robot tasks would be completely predictable from the top level task description. Therefore the system could juggle the utility of doing some extra task to eliminate some of the uncertainty against the *cost* of the task. Examples of such systems can be found in [Munson 70] and [Munson 71]. These papers are some of the first to suggest the importance of rating plans on their possible *payoff* and on their probability of success. By being able to calculate the *utility* of a plan it is possible to formalize the decision-making process that chooses one plan over another in a given situation.

Jacobs built on this work in [Jacobs 73]. Here, a simulated cockroach named Percy ran about in a simple world trying to eat, build a nest, and avoid being stung by the scorpion that also existed in the world. Percy chooses its plans based on the decision theory operators developed by Munson. The utility of each plan is given an initial value at the start of the program run. These values are modified based on the success or failure of a plan's simulated execution. The Percy program's performance improves (i.e., it reduces the amount of time required to build its nest while also lowering the number of stings it receives) the longer it runs.

[Feldman 75] did a similar simulation for a variation of the *Monkey and Banana* problem. [Coles 75] describes an actual implementation of this problem using Jason, the UCBerkley robot. In that problem situation the robot had to get through a blocked doorway. The tool to unblock the doorway was available, but in a room that also had several objects that mimicked the tool. The robot had a sensor that would recognize a tool from a non-tool with certain probability of being correct form a far distance, and had a much more reliable sensor that could only be used from closeup. The robot had to decide which of the several objects in the room to go towards, and when to do the tests. Jason was able to perform quite well by expanding the decision tree to its leaves. It was then able to choose correctly the probabilistically best plan for its situation.

Unfortunately, this strategy only works if there are a few discrete outcomes for any action and observation. In domains such as controlling a robot's movement, expanding the decision tree to its leaves is a computational impossibility. The number of actions for any given situation is large[2] and the places and amounts of error that can occur are infinite.

Noah-style, Strips-like, and decision-theoretic planning all rely on restricted domains to keep errors from creeping in during the execution of plans. Other planning systems

[2] A typical robot can move any of its motors to hundreds or thousands of different positions.

have been constructed to try and deal with errors when they occur. The Elmer system [Ward 82], [McCalla 82] models a taxi cab driver trying to reach a destination by taking instructions. The system inserts *monitor* tasks into the plan. These monitors are designed to look out for expected error conditions, (e.g., *if you pass the supermarket, you've gone too far*) and force the appropriate corrections.

Some work has also been done towards analyzing robot errors after they occur and then patching the plan. [Gini 83] suggests how a Strips-like planner could be used to do real time robot error correction. The expected state of the world could be carried through by Strips' add and delete lists. As execution goes on, the expected state is compared to the actual state. Any differences between the two cause the appropriate subtasks to be spawned. Unfortunately there is no guidance presented on how to decide what are the relevant features of the world (i.e. those features that need explicit observation tasks in order to be verified). Without such guidelines the system can only correct for expected errors, and the system starts to look very much like that in [McCalla 82].

There has been some work done in automated robot control that is similar in some ways to the Bumpers system. Taylor [Taylor 76] has created a system that automatically generates AL programs[1] for a simple assembly task (screwing a lid onto a box). Taylor states that there are seven important points in putting together a general purpose system of this sort:

1. Have an adequate manipulator-level target language
2. Have a nice task-level specification formalism
3. Develop a suitable representation of object models along with a CAD system for computing relevant values needed by the planner
4. Form a situational representation that handles uncertainties
5. Have a large knowledge base of manipulator techniques to draw upon
6. Have a means for making coherent assembly strategies, so that interactions between assembly steps do not lead to inefficient programs
7. Have a good user interface so that the program can ask for help.

The work described in his paper concentrates mainly on the first five points. Thus, the resulting plans from the system have local optimizations but lack a global efficiency that would be provided by a planning system equipped with a task scheduler.

[1]AL is a robot programming language that has built in utilities for asynchronous feedback, motor control, etc. A good description can be found in [Finkel 74].

There has been much research done on feedback, but little on the actual planning of feedback tasks. One of the exceptions is the system described in [Brooks 82a]. This system inserts feedback steps into plan for maneuvering a robot manipulator in hand-eye coordinated tasks. Like the Bumpers system described in Section 1.3.2, Brooks' system is faced with decisions of where and when to schedule feedback tasks in order to ensure the successful completion of the robot's tasks. Unlike Bumpers, the domain in which this robot is working in is constrained neither in time nor by resource limitations. The emphasis of Brooks' research instead concentrates on the propagation of the accuracy and reliability constraints on the robot's sensors and effectors. The calculations on these constraints are used to derive exactly where and how much feedback is needed by the robot to accomplish each part of its task.

1.5.2 Temporal Relationships and Representations

The research described in this dissertation deals with scheduling. Scheduling deals with the placement of actions and events in time. It is therefore worthwhile to review some of the relevant research on the representation of time.

Two "types" of temporal relations arise in planning problems:

Absolute: those temporal relationships that are referenced off some fixed coordinate system such as a calendar. The distance between two events that have absolute times associated with them can be trivially calculated without any other knowledge about the events in the world.

Relative: temporal relationships that are referenced off other events going on in the world. The temporal distance between events cannot necessarily be calculated exactly without complete knowledge about other events going on in the world.

The relationships between two events in time can also be absolute or relative. Some relationships specify an exact amount of time that must come between the two events: ten minutes after the plastic has been poured, remove the widgets from their molds. More common are the simple relative relationships which just specify that one task must follow another. In between are tasks that are related by some interval: to ensure unwrinkled clothes remove them within an hour of the dryer finishing.

There have been several attempts to formalize temporal relationships so that inferences from time information could be made. There have been three basic approaches towards achieving this that have met with some success.

One of these formalizations has its origin in the *situation calculus* [McCarthy 58], [McCarthy 69] and has been expanded upon by McDermott (see [McDermott 82]). This system looks upon time as an infinite number of distinct *states*. A state is an instantaneous description of part or all of the universe. *Actions* and *events* are modeled in this system as operators for getting from one state to another.

Hayes created a logic where the entire set of effects of an action could be represented. The main component of his system was the *history*, a four dimensional space-time object that represented all the relevant parts of an event as it progressed through time. The ways in which different histories interact with one another, and the time over which a given history can propagate, are all determined by a set of axioms given in the first-order logic. Some of the axioms necessary for representing the domain of liquids is presented in [Hayes 84].

A temporal logic by Allen [Allen 83], [Allen 81] assumes that intervals are the basic unit of representing temporal information. Allen's system has thirteen relations that can hold between intervals. Rules on how these relationships can be combined and their transitive properties are also given. By assigning intervals to each event and action, and then enumerating all the relationships that are known to exist between various intervals, Allen's system allows additional relationships to be inferred.

The major drawback of Allen's system is that it can do only limited inferences involving metric information. In order to be able to propagate all the metric information that can be associated with an event intervals must be broken into their endpoints and operations done on those points. This combines the interval system with state-based information. Temporal data base systems such as [Dean 83] and [Smith 83] do just that.

Dean's style of temporal data base, called a *time-map*, is especially useful for storing temporal information for certain planning systems. The time-map is an elegant extension of the effects and preconditions network used in Nonlin. The time-map contains links, between the elements in the data base, that capture all of the ordering information contained in the partial-order of those elements. In this way the time-map can capture all of the information normally contained in the task network associated with least-commitment planners such as Nonlin. Additionally, the time-map can store the amount of time necessary to accomplish each action in the network. By *projecting* the effects associated with each action in the network, the time-map can be used to calculate the *persistence* of any fact in the data-base. For instance, the fact that a loaded gun is loaded will persist until the gun is unloaded or the gun is fired. Either of those two actions is said to have *clipped* the persistence of the

gun being loaded. A time-map can be used to detect some adverse plan interactions by noticing places in the data base where the persistence of one plan's prerequisite is clipped by some other plan.

A time-map can be used in a least-commitment planner to detect all of the potential conflicts that could be detected by Noah or Nonlin. Additionally, it can detect some conflicts involving metric time constraints. For example, if two tasks must be accomplished within three hours, each requires two hours to be executed, and they must be done sequentially, then the time-map can detect that a deadline violation will occur. However, a partially-ordered network like the time-map is still inadequate to detect conflicts that arise from the state-transition delays and effects between tasks. Only a complete ordering can reveal the results of the state-transitions and their effects on the rest of the plan.

1.5.3 The Scheduling Problem

The ordering of tasks and the assignment of resources to those tasks can form many different types of scheduling problems. Solutions to some special scheduling problems have been developed, but for many problems, the only algorithms that guarantee a solution have a computational complexity that is exponential. Task scheduling, where the constraints can vary depending on the order of the tasks, is one of the scheduling problems that is believed to be NP-hard.

This section briefly reviews some of the problems and research done on scheduling. This work has been performed for the most part by people in computer science and operations research. Because of this, the bulk of the research has been on scheduling problems that arise in computer systems and in factory setups. It is possible to move the algorithms and heuristics developed in one domain to another, but as has been pointed out in [Lenstra 77], there are only slight differences that separate problems with polynomial-time solutions from those that are NP-hard.

Traditional scheduling problems can be classified on the basis of three aspects of the problem [Graham 77]:

1. The arrangement of processors to tasks
2. The task characteristics
3. The optimality criterion.

The first of these criterion, the allocation of processors to tasks, is used to discriminate between tasks that can be multi-processed, must be done serially, or must be multi-processed is some special order. If a task must be done on several processors then the scheduling problem is one of the following:

An Open Shop exists when the steps must be done on various prespecified machines, but the steps can be done in any order.

A Flow Shop is like an open shop but the job steps must be done in a specific total order.

A Job Shop scheduling problem is a more specific classification of an open shop. In a job shop no two adjoining steps in a task may require the same processor.

The task characteristics also go into classifying the type of scheduling problem. These characteristics are:

Preemption: can a task be broken up into parts that may be executed noncontiguously?

Resources: does the task have any resource requirements? If it does, does the task need exclusive rights to the resources?

Precedence: does the task have any prerequisites and/or is the task a prerequisite for any other task?

Delays: what is the earliest time that the task can be executed? Is that time an absolute time, or dependent on when its prerequisites are to be executed?

Deadlines: what is the latest this job can be started? Is that time absolute or relative? What penalty is incurred for violating the deadline?

Execution Times: how well is the execution time of the task known?

The type of optimality being sought also effects the classification of the problem. Two possible optimization criteria are minimizing a schedule's *completion time* and minimizing the total cost of the schedule's *deadline overruns*. These criteria can be combined to form many different classes of scheduling problems. Some of these problems have been solved (i.e. polynomial time solutions have been found). Others are known to be NP-hard, while still others have a computational complexity that has yet to be determined [Lenstra 77].

The most researched type of scheduling optimality is overall completion time. A common problem of this type is to minimize the total amount of time needed to execute a set of processes on a computer system. [Kafura 77] is a good example of a scheduling system for a multi-processor computer. In this system a task consists of an amount of contiguous CPU time required, and a specific amount of memory needed during that processing.

The processors each have their own independent amount of memory. Heuristics for assigning tasks to processors allow the system to produce good schedules in a reasonable time. [Ramamritham 84] presents a dynamic system for handling tasks with deadlines on a distributed computing system. The environment of a distributed system introduces problems of synchronization not encountered in other domains. The system handles these problems in a dynamic way; the task is passed to a processor which has indicated it could handle the task; if the situation has changed by the time the task arrives the task is passed off to the next processor which indicates a willingness to take it on.

Another area where scheduling has been extensively studied is in vehicle routing (see [Fisher 76] for an example of vehicle routing, and a solution). A problem in vehicle routing is specified by the number of vehicles in the fleet, the number of customers, the capacity of each vehicle, the size of each order, and the cost of traveling between any two customers. The general way in which these problems are approached is to first assign customers to each trucks, and then perform a traveling salesman approximation over the customers that each truck must serve.

Perhaps the branch of scheduling research that most closely parallels that of task scheduling is a form of scheduling recently developed to handle the allocation of program steps to machine instructions in Very Large Instruction Word computers. VLIW computers have machine instructions that execute several operations in parallel. To make efficient use of these processors the language compilers must transform ordinary linear algorithms into highly parallel code. There are very specific constraints on what kinds of operations may go on in parallel in a VLIW machine so the compiler must be very selective in the way it rearranges the code. The code itself may contain constraints on the order in which its parts may be executed — results obtained in one step may be necessary before another step may be sent to the processor. The process of deciding on the order in which to execute the program steps is called *trace scheduling* [Ellis 85], [Fisher 81].

Trace scheduling deals with many of the same issues as task scheduling. Loops, somewhat similar to those in the robot-feedback domain, exist in programs and must be unrolled in order to add parallelism to the final code. Precedence restrictions exist in code not unlike those that are found in the task network through which the task scheduler must search.

The differences between trace scheduling and task scheduling arise from differences in their environments. In trace scheduling the environment is that of the VLIW machine. It is a fixed environment that can be much more predictable than the environments dealt with by robot planners. No matter what the task given to the trace scheduler it never has to

entertain the thought that a new capability has been added to the system. No amount of task reordering will allow the VLIW machine to perform three floating multiplies a cycle where before it could only do two. On the other hand, a robot's abilities might be affected by the order in which it does certain tasks. For example, if a robot is to pick up groceries and repair the car as errands, it will be able to do the groceries faster if it has already repaired the car.

Trace scheduling has other differences from task scheduling:

- A workable solution is known in advance
- No absolute deadlines are enforced on the scheduler's performance.

The program first given to the trace scheduler is a workable, though inefficient schedule. The trace scheduler's job is to find the potential parallelism in that program; parallelism that fits the VLIW architecture. Towards this end the trace scheduler should produce efficient traces. However, there are no absolute deadlines that the resulting trace must meet; the trace through the code should yield as efficient a schedule as possible, but there is no situation where the results are "not short enough." In task scheduling it is not known in advance whether a final solution even exists. This is mostly due to the efficiency constraints imposed on the problems handled by robot planners; constraints that dictate the minimum acceptable performance on any particular problem.

None of the scheduling problems discussed in this section capture the full complexity of that which is encountered when doing general-purpose task scheduling. Job-shop and machine scheduling problems contain precedence, deadline, and resource constraints. Routing problems contain resource constraints and differential ordering costs due to the different distances which a vehicle must travel depending on the order in which it makes its stops. The generalized planning problem contains all of these types of constraints and costs with some additional flexibilities: such as indefinitely repeated tasks and precedence constraints that can be altered by the ordering of the early part of the schedule. In task scheduling it is even possible that some tasks will not be included in the final schedule, depending on how earlier tasks are scheduled. As with the problems mentioned above, general-purpose task scheduling does not have a polynomial-time solution and must therefore be solved by heuristic means.

1.5.4 AI Planners That Do Scheduling

During the past few years task domains that require a realistic representation of time have been tackled by AI planning systems. These domains include airborne observatory flight planning [Gevarter 85], spacecraft operations control [Vere 83b], collision-free control of multiple robot manipulators [Roach 85], and various forms of factory and assembly control [Fox 85], [Cheeseman 84], [Fox 83]. These systems typically trace their origins either back to a Noah style planner, or some operations-research methodology of scheduling. This section will explore one example system from each of these categories.

Isis [Fox 83] produced job-shop schedules for a Westinghouse rotor factory. As described in the previous section, job-shop scheduling problems involve tasks being done on predetermined processors where no two adjoining steps are performed on the same processor. Additionally, all of the subtasks in the job that involve the same step must be completed before any subtask doing a different step is begun (i.e., if twenty rotors are being produced, then all twenty rotors must be turned on the lathe before any of them may be milled). Isis performed the scheduling by doing a heuristic search, both forwards and backwards, over the possible schedules.

When Isis is given a job to schedule its major problem is not to calculate the order in which to execute the various steps of the job. There are a very limited number of ways in which to produce a given product at a given factory. The scheduling decisions that Isis must deal with involve the assignment of specific machines (e.g., lathe-33) to accomplish the specific steps of the job, and also to make sure that the new job does not adversely affect any of the other jobs that are scheduled. The different jobs that Isis handles can all be considered to be independent of one another in most respects. The only interaction that the jobs have with one another is that they compete for the same factory resources. While the structure of this type of factory problem greatly limits the complexity of the problem from what it conceivably could be, the problem is still NP-hard.

A large amount of the work done by Isis involved the extraction and organization of the constraints that were relevant to the problem being worked on. Isis was not really designed as planning system; the constraints for the problem were therefore not attached to the steps of the problem but instead had to be extracted, for the most part, from the state description of the factory. The Isis system presented a representation and description for four classes of constraints: costs, physical limits, preconditions, and preference. The organization of these constraints made the task of constraint extraction and application more tractable.

The schedules developed by the Isis system are heavily dependent on having complete knowledge of all of the other events going on in the factory. One piece of knowledge that the system relies on for constraining its search space is the knowledge of what constraints are relevant to the search. For example, when Isis gets an order that is a *forced outage* then it knows that the due date is the major constraint. Specific strategies for dealing with that form of constraint could then be brought to bear towards finding the solution to problem.

In the factory situations in which Isis operates, state-transition delays are considered insignificant; the amount of time needed to move a turbine blade from one machine to another is a minute fraction of the amount of time needed to perform the operation on that machine. The state-transition delays are so small that Isis' representation of time is exact; the system does not model an interval of time as containing any uncertainty. The setup times necessary for various machines are small and similar so that they can be worked into the time Isis schedules for the operation of that machine. All of the time intervals dealt with by Isis are "exact." The times needed to complete specific production steps are assumed to be known precisely as are the times when a unit is due for delivery to the customer, or a delivery of supplies is due in at the factory. This is a situation that is found in few places outside of a well run factory.

Isis performed successfully in its domain, producing good schedules in a reasonable time. However, the assumptions that were made, in order to optimize its performance on job-shop problems, make Isis unsuitable for handling general task scheduling. In typical task scheduling situations:

- Uncertainty is common
- It is unknown which constraints will dominate a problem are unknown
- Some steps of the problem are poorly defined
- State-transition delays can be very significant
- There may be a large number of substantially different plans for performing any given task (see section 6.4).

Job-shop scheduling is much more concerned with the allocation of machine resources to various tasks than it is with the ordering of those tasks. These differences from the job-shop factory domain make the specific rating functions, heuristics, and search techniques used by Isis inadequate for solving problems in domains such as robot navigation.

New work on the Isis project [Smith 85] indicates that state-transition delays are starting to become a factor in their research. Transition delays appear in the job-shop domain as

setup times for a particular factory machine. For example, if it took half an hour to convert the lathe setup from that needed for an eighteen inch blade to the setups needed for twelve or twenty-four inch blades, and an hour was needed to change setups between the twelve and twenty-four inch blades, then the order in which the blades are run on the lathe will affect the total time required. To handle problems with order-affected setup times, while still maintaining the systems ability to allocate resources efficiently, the problem is decomposed from both job-based and resource-based perspectives. The task scheduler provides the machinery to do much of this work. The adaptive scoring techniques used by the task scheduler effectively examine the problem from different perspectives, choosing the dominant perspective for the particular part of the problem. The scheduler's role in a multiple perspective planner is discussed more fully in section 7.2.3.

Deviser (as described in [Vere 83b]) planned out the sequence of photographs and camera movements for the *Voyager* Spacecraft during its planetary flybys. Deviser was based heavily on the Noah and Nonlin systems, but could deal with temporal constraints in ways that were missing from both those systems. Deviser had a simple number-line representation of time. For each task given the system, an *ideal* time was specified — a time where the system should try to schedule that task. Deadlines and minimum start times could also be specified for tasks. The system would then proceed to expand tasks in a Noah-like way, ordering tasks only when an obvious conflict was encountered. If a specific ordering of two tasks was not sufficient to resolve the conflict then the system would backtrack to its last plan expansion or ordering choice point, and restart planning from there.

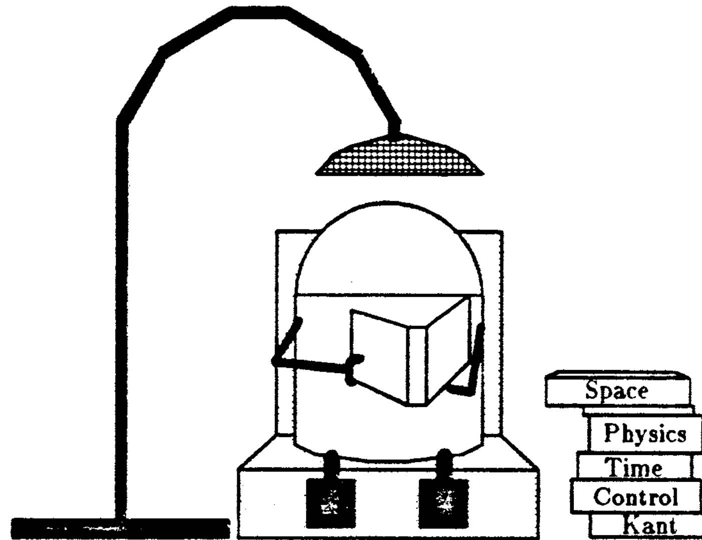
When expansion was complete the system would attempt to make a total ordering out of the partial ordering it had created. The ordering process was controlled by the ideal times. After an ordering was constructed, task times would be figured in and their effects propagated. Should any conflicts or deadline violations occur as a result, the system would backtrack and try a different ordering. Deviser would always find a solution, if one existed, but a tremendous amount of backtracking was sometimes done in the process.

The amount of backtracking performed by Deviser is due in part to a lack of communication between the scheduling and planning processes. In Deviser, initial plan choices and plan ordering are done without any consideration of how long the tasks will take to execute. The amount of time necessary to perform any state-transition tasks (such as turning the spacecraft's camera platform) are also ignored during the initial planning. The only guidance given the planner in the area of temporal ordering comes from each plan's ideal execution time. It is quite possible that these times will bear no relation to a feasible schedule. In such cases the planner will probably end up backtracking through an appreciable

amount of its exponential search space.

An attempt has been made to avoid this situation in the task scheduler described in this research. The task scheduler used here does initial ordering, calculates the effects of execution time and state-transition delays, and works in its version of ideal start time all in one search operation. This search can be interleaved with the task selection (and used to help guide that selection) of a least-commitment planner.

Chapter 2



Space, Time, and Representation

2.1 Introduction

This chapter presents the knowledge requirements and representation necessary for performing automatic robot planning in the robot-feedback domain. This domain contains loops, temporal and spatial constraints, and resource restrictions — perfect conditions for the production of adverse context-dependent task interactions. Because of these interactions traditional planning techniques will not suffice — it is necessary to perform some amount of task scheduling during the construction of plans in this domain. This chapter presents some of the details of this domain and of the representations used by the Bumpers system in order to pass information inherent in the domain to the task scheduler.

The robot domain in which Bumpers produces its plans consists of a simple mobile robot and a set of interconnecting hallways such as could be found in any school or office building. The tasks that are given to the problem solver are of the form `(move A B)`, where `A` and `B` are locations designated in a map in Bumpers memory. The system then attempts to construct a program that will guide the robot from its starting position to the designated destination.

The programs are constructed by Bumpers prior to the robot starting out on its task. Therefore each step in the program assumes that the previous steps have been carried out successfully. This would be a reckless assumption (considering the reliability of the robot being modeled, see section 2.2) if steps were not included in the program to help ensure satisfactory execution. Towards this end the programs created are heavily feedback driven.

Feedback, of course, involves getting information about the world around the robot. In order to get information the robot must have some sensors and some way to decode the information coming from them. In the mobile-robot domain there are several types of information the robot needs to know:

1. The orientation of the robot's steering wheel
2. The orientation of the robot's head
3. How far the robot has traveled
4. The amount of deviation from the robot's predetermined path
5. The robot's current position.

The first two of the above are completely internal to the robot; the robot has special hardware for deriving this information (see section 2.2). The last three involve relating the robot's actions to the affects that they have on the robot and its surroundings. How these pieces of information can be calculated, and how the availability of this information effects the planning process, are the subjects that make up the bulk of this chapter. At the end of this chapter is a description of the task formalism used by the Bumpers system. The formalism is designed to contain all of the information necessary to drive a scheduler based planner.

2.2 The Mobile Robot

The robot used in this domain is modeled after the Heathkit *Hero-1* robot. The *Hero-1* robot is a small, relatively inexpensive mobile robot that was the prototype in most aspects to the entire class of personal robots that are currently available. This class of robot has many features not commonly associated with simulated robots or other research robots. Most notable among these are:

- low sensor bandwidth

- low mechanical accuracy

The most advanced robots currently being made have a very limited amount of sensing abilities when compared to a person. Their mechanical accuracy, no matter how good, can be trivially overcome by a bump in the floor or a change in surface friction. The robots of today are able to perform their astounding feats only because they attempt their tasks in highly controlled environments. By further controlling the environment in which the *Hero-1* operates, we are able to mimick the performance of a more sophisticated robot operating under harsher conditions. Controlled environments are necessary for doing research; there are too many unsolved problems that would have to be solved simultaneously in order for a robot to function in the real world. Since the purpose of this research is to study the coordination of multiple time-dependent tasks (such as sensor and effector tasks in a feedback loop), a domain in which such tasks commonly occur is desirable. For this research, the maneuvering of an unsophisticated robot in a controlled environment has been selected as the domain. Such a domain allows the study of the coordination of time-dependent tasks without the complications that arise from trying to run a robot in a realistic environment.

2.2.1 The Robot's Effectors and Sources of Error

The Bumpers' robot has a tricycle wheel arrangement where the front wheel is both the drive and steering wheel. This wheel is equipped with a shaft encoder so that the number of rotations done by the wheel may be accurately recorded. However, the shaft encoder is only enabled when the drive motor is powered up; when the motor is disabled the counter stops, whether or not the wheel is still turning. The drive motor, like all of the motors on the robot, can be run at SLOW, MEDIUM, or FAST speed. Since the robot is fairly massive, and since there are no brakes on the robot, it does tend to coast a noticible amount after the drive has been shut down. The amount of coasting depends on the speed at which the robot was moving, and on the surface over which the robot travelwd. The coasting combined with the drive wheel's imperfect traction with the ground form the major sources of error to the calculation of the robot's travel distance.

The robot's drive wheel is steered through the use of a single stepper motor. There exists a counter in the robot's memory which is incremented or decremented accordingly every time the steering motor is pulsed. The counter is automatically zeroed whenever the wheel is turned to its leftmost stop. By initializing the wheel at its leftmost point, the counter can be used to keep track of the position of the wheel. Unfortunately, the motor

does not necessarily turn one step for each pulse it receives. Friction between the steering wheel and the surface it is traversing can sometimes cause the motor not to move when pulsed. On the other hand, torques due to the robot's own movements can sometimes cause the steering wheel to be turned without any signals being sent to it. For these reasons the steering wheel position counter does not always accurately represent the true orientation of the steering wheel.

The robot is also equipped with a *head*. The head serves as a platform for the robot's various sensors, the most important of which is the ultrasonic distance sensor. The head can be turned from side to side independently of the orientation of the main body of the robot. A counter in the robot's memory is dedicated to keeping track of the head's orientation in a manner similar to that employed for the steering wheel. The counter is initialized when the head is rotated to its full left position. Since the head moves more slowly than the steering wheel, and since there are fewer forces acting on it, the head's orientation relative to the body of the robot is more accurately known than the robot's other aspects.

2.2.2 The Ultrasonic Sensor

A sonar sensor is made of two parts: an emitter and a receiver. The emitter sends out a coded pulse of ultrasound in a semifocused beam. At the same time that the emitter releases its pulse, a clock is started. The first reflection of the pulse that is detected by the receiver stops the clock. The distance to the object is calculated by using the speed of sound and one half the clock time.

Since the sonar on the robot used in this domain is always aimed in the plane of the floor, a tight constraint on the maximum distance at which the sonar can sense an object may be defined. This distance, D_{max}, is dependent on the height H above the floor at which the sensor is mounted. The sonar *beam* is actually a cone with the point located at the emitter and an angular dispersion of 2ϕ. Thus, at a distance of $\frac{H}{\sin\phi}$ from the emitter the cone of ultrasound will intersect the floor (see figure 2.1) and be and possibly be reflected back to the receiver.[1] Therefore:

$$D_{max} = \frac{H}{\sin\phi}$$

[1]Whether or not the sonar is actually reflected back to the sensor depends on the surface qualities of the floor. A rough surface such as a thin carpet will often send a reflection back, while a smooth tile surface may just reflect the beam at an angle equal to the angle of incidence.

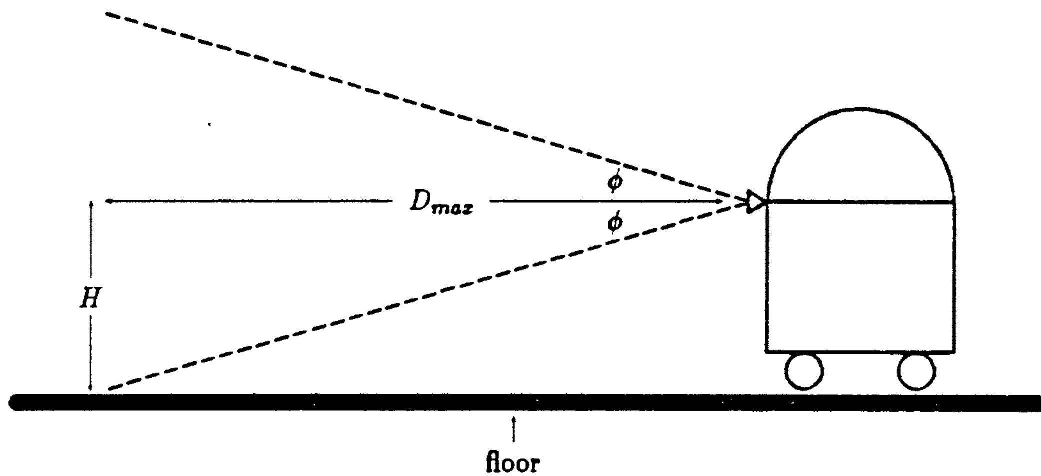


Figure 2.1: Sonar Dispersion & Definition of D_{max}

Because the sensing cone of the ultrasonic sensor expands quickly over the distance between the sensor and the object being sensed, certain constraints must be placed on the internal map of the world, and those objects being represented in that map.

1. Any significant object in the world (i.e. any object that could act as a barrier or a reference point to the robot) must be of a height greater or equal to the height H of the sensor on the robot.
2. The projection in the plane of the floor of any object must be identical to the contour of that object at height H.
3. Objects in the world are represented in the map as polygons with a grain-size[2] less than 2ϵ from the contour of the object at height H, where ϵ is the linear accuracy of the sensor.
4. Every side of an object is represented explicitly in the map.

The first restriction above is necessary to insure that any object that is visible to the sensor from far away is also visible close up (see figure 2.2).

Restriction (2) is necessary to assure that the apparent distance to an object changes in a manner directly related to the actual changes in distance to the object. If the closest

[2]Grain size is used as defined in [Davis 84] and is the upper bound of the difference between the representation and the object it is representing.

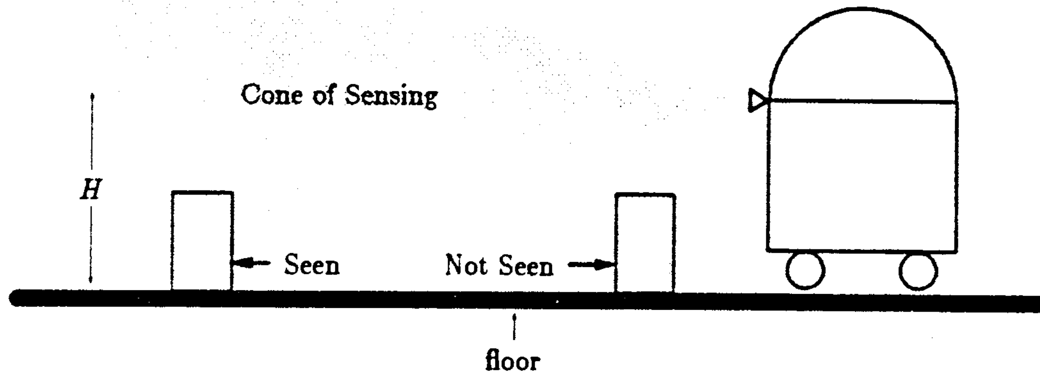


Figure 2.2: Why Objects in the Map Must be at least of Height H

point of the object to the sensor is not at height H (see figure 2.3) then at small distances the object will appear farther from the sensor than it actually is.

The third restriction specifies that the map will be accurate to within the sensitivity of the sensor. The last restriction guarantees that anything the sensor sees will be represented in the map.

The last restriction

The abilities of the robot to sense its environment are the determining factors in defining how that environment will be represented internally to the robot. There is no sense in giving a robot, equipped only with an ultrasonic sensor, a map that contains references to the colors of objects. Such information is useless in aiding the robot to navigate. Similarily, a map that is completly accurate is of no more help to an error prone robot then a map that contains some uncertainty. The accuracy will be lost by the robot in its interpretation of its surroundings by its sensors. The next section presents a representation scheme geared to a robot with the abilities and limitations that have just been presented.

2.3 The Spatial Representation

In the domain of mobile robots, space, and how it is represented, plays a pivotal role in any problem solution. The spatial representation system serves three purposes:

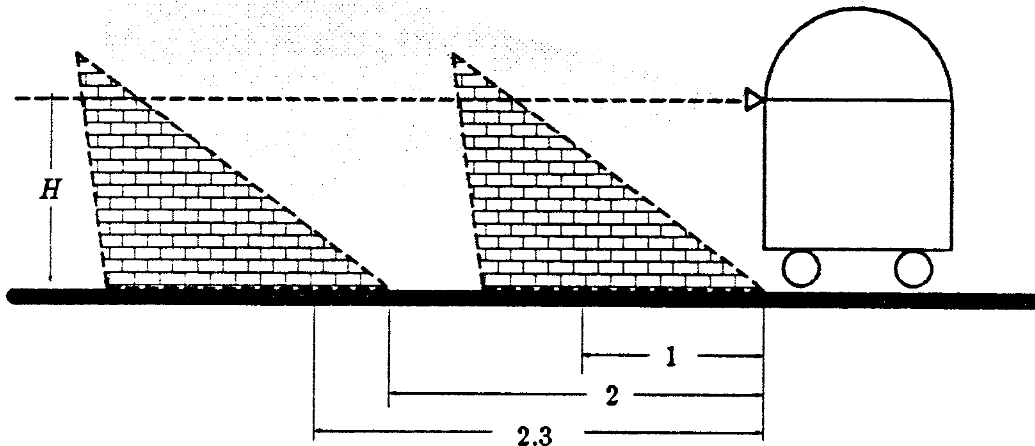


Figure 2.3: The Importance of the Outer Hull Being at the Sensor Height

1. It provides a framework for incorporating newly discovered information about the robot's environment.
2. It provides the necessary information to do route planning.
3. It gives information for monitoring the position of the robot during task execution.

Over the years several representation systems have been developed to tackle each of the above problems, although most of these systems have been designed to attack only one or two of them. Only a few have been designed to work with a real mobile robot.

The representation system to be presented allows easy calculation of the robot's position using only its onboard sensors. Methods for plotting navigable routes and monitoring their traversal are also given. The emphasis of this system is not to find the shortest route, nor to calculate the robot's position to the highest accuracy. Rather, this system is designed to find the *easiest* route for the robot to follow and to provide corrective information should the robot's path deviate from the precalculated one.

This sections presents a representation system that is suitable for the limited abilities of the Bumpers system robot. The system is designed to aid in the planning and execution of simple navigation tasks in a world of corridors, obstacles, and doorways.

For the purposes of this research the world consists of a flat open plane on which vertical walls and obstacles are placed in a manner suitable to the constraints specified in section 2.2.2. The walls are also flat, but can be joined with other walls to form rooms and obstacles of arbitrary polygonal shape. Since the robot is limited to motion on the plane of the floor, the projection of walls and obstacles onto the plane of the floor captures all the relevant information about the world.

The basic unit of the spatial representation system is the *map*. A map is made up of linked *regions*. Regions have a local coordinate frame. Walls and obstacles that exist in the area represented by the region are themselves represented by line segments whose endpoints' positions are designated by coordinates in the frame of the region. The borders of the region are marked with labels that specify the adjoining regions.

A distance sensor such as a sonar reports the distance to the nearest object that occurs inside its sensory cone. The sensory cone has a dispersion angle ϕ and a length D_{max} (see figure 2.1). A map is broken into regions by labeling every point within D_{max} of a wall with the name of the wall. All contiguous areas with the same markings are in the same region (see figure 2.4).

Regions can be any of four types: zero-F, one-F, two-F, or three-F. A floor-dwelling mobile robot has three degrees of freedom: the two planar dimensions and the robot's orientation. A type designation of j-F means that the sensor can be used to eliminate j degrees of freedom. So a zero-F region consists of an empty space that is at least D_{max} in any direction from the nearest landmark. Neither the robot's position or orientation can accurately be determined.

A two-F region can be formed by a single-sided large wall. The area within D_{max} of the wall and at least D_{max} away from each corner is a two-F region because the robot would be able to calculate its orientation and distance from the wall, but not its position along the wall. The areas within D_{max} of the corners are three-F regions since the robot's position and orientation can be fixed absolutely within the frame of the region.

In a one-F region the robot's exact position can only be determined if two of its three coordinates (x, y, θ) are known apriori. A one-F region exists when there are two or more symmetrical positions where the robot might be located. For example, if the robot is in a hallway with two smooth parallel walls of a length greater than $2D_{max}$ its position can only be located relative to the walls, but it cannot tell which wall is which, and therefore which of two possible orientations it has. If a one-F region is entered from a two-F or three-F region then the orientation can be kept track of as long as the robot takes measurements

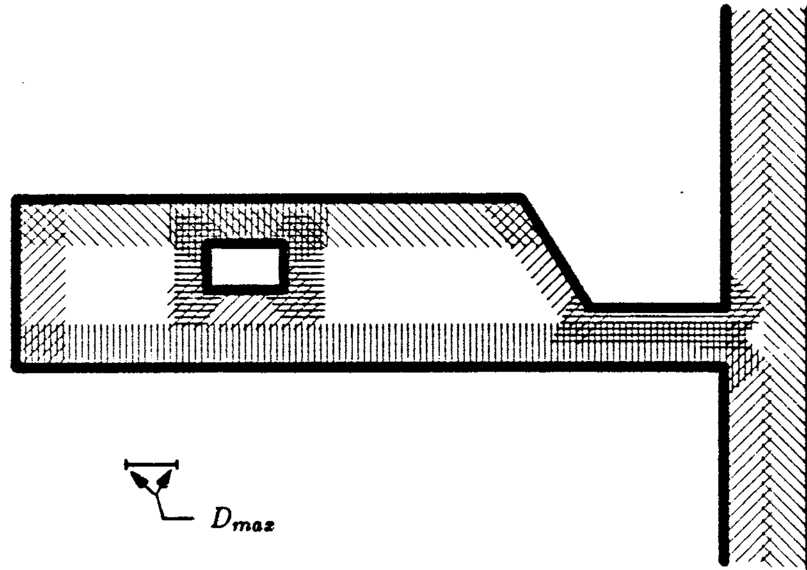


Figure 2.4: Dividing a Map into Regions

often enough to keep its orientation errors less than one half the smallest change between the region's possible orientations (e.g. in the one-F regions in figure 2.5, the robot must sample often enough to keep its errors below 90°, since there are two possible orientations 180° apart)

Each region has its own coordinate frame. A three-F region has exact positions for each measurement in it. Two-F and one-F regions use fuzzy numbers in one or more of their dimensions. The range of the fuzziness depends on the sensors used to determine it; in this research the sensors are comprised of the robot's odometer and sonar.

Regions are made up of a set of *edges*. Each edge is represented by a pair of endpoints whose Cartesian coordinates are specified in the frame of reference of the particular region. Zero-F regions have a coordinate frame, but do not have any edges. Each zero-F region contains the approximate coordinates of the nearest edges of each region which borders it.

The relative positions of features in two different regions cannot be known with great precision. The more zero-F, one-F, and two-F regions on the path between the regions in question, the less accuracy with which the two regions may be related. However, it is possible to get an approximate idea of the distance that must be traveled in order to go from one place to another. This is accomplished by adding up the tour distance for each

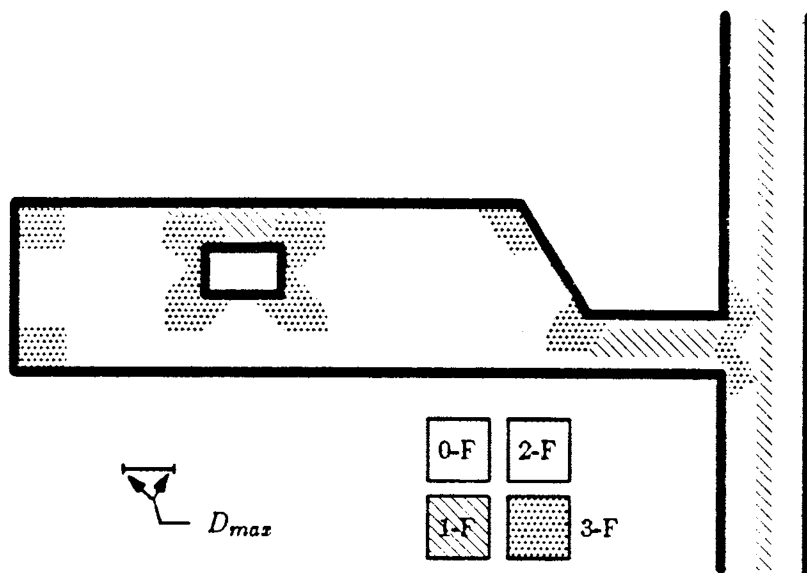


Figure 2.5: Typing the Regions in a Map

region that must be traversed. A lower bound is found by adding up all the lower bounds for each of the one-F and two-F regions, along with the distances from the three-F regions. A similar process is done to find the upper bound. But this does not help in relating position and orientation information from one region to another. The propagation and assimilation of new fuzzy information into the map are difficult processes but have been covered in detail in [Davis 84]. Neither process is necessary for the development of the navigation plans that the Bumpers system is designed to create. For the purposes of the planning process it is assumed that the maps given to the system are fully complete. No new information can be derived from the robot's sensors as it roams the area specified by the map.

2.4 Robot Positioning

One of the most important actions that a robot has to carry out while traveling to its destination is the determination of some of its intermediate positions. These positions can be relative to a wall, a specific landmark (e.g., a series of closely connected walls), or the robot's absolute position as related to its internal map of the area. Each of these positionings must be accomplished using only the robot's onboard sensors. Each of these types of

positioning rely on somewhat different techniques which are decided upon depending on the information available to the robot.

The type of region that the robot is in determines the amount of positional information that can be calculated. If the robot is in a zero-F region then the only positional information available would be extrapolations from the last known position, based on the robot's ability to do dead reckoning.

If the robot is known to be in a region that is one-F or greater, then positional information can be found using the systems described in [Miller 84] and [Drumheller 84]. The systems take several sensor readings and perform a heuristic search over the tree of possible matches between the observations and the edges in the map (see figure 2.6). Once a consistent matching between the observations and map edges has been found, a simple geometric construction is done to pinpoint the robot's location and orientation. While a tree of m edges and k observations has m^k possible matchings, the various heuristics used by the search algorithms usually allow the algorithms to find the robot's position in time $O(m^2)$.

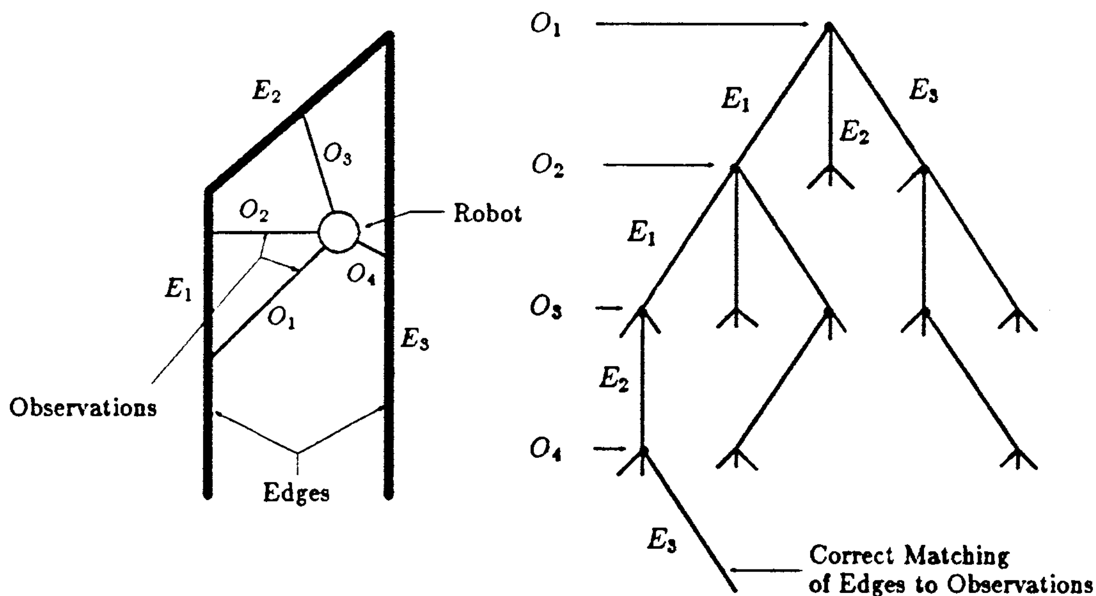


Figure 2.6: The Tree of Observations and Edge Matches

This performance could further be improved if the system could know which map edge the robot was closest to. With that information the positioning system could rate the partial matchings according to how consistent they were with that information, i.e. a matching that had the nearest observation matched to the wall known to be closest to the robot would be pursued before a matching that had that observation matched to some other wall.

In the positioning systems mentioned above, most of the heuristics are not really effective during the first two levels of of the matching tree; thus their m^2 performance. If the first observation could be assigned to a single map edge with a very high probability of being correct, the performance of the system would become approximately linear in the size of the map.

One way to do this would be to divide the regions of the map into areas such that every point in a specific area was closer to one specific edge in the region than any other edge (see figure 2.7). Such a construction is based on the standard Voronoi diagram [Shamos 75]. Algorithms that can find the Voronoi diagram of an arbitrary polygon in $O(m \log m)$ time were developed by Kirkpatrick in [Kirkpatrick 79].

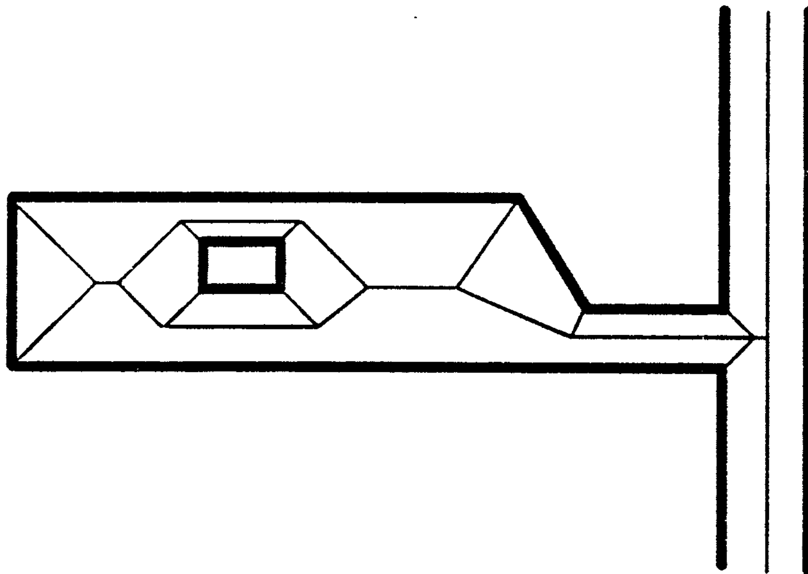


Figure 2.7: Voronoi Diagram of the Example Map

As the robot moves about the world, it keeps track of which *V-area* (Voronoi diagram

area) it is in. Whenever an exact positioning of the robot is needed, a rough sonar scan[3] is made of the robot's surroundings. The shortest observation is then assigned to the map edge associated with that V-area. This gives the positioning algorithm a rough orientation with which to start pruning the search space. The rest of the positioning process is carried out in the normal manner.

The V-areas also allow the robot's distance from, and orientation to, a specific wall to be trivially calculated in many instances. If the region the robot is in is convex then the shortest observation taken in a fine scan[4] of the robot's surroundings will be the observation that is orthogonal to the map wall associated with the V-area where the robot is located. (see figure 2.8). Such a check is very useful for monitoring the robot's progress as it moves about.

2.5 The Route Planner

Route planning is a vital function in accomplishing most mobile robot tasks. It is therefore not surprising that a large amount of work has been done on the subject. The route planning method presented below is based heavily on previous systems created by other researchers. However, the other systems tried to find the shortest or safest paths, although these are often the most difficult for a robot to follow. A strategy employed by many systems is to keep as far away from obstacles as possible. Other systems plot the most direct course possible, barely skimming past obstacles. Unfortunately, both strategies often produce routes that require large amounts of dead reckoning (see figure 2.9). The route planner used by Bumpers plans out routes that are within the Bumpers robot's abilities to traverse.

2.5.1 Finding the Easiest Route

The Voronoi diagram of a polygon is a made up of line segments that are equidistant between their associated edges of the polygon (see figure 2.7). Therefore, a path made up of the segments of the Voronoi diagram will stay as far away from the walls of a polygon

[3] A rough sonar scan is done by panning the sensor in a full circle, collecting four to eight approximately evenly spaced readings.

[4] A fine scan is when the sensor is panned in a full circle with observations being taken every few degrees. Since the time consuming part of a scan is the aiming of the sensor, fine and rough scans require approximately the same amount of time to execute.

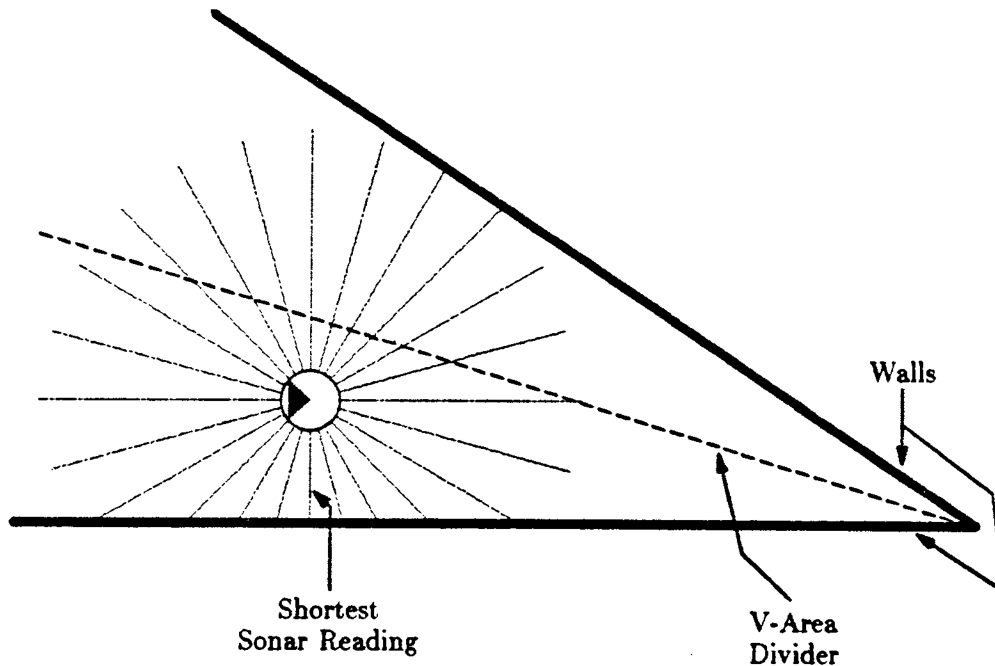


Figure 2.8: A Quick Check of Orientation and Distance to the Nearest Wall

as is possible. Because of this property, Voronoi diagrams have been used as the basis for several route planning systems: [O'Rourke 84], [Brooks 82b], [Rowat 79], [O'Dunlaing 83].

Unfortunately, routes from such a system would eliminate any advantage received from the modifications to the positioning algorithm described in the previous section. The advantage that the Voronoi diagram brought to the positioning algorithm was based on the system knowing what V-area the robot was in *before* the positioning process was started. A route made up of the Voronoi diagram segments would keep the robot on the borders between the V-areas.

What is desired is a route that travels down the centers of the appropriate V-areas, but which areas are these? The ease of navigation for a specific V-area can be determined by the type of region that contains it. Zero-F regions are very difficult to travel through because they rely on dead reckoning. Three-F regions contain ample information for execution monitoring, but tend to be more crowded than other regions. They therefore require more

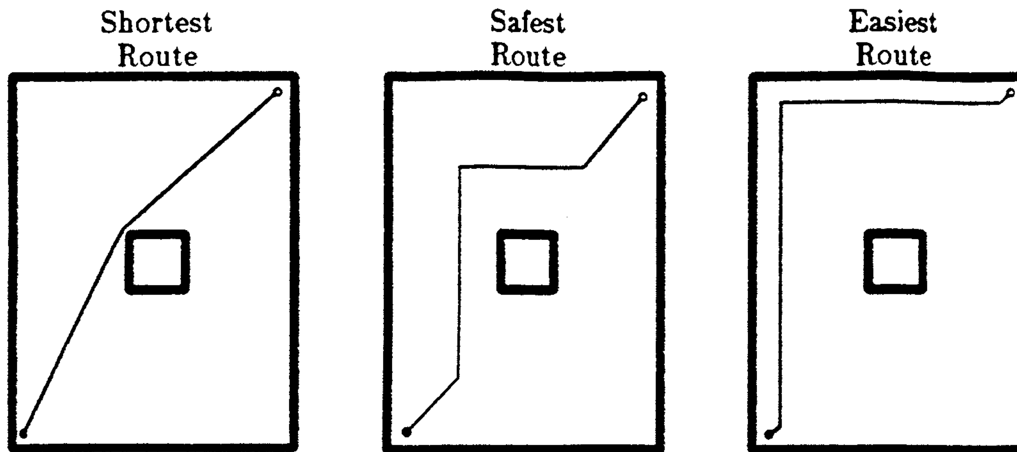


Figure 2.9: Navigating Across a Gymnasium with a Table in the Center

care than other regions. One-F and Two-F regions can cover large distances with continuous low level monitoring. If the destination lies along the path of the one-F or two-F region then large distances can be easily and reliably traversed.

The general rules for finding an easy-to-navigate route are:

1. Traverse a two-F or three-F region as early as possible in the route. This allows the robot to orient itself early on, making it possible for the robot to traverse a one-F region without ambiguity.
2. Minimize the total number of regions to be traversed
3. Travel along one-F and two-F regions when possible to get the most distance for the fewest regions.
4. Do not traverse zero-F regions.[5]
5. Traverse a three-F region as close to the destination as possible in order to allow a full positioning of the robot.

If the projection of the robot into the plane of the map is circular, then the easiest-to-follow path can be found by the following algorithm:

[5]Zero-F regions can be traversed by robots with limited dead reckoning abilities (see [Miller 85b] for a discussion of this), but the algorithms for crossing them are feedback free, and therefore do not cause any interesting plan interactions.

1. Grow the walls and obstacles in the map by an amount R which is equal to the radius of the robot. This allows the robot to be considered as a point for the remainder of the route planning process.
2. Calculate a conduit, [McDermott 84], made up of connecting *V-regions* (the areas found from overlaying the V-areas onto the regionalized map, as shown in Figure 2.10) that contain the starting and destination points. This can be found by doing a heuristic search with a rating function based on the rules in the paragraph above.
3. Calculate a centerline path through the conduit (see Figure 2.11).
4. Plot a direct course from the starting point of the robot to the nearest point on the centerline path. Do the same for the destination point.
5. Annotate each part of the route with the V-area it traverses.

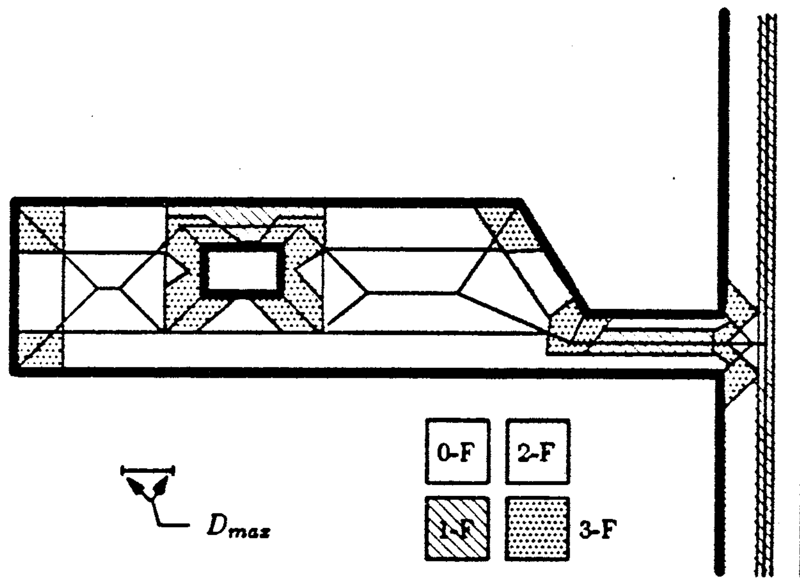


Figure 2.10: V-Regions, The Combined Voronoi Diagram and Regionalized Map

2.6 Spatial Information and Planning

The purpose of using a mobile robot in this research is to provide a planning domain that involves time-dependent feedback-driven tasks. The purpose of the spatial representation that has been presented above is to give the planner access to the information it needs

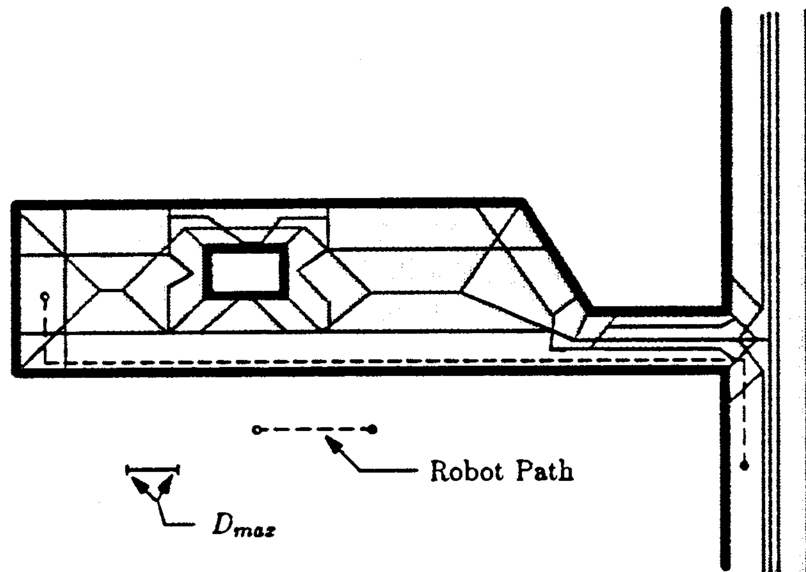


Figure 2.11: The Easiest to Navigate Route

to choose the appropriate feedback strategy for any given task. There are three decisions the planner must be able to make that are based on information contained in the spatial representation. They are:

1. What is the proper observation strategy; should the robot be following a wall, tracking a landmark, etc.
2. What should be observed
3. How often must the observations be made

This information can be derived for each part of the route the robot is to traverse by examining the region, V-area, and adjoining map edges.

2.6.1 Choosing an Observation Strategy

There are only a few types of general activities necessary for a robot to do in order to move from one place to another. The specifics, and how the activites interact with one another change from task to task, but the basic activities are very limited. With the mobile robot and representation system that have already been presented, there are just

five basic activities that a robot has to do in order to move from any one place in the world, to another. They are:

1. Follow a wall
2. Find a landmark
3. Track a landmark
4. Orient itself
5. Position and orient itself

The five activites above are sufficient for a robot to accomplish all of its movement tasks. Following a wall is used to traverse one-F and two-F regions. Finding a landmark is an ability used by the robot to keep track of what V-region the robot is in at any moment. Tracking a landmark is used for maneuvering through three-F regions. In crowded regions it is necessary for the robot to keep an eye out for landmarks in order to make sure that the robot does not vary from its desired course. Strategies for orienting the robot (see Figure 2.8) and doing a complete positioning have already been discussed in section 2.4.

The exact strategy used by the Bumpers robot in following a wall is dependent on the V-region being traversed by the robot, the robot's speed, and the robot's probable sideways drift (the distance per distance traveled that the robot might drift to the left or right of center). The basic strategy is for the robot to look at an angle 45° forward of the wall (see Figure 2.12). Since the robot views its world through a distance center, orienting the sensor directly at the wall (90° to the direction of travel) would make it impossible for the robot to tell what direction it was deviating in should the robot turn slightly towards, or away from, the wall. With the sensor initially oriented directly towards the wall, a deviation in either direction would cause the sensor to report an increased distance. By orienting the sensor at an angle of 45°, a drift away from the wall will give an exaggeratedly increased reading by the sensor. A slight turn towards the wall will cause the sensor to report a marked decrease in the distance to the wall. In this way the sensor will yield unambiguous data about the robot's position, relative to its starting distance from the wall, as well as the robot's directional trend at the moment of observation.

Since the robot's steering errors are basically random, an observation, and possibly a correction, must be performed by the robot every time the robot travels a critical distance. This distance is determined by the expected maximum error of the robot, the width of the V-region in which the robot is traveling, and the range of the distance sensor. A correction

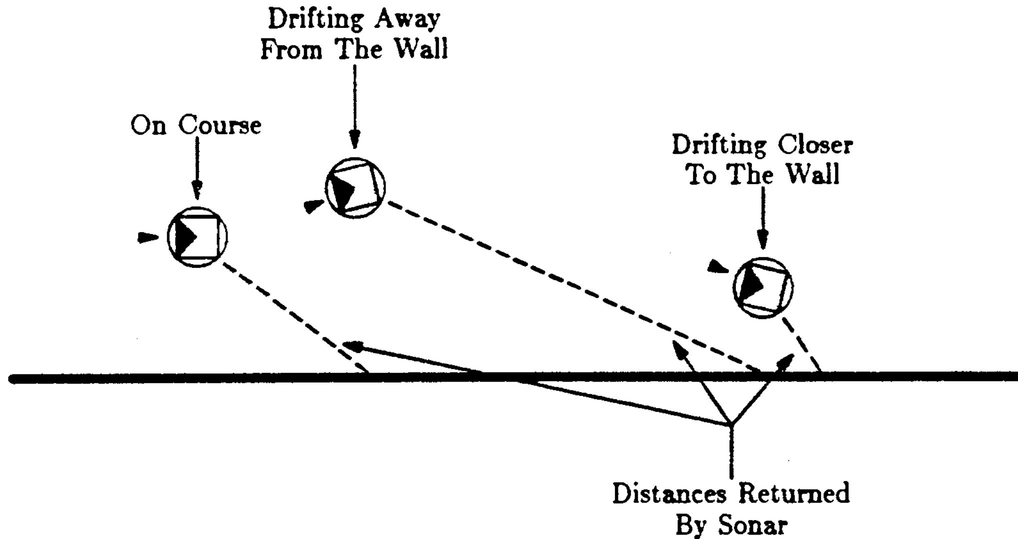


Figure 2.12: Sensor Direction for a Robot Following a Wall

must be made often enough to ensure that the robot does not leave the region and/or get out of sensing distance from the wall. Because the sensor is angled at 45° with respect to the wall, the amount of error acceptable to the robot is also limited to ensure that the robot does not get mis-oriented by more than 45°.

The reading from the robot's distance sensor is a function of the robot's orientation with respect to the wall, and the robot's distance from the wall, yet what part of the reading is due to which factor is not important. If the sonar reports the robot is in the correct position, but it is in reality too close to the wall, then the orientation of the robot is away from the wall; the robot will soon be in the desired position. The same is true if the robot is too far away from the wall. Should the sonar report a reading at or near D_{max}, then the robot is either at the far edge of the V-region or heading quickly in that direction. In either case the corrective strategy is the same.

A good correction strategy is to plot a course that will take the robot to the far side of its original course half as much as the robot appears to be from its course at the moment. Thus, if the robot strays away from the wall by two feet the correction should put the robot one foot closer to the wall then it belongs. The correction should be designed to get the robot to its *corrected* position by the time of the next observation. Once an error has occured (e.g., the robot has drifted to the right of its planned course) then the errors that

follow are often in the same direction, and of a similar magnitude.[6] Therefore, it is safer to try to slightly overcorrect the robot's position than it is to undercorrect it. If the errors are completely random then the robot is still usually safe since the correction should introduce a new error that is smaller in magnitude then the original one. If no further errors occur over the course of several observations then the robot's course will quickly converge on the desired one.

Finding a landmark with a sonar is a somewhat simpler affair. A landmark is a wall or intersection of walls that can be recognized by the sonar. If the robot is otherwise undirected, then the system should plot a course for the landmark, and the robot should observe straight ahead. When the sonar picks up something (i.e. the sonar reports a reading of less than D_{max}) then the landmark has been sighted. If the robot already has a course to follow then the robot should scan an area from the direction at which the landmark will be when the robot is D_{max} away from it to the direction that the landmark will be at when the robot is at its closest approach (see Figure 2.13).

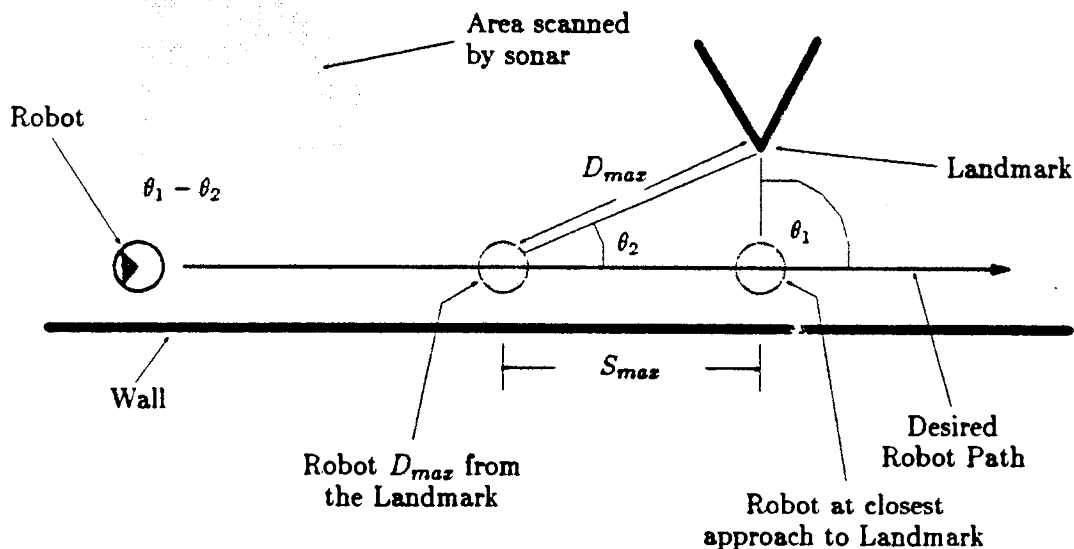


Figure 2.13: Sighting a Landmark Off to the Left

[6]Trends in a robot's steering errors are sometimes due to an unevenness of the surface over which the robot is traveling, or a bias in the robot's steering wheel.

The entire scan shown in Figure 2.13 must be performed at least once every time the robot traverses a distance equal to S_{max}. If scans are not completed at that rate then the robot might pass the landmark without ever sensing it. When the landmark does not lie on the robot's path, but the path goes past the landmark (as in Figure 2.13) then

$$S_{max} = D_{max} \sin \theta_2$$

When the landmark lies along the path $S_{max} = D_{max}$.

Once a landmark has been sighted, it is possible for the robot to use that landmark to travel any course desired — while the landmark is not occluded and within D_{max} of the robot. The robot picks a course, and heads off in the desired direction. The head is oriented so as to keep the sensor pointed at the landmark. The head position is calculated from the predicted course and the output of the robot's odometer (see Figure 2.14). By scanning the sonar slightly to each side, the exact position of the landmark, relative to the robot, can be kept track of. This information can be used to make steering corrections, in order to keep the robot on course. The amount of distance traveled between sightings depends on the expected steering errors of the robot and the size of the area being scanned.

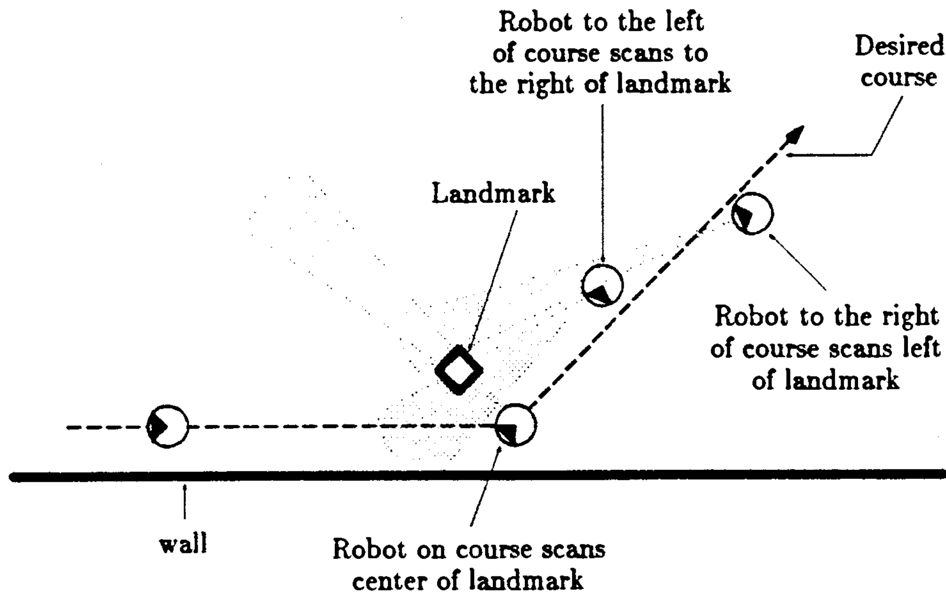


Figure 2.14: Tracking a Landmark to Follow an Arbitrary Course

2.7 The Task Specification Language

2.7.1 Introduction

A planner must have certain information about the problem it is to plan for in order for it to have any chance of deriving a solution. This information includes the task to be solved, the plan schemas the planner has available to it, and the constraints that exist on the problem and the schemas. Representing this information is the job of the task specification language. The language a planner uses to specify tasks gives insight into how much work the planner is going to do and what approach it is going to take. The language described below is designed for use with very simple mobile robots. It has facilities for describing feedback actions that need to be repeated indefinitely. Since the robot is operating in a somewhat realistic environment (rather than a flat, empty, infinite plane) the task specifications must be annotated with some information on how the robot is to maneuver through the world. These annotations can be used to keep the robot within certain physical bounds and alert the planner to any constraints which must be satisfied in order for the plan to be achieved.

During the planning process the plan schema for a task becomes instantiated; the instantiated schema is the plan for that task. A plan schema for stopping the robot near a wall might have the robot look at the wall and turn off its drive motor when the wall is within a certain distance. The plan for accomplishing this task would specify where the robot should look in order to see the wall, and how close it should get to the wall before stopping.

The task specifications contain all the information that is needed by the planner to produce a workable plan for the task specified. Figure 2.15 contains the full task specification needed to build the plan for the task of having our robot stop at a specified distance from a wall. The task specification is stored in the task network along with the other specifications for that task, and specifications for other tasks. As can be seen from the figure, there is more to a specification than just the plan schema. The other parts of the specification are needed for bookeeping and for the modeling of processes. These parts of the specification are used to predict interactions with other tasks. The full definition of a task specification (including optional fields marked with [ ]) is:

```
(index name parameters [declarations] [monitors]
      [constraints] [resources] plan-schema [utility])
```

```
(landmark-stop stop-near-wall (obs-direction min-stop max-stop)
    (WITH-MONITORS (dist-to-wall (SONAR)))
    (WITH-CONSTRAINTS
        (STATE-REQUIREMENT rl1s1 (= (STATE 'sensor) obs-direction)))
    (WITH-RESOURCES
        (sensor obs-direction discontinuous)
        (position *varied* overall)
        (drive on discontinuous)
        (drive off end))
    (PLAN-SCHEMA
        (REPEAT rl1
            (WHEN (CHANGE 'position (- max-stop min-stop)))
            (STEP rl1s1 (EVALUATE dist-to-wall))
            (WHILE (> dist-to-wall max-stop))
            (AFTER (stop-motor 'drive)))))
```

Figure 2.15: Task Specification for Stopping Near a Wall

This section will explain each part of the task specification in more detail. The task specification language presented here has been designed to be used by the Bumpers robot planning system in the domain of mobile-robot feedback-driven navigation. However, the language is applicable in other domains as well. As each part of the language is presented examples will be given from the robot feedback domain, or other relevant domains, as is appropriate. At the end of this chapter is a short discussion of how TSL compares to some of the previous languages used by automatic planners.

2.7.2 The Task Specification Index and Name

The first item in any task description is the *index*. The index serves as a pointer into the task network for finding appropriate plans for a given task. All specifications that contain plan schemas for accomplishing the same task will have the same index. For example, there are at least two strategies for navigating across a three-F region:

1. Orient the robot towards the destination, travel a few feet, do a complete positioning of the robot, reorient and repeat
2. Follow the walls along the perimeter of the region until the robot reaches the destination face of the region.

Each of these strategies would have its own task specification, but both would have the same index: `cross-3F-region`.

The *name* of a task specification is used to uniquely identify that particular specification. The two strategies presented above for crossing a three-F region would have the same task-specification index, but would have their own name. The name `stop-near-wall` is unique in the task network to the specification shown in Figure 2.15.

2.7.3 Parameters and Declared Variables

The *parameters* list contains variables whose values are set by the plan instantiator of the Bumpers system. The plan instantiator takes information provided by the Bumpers route planner, along with information gleaned from the original problem statement, and uses that information to fill in the values for the specification parameters. The values passed to the specification through the parameters list are used throughout the remainder of the task specification. In Figure 2.15 the parameter `obs-direction` specifies the direction that the sensor should be oriented for taking distance observations of the destination wall. The value of `obs-direction` is based on the wall's orientation with respect to the robot's path; it is calculated by the route planner.

Depending on the complexity of the calculations that must be done in the task specification, it may be desirable to store some values for later retrieval and use. Temporary variables for this purpose can be defined in the `WITH-DECLARATIONS` statement of the task specification. In the plan schema of Figure 2.15 the difference between the minimum and maximum acceptable stopping distances is calculated for every iteration of the repeat loop. The plan schema line:

```
(WHEN (CHANGE 'position (- max-stop min-stop)))
```

could have the subtraction replaced by a temporary variable whose value is that of the difference, since the acceptable stopping distances are defined by parameters that are fixed throughout the instantiation. The declaration statement:

```
(WITH-DECLARATIONS (distance-difference (- max-stop min-stop)))
```

creates the variable `distance-difference` whose initial value is the difference between the parameters `max-stop` and `min-stop`. `distance-difference` could be substituted into the appropriate line of the plan schema, providing a more efficient code for the schema. The `WITH-DECLARATIONS` statement can be used to create a variable for use in any part of the specification — wherever a temporary can be used to simplify the computation.

2.7.4 Monitor Functions

In the robot-feedback domain there are many quantities whose values can only be calculated at execution time. The *monitors* section of the task specification is used for describing these quantites. The `WITH-MONITORS` statement is used to specify these quantities and their defining functions. For example, in Figure 2.15 the statement:

```
(WITH-MONITORS (dist-to-wall (SONAR)))
```

means that the "current" value of `dist-to-wall` is found by evaluating the expression `(SONAR)`. For a monitored value, current refers to some point during actual execution by the robot. The value of `dist-to-wall` is made current by referencing it through the `EVALUATE` function. This function executes the right-hand part of the `WITH-MONITORS` statement (in this case, the expression `(SONAR)`) and stores it in the variable given in the left-hand part of the statement (in this example, `dist-to-wall`). The monitor variable can be referenced at any point in the task specification. If it is not wrapped inside of an `EVALUATE` then the value it contains will be that which was calculated from the last time it was `EVALUATE`d.

2.7.5 Resource Specification

In order to accomplish most tasks, something must be done that affects the state of the world. This often involes sensing, moving, or consuming something in the world or the robot. In the robot-feedback domain parts of the robot must be used to accomplish most tasks. The parts that will be used, the states in which they will be needed, and the times in which they will be in use are all specified in the `WITH-RESOURCES` portion of the task specification. The syntax for this part of the specification is:

`(WITH-RESOURCES -(`***resource state time*** `)- )`

where *resource* is the thing being used, ***state*** is the state the resource must be in (e.g., on, off, 42) in order for it to be useful to the task, and ***time*** is one of:

continuous — the resource will be in constant use during the execution of this plan (e.g., a space-suit is in continuous use during a space-walk).

discontinuous — the resource will be needed during one or more, not necessarily adjoining, intervals during the execution of the plan (e.g., a thesis advisor is used discontinuously by a graduate student during his thesis research).

start — the resource will be needed at the begining of the execution of the plan (e.g., the solid fuel boosters are a required resource during the start of each space shuttle mission).

end — the resource will be required at the conclusion of the plan's execution (e.g., the shuttle landing gear is needed at the end of each space shuttle mission).

overall — the resource will be used throughout the task, but not necessarily in the same state and not necessarily continuously (e.g., the robot's position is a resource that must be carefully controlled over all of the execution of a tightrope walking task).

The state that the resource must be in can be defined by a constant, parameter, or previously declared variable. In the statement:

```
(WITH-RESOURCES (sensor obs-direction discontinuous) ...  )
```

the use of the robot's sonar sensor has been reserved to be in the state designated by the parameter `obs-direction` at one or more times during the plan execution. The statement

```
(WITH-RESOURCES ...  (drive off end))
```

indicates that at the conclusion of the plan given in Figure 2.15, the robot's drive will be turned off.

```
(position *varied* overall)
```

indicates that the robot's position will be at various locations throughtout the execution of the plan. `*varied*` is a reserved word meaning that the state of the resource will change during the interval that the resource is required. The specification on the resource is made to ensure that no other task takes control of the resource during the interval it is needed. The next chapter presents more details on resources. Their role in planning and task scheduling is described in section 3.2.

2.7.6 Declaration of Scheduling Relations

Sometimes a plan is dependent on a factor in the state of the world that is not a resource under the planner's direct control. For example, the strategy for having a robot traverse a two-F region is to have the robot observe and maintain a constant distance from the wall that forms the region. The factor that controls how often the robot must observe the wall

is the robot's lateral drift. The lateral drift is not a factor under the robot's control. Unlike the robot's position, there are no commands that can be issued to change the robot's drift perpendicular to its direction of travel. However, the maximum lateral drift for any amount of travel can be predicted, and then checked by direct observation.

---

lateral-drift = 0.2(distance-traveled at (speed = fast))
+ 0.15(distance-traveled at (speed = medium))
+ 0.1(distance-traveled at (speed = slow))

Figure 2.16: Function for Calculating Maximum Lateral Drift

Figure 2.16 presents the function for calculating the maximum lateral drift for the Bumpers robot. Note that the amount of drift is dependent on the distance traveled **and** the speed at which the traveling is done. The speed at which the robot travels is not a resource that is specified in the task description; it is a state that a resource (the robot's drive) can exist in. In Figure 2.15 the state of the robot's drive is only constrained in that it must occasionally be **on** in order for the robot to successfully carry out its task. **on** does not describe the speed of the drive; that decision is left to the task scheduler, which decides on a speed that allows the task to be merged with any other duties that the robot must carry out. Deciding on a speed in advance could cut out many possible arrangements of the robot's activities. The plan is left flexible and the scheduler uses its larger view to decide on details such as motor speeds.

Plans that will be greatly affected by the decisions the scheduler will make (such as a plan for crossing a two-F region where the calculation of lateral drift depends on the drive speed assigned by the scheduler) need a method of letting the scheduler know what parts of the plan are affected by the various scheduling decisions. This need is fulfilled by the `WITH-RELATIONS` statement. This statement is a method for supplementing the planner's normal physics (see section 3.1.1) with a relationship that is relevant to one particular task description. It can be thought of as adding a custom resource or factor to the task description, and describing how that resource behaves. The syntax of the `WITH-RELATIONS` statement is:

(`WITH-RELATIONS` (*property delay-function*))

The *property* is the resource or factor that is being modeled (e.g., lateral drift). The *delay-function* provides the scheduler with the information needed to make scheduling de-

cisions based on the behaviour of the property. The delay function calculates the amount of time needed to change the state of the resource by some amount (see section 3.1.1). For our example this means that the delay-function would take the schedule of the robot's activities and determine when the lateral drift might exceed some predetermined maximum acceptable level. WITH-RELATIONS and delay functions will be discussed in more detail in the next chapter.

2.7.7 The Plan Schema

The *plan schema* is described in the PLAN-SCHEMA portion of the task description. It consists of a list of statements from the set:

1. STEP
2. PROG
3. REPEAT

The statements are combined to form the basic structure of the plan.

The format of the STEP statement is:

(STEP *key action*)

The key is some globally unique string with which the step can be referenced. The action refers to a primitive action which the robot is to perform. The STEP statement is the basic building block of the plan schema.

```
(PROG key
        plan-step-1
        ...
        plan-step-n  )
```

Figure 2.17: The Syntax of the Prog Statement

The format for PROG statements is given in Figure 2.17. The steps in a PROG statement are executed in the order in which they are listed. The steps of a PROG can be STEPs, PROGs, or REPEATs.

```
(REPEAT key
    [WHEN s]
    [TIMES n]
    [WHILE c]
    plan-step-1
    ...
    plan-step-n )
```

Figure 2.18: The Syntax of the Repeat Statement

The REPEAT statement has a slightly more complex construction as shown in Figure 2.18. The EVERY, TIMES, and WHILE substatements may appear before, after, or in the midst of the body of the REPEAT statement.

The s referred to in the WHEN substatement is some state that must be achieved before the next iteration of the REPEAT statement may be executed. The WHEN substatement is checked at the begining of every iteration of the REPEAT in which it appears. The WHEN substatement is used to space out the iterations of a REPEAT statement, i.e. the WHEN substatement controls the timing of a REPEAT loop.

Sometimes the state that must be achieved in a WHEN statement is not an *absolute* state (such as look straight ahead), but is instead a state expressed in terms relative to the state that must preceed it (e.g., look 5° to the left of where you were looking). In these *relative* state cases it is the change that is important. The CHANGE statement is used to express this change. The syntax of the CHANGE statement is:

(CHANGE *resource amount*)

The arguments to the statement are the resource whose state must be changed, and the amount by which it must change. The amount is a signed number. The task specification in Figure 2.15 contains the CHANGE statement:

```
(WHEN (CHANGE 'position (- max-stop min-stop)))
```

This statment means that before an iteration of the loop that contains it may be done, the position of the robot must be changed by an amount equal to

```
(- max-stop min-stop)
```

from the position of the robot during the previous iteration of the loop. The state given in the WHEN substatement is a goal that must be achieved before the next iteration of the loop may be performed. The robot must move the specified amount between each iteration of the loop.

The TIMES substatement informs the planner that the loop should be repeated n times. WHILE specifies the conditions c under which the loop should be continued. If c becomes false the loop is terminated at the point the WHILE occurs. There can be multiple WHILEs in a REPEAT statement. In REPEAT statements that have both WHILE and TIMES substatements the loop will terminate with whichever indicates to do so first.

2.7.8 Specifying Constraints

The *constraints* on a task specification are defined in the WITH-CONSTRAINTS portion of the specification. Constraints can come in many forms and serve varied purposes. Constraints are used for defining and placing restrictions on state transitions, declaring mandatory orderings, setting up deadlines, and guiding the choice of which plan schema to use in a particular situation.

For this research, six different types of constraints have been defined; all are specified in the WITH-CONSTRAINTS statement. The constraint types are:

- Ordering Constraints
- Preconditions
- Metric Time Constraints
- State Requirements
- Interval Time Relations
- Conditional Constraints

Ordering constraints specify the order that subtasks must be executed in. They can be specified in the WITH-CONSTRAINTS field using the PRECEDES and FOLLOWS predicates (e.g., (PRECEDES a b) or (FOLLOWS b a)), or they can be derived from the PROG field in the plan schema of the task specification (see section 2.7.7). Note that these predicates do not imply that the execution of b must immediately follow a, only that the execution of b comes sometime after that of a.

Preconditions are specified in the `WITH-CONSTRAINTS` field by the `TRUE` predicate. This predicate specifies that its argument is true at the start of the execution of the task in which it appears. The `TRUE` predicate is designed primarily to help choose between multiple plans for a single task. If there are several specifications for a given task, only those that have all their preconditions met should be considered. For example, if the specification in Figure 2.15 contained the precondition:

```
(TRUE (> 10 (abs (obs-direction))))
```

then that specification would only be used in situations where the task was to stop near a wall and the observation direction was within ten degrees of the direction of the robot's travel. The implication of this is that there exists another specification that works better at larger angles.

Metric time constraints can come in two forms:

1. Absolute
2. Relative

Absolute metric constraints specify wall clock times that certain things may or must be done by. They are expressed by the `DEADLINE` and `MAY-START` predicates. The constraint:

```
(DEADLINE stop-in-hall-132 1500)
```

means that task `stop-in-hall-132` must be done by time 1500; a deadline violation should result if the task is not done by that time.

The relative form uses the same predicates, but with an extra argument. This argument is the task to whose execution the metric value is added. For instance:

```
(MAY-START walk-across-kitchen-floor 10 wash-kitchen-floor)
```

means that the task to walk across the kitchen cannot take place till at least ten minutes after the floor has been washed. There are implicit ordering constraints in relative metric time constraints.

The `STATE-REQUIREMENT` statement specifies requirements that must be met before its first argument (part of the plan schema) may be placed onto the schedule.

```
(STATE-REQUIREMENT wash-kitchen-floor (= (STATE 'position) 'kitchen))
```

specifies that the robot must be in the kitchen in order to wash the kitchen floor. The second argument to a STATE-REQUIREMENT must be some factor that is knowable by the system at the time of task scheduling. Usually these are checks that are done to a task's *state object*, as described in 3.2.

Interval time relations are expressed by the "true throughout" or TT predicate. The constraint:

```
(TT i f)
```

has been met if the fact f was kept true over the interval i. This type of constraint is used to set up *protection intervals* — periods where a fact is protected to be true so that some action may take place successfully.

There are situations where the exact order in which tasks are scheduled affects the constraints that exist on those tasks. For example, if the robot has the tasks of visiting the South Pole, and of launching a nuclear first strike against the the Antarctic penguins, then there are two choices: visit the Pole before or after the strike. It is very important that the two tasks do not overlap by even a little bit. In fact, it would be best, if the nuclear strike should come first, that there be a sizable delay between the attack and the visit. The delays and orderings can be set up by PRECEDES, FOLLOWS, and MAY-START constraints. Unfortunately, a contradictory set of constraints would be set up (e.g., *a* follows *b* and *b* follows *a*). To avoid contradictory constraints, allow flexible ordering, and still capture all the information the task specification may contain *conditional constraints*.

Conditional constraints depend on how the plan is being scheduled in order to decide whether to go into effect. They are specified using the IF-SCHEDULED and IF-NOT-SCHEDULED statements. Both of these statements take two arguments:

1. The task step whose scheduling status must be checked
2. The constraint that will (not) go into effect if the task step has (not) been scheduled

IF-SCHEDULED statements cause constraints to be added as a schedule is being created. IF-NOT-SCHEDULED statements initially add constraints to a problem and then cause them to be erased as the plan-schedule is developed. The two statements can be used in conjunction to cause a constraint to appear only if two tasks are scheduled in a certain order. For example, the constraint

```
(IF-SCHEDULED first-strike
    (IF-NOT-SCHEDULED visit-sp (MAY-START visit-sp 100 first-strike)))
```

would cause a delay of 100 between the nuclear first strike and the visit to the South Pole if the strike is scheduled to come before the visit. This constraint has no affect if the visit is scheduled to happen before the strike.

Conditional constraints may also be used to make small modifications to the plan design. A plan schema can contain two or more ways to accomplish a single step in the plan. When one of these ways is scheduled a conditional constraint can be used to make sure none of the other ways are also put onto the schedule. This is accomplished via the MAKE-EVAPORATE constraint. For example, the constraints in Figure 2.19 make sure that only one of the plans for sending a message will be used.

```
(IF-SCHEDULED send-telegram (MAKE-EVAPORATE send-letter))
(IF-SCHEDULED send-letter (MAKE-EVAPORATE send-telegram))
```

Figure 2.19: Using the MAKE-EVAPORATE Statement

When a plan contains a repeat loop it is possible to terminate the loop when a particular task step is placed onto the schedule using the MAKE-FINISHED constraint. For example, a typical high-school student's plan for getting an education is to read books on a subject until the test on that subject is taken. Any books that have yet to be read on that subject will never be read. More formally:

```
(IF-SCHEDULED take-test (MAKE-FINISHED read-books))
```

The conditional constraints can also be used to create *conjunctive* and *disjunctive* constraints. A conjunctive constraint is needed when two or more tasks must be scheduled to be executed with a limited time of one another, but their order does not matter. Figure 2.20 gives the conjunctive constraints for executing a proper parachute jump. Disjunctive constraints maintain a minimum time period between tasks — regardless of their order. The first strike against the penguins is an example of a disjunctive set of constraints. The message sending example given in Figure 2.19 is an extreme example of a disjunctive constraint set. In that example there is an infinite delay between executing the tasks.

Conditional constraints are also use to synchronize loops in a plan schema. The example in section 2.7.10 shows a specification where the termination of one repeat loop is linked to

```
(IF-NOT-SCHEDULED yell-geronimo (DEADLINE yell-geronimo 2 jump-from-plane))
(IF-NOT-SCHEDULED jump-from-plane (DEADLINE jump-from-plane 2 yell-geronimo))
```

Figure 2.20: Constraints for Executing a Parachute Jump

the termination of the other. The IF-SCHEDULED and MAKE-FINISHED constraints are used to terminate one loop when the other is finished.

2.7.9 The Utility Function

The statements of the plan schema are unordered unless inside a PROG. Figure 2.21 contains a schema with several statements. Because they are in a PROG, the statements: go to the basement, hammer in the nail, and go upstairs will be done in the order in which they are listed. However, the schema does not specify at what point it will read *Critique of Pure Reason*. The book may be read before the other task is started, during that task, or at some later time. It is the job of the task scheduler to decide on the exact ordering. The plan schema provides only enough ordering information to guide the scheduler with constraints that are absolutely necessary.

```
(PLAN-SCHEMA
    (STEP s11 (read 'critique-of-pure-reason))
    (PROG p1
          (STEP s21 (goto 'basement))
          (REPEAT r1
                  (WHILE (above? nail wood))
                  (STEP r1s1 (hammer 'nail)))
          (STEP s22 (goto 'upstairs))))
```

Figure 2.21: Plan Schema for Hammering a Nail and Reading a Book

Occasionally when writing a task specification the author may envision the "best" way to accomplish the task, but hesitate to enforce that way on the robot sight unseen. For example, for the schema in Figure 2.21 the designer may know that the robot will be able to concentrate better, and get more out of the book if it is read in one of the comfy chairs available in the upstairs room. However, it is possible that in some situations (e.g., where the upstairs is occupied by several rowdy robots) it would be better to read the book in

the basement between hammer strokes. To aid in the scheduling of these tasks, the task-specification designer may include a *utility* function in the specification. This function takes the world state and partial schedule as arguments and returns the value for doing each step of the schema in the current context. For the schema in Figure 2.21 the utility function can rate the book-reading task very highly if the robot is upstairs and and there are comfy chairs available. If conditions in the upstairs room are not very favorable then the function will return a higher value for the book-reading step when the robot is located downstairs. The values returned by the utility function are integrated with values returned by other scheduling heuristics, and are used by the scheduler to determine the exact task ordering (see section 4.7.3).

When evaluated, the `UTILITY` statement must return a function that takes four arguments and returns an integer between zero and one-hundred; one-hundred indicates that the plan-step should definitely be scheduled next, fifty means that it does not really matter, and zero says that if any other plan step can be done instead — it should. The four arguments that a utility function takes are:

1. The schedule so far
2. The "current" state of the world
3. The "current" time
4. The plan step.

The `UTILITY` statement can reference any variables that are known at the time that the task specification is instantiated. For example, the statement:

```
(UTILITY (reading-location-utility comfy-quotient))
```

should return a utility function, that takes the four utility-function arguments, for the plan schema in Figure 2.21. The parameter `comfy-quotient` must be one of the task specifications parameters or declared variables — it must have a value after the specification has been instantiated, but before any scheduling has taken place.

2.7.10 Example Task Specification

This section presents the specification, and the reasoning behind it, for having a mobile robot perform a complex feedback-driven task. The task is for the robot to navigate down

a long hallway and stop when it gets near the end — before running into the far wall. The robot is of limited mechanical accuracy and therefore must use feedback to contiuously monitor and correct its path during its travels. There are also time constraints on the execution of this task; the robot must perform the task at its maximum safe speed.

A desirable level of detail to have the user describe a task such as traversing and stopping at the far end of a hallway, would be something like:

1. `roll-down-hall hall-132`
2. `stop-at-end-of-hall hall-132`
3. `finish within 200 seconds.`

The map of `hall-132` is shown in Figure 2.22. The route planner breaks this task into the traversal of V-regions `hall-132-vr1`, `hall-132-vr2`, `hall-132-vr3`, and `hall-132-vr4` where the first, third, and fourth are through three-F regions, and the second a two-F. In this domain there are only a limited number of top-level tasks that are possible. They consist of different region-types to be traversed. The task specification for the traversal of `hall-132-vr2` provides a good example of most of the features of the task specification language that has been discussed above.

The traversal of `hall-132-vr2` occurs in two parts. This is due to the uncertainty in the length of the V-region. In the first part of the traversal the robot relies on its basic wall following strategy discussed in section 2.6.1. The robot will follow the wall without any "fear" of collision until it has traversed a distance of 40, the lower bound on the length of the V-region.

For the remainder of the robot's trip down `hall-132-vr2` the robot must be on the lookout for the far wall —a sign that the robot has passed into `hall-132-vr3`. In order to accomplish this the robot must start executing the proper observations (described in section 2.6.1) to spot the far wall as soon as it appears, or at least in time to avoid colliding with it. It is impossible in advance to predict when the robot will first sight the far wall; all that is known is that the robot should sight it by the time it has traversed a distance of 60 from the start of `hall-132-vr2`.

With the information available about the region, the task-instantiator module of the Bumpers system can choose a plan schema to accomplish the traversal of the region. The choice of plan schema is done in two parts: first the plans with the correct index are found, then the constraints are checked to find the most specific plan for the task to be accomplished. The constraints that are checked are the `TRUE` constraints in the `WITH-CONSTRAINTS`

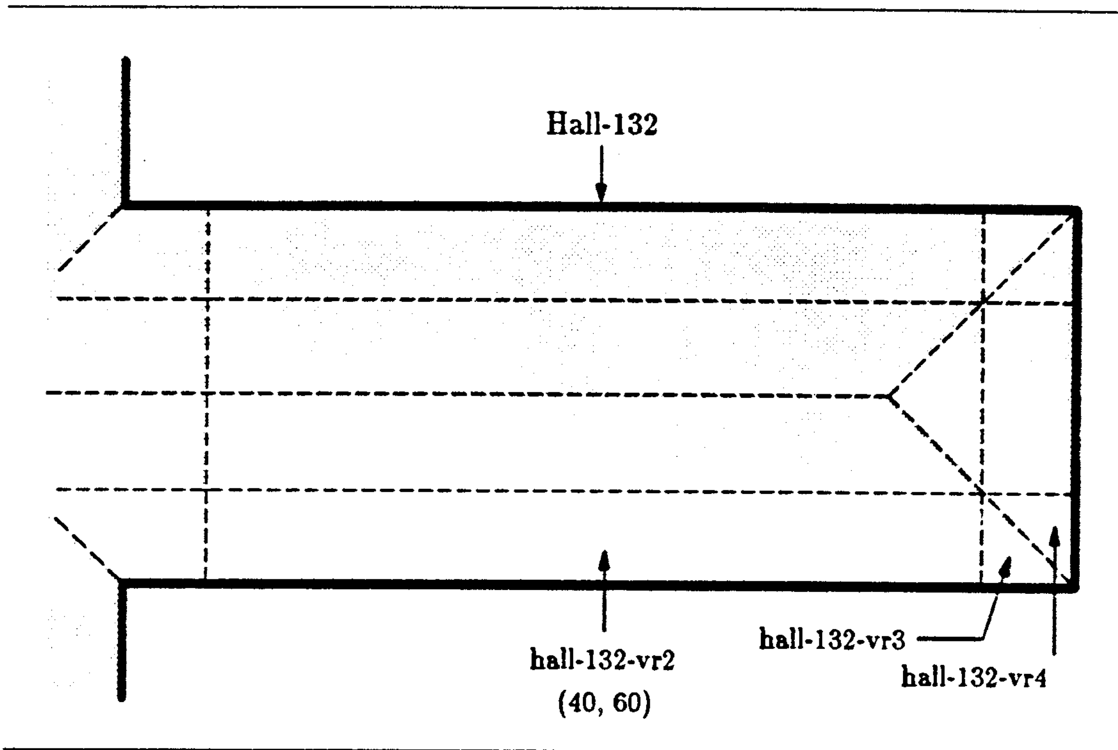


Figure 2.22: Map for the Example Task

portion of the specification. For the region hall-132-vr2, the best specification is given in Figure 2.23; a specification for traversing a two-F region terminated by a three-F region where the landmark is perpendicular to the traversal path.

The WITH-RELATIONS statement introduces the scheduling variable ?drive-speed into the specification. This will be used later by the scheduler to calculate the time it takes for the robot drift laterally one half the value of optimal-dist.

In the specification the loop rl1 will be repeated every time the robot might be expected to have deviated a quarter region width from its course. The loop rl2 is done every time the robot travels a distance d-max. The second loop will terminate as soon as the landmark is encountered; the termination of the second loop causes the first loop to be halted. The monitor dist-to-wall is updated by the function EVALUATE. The monitor retains the value of its last update until it is re-evaluated. dist-to-wall is used to keep track of the distance to the hall-way wall. dist-to3f contains the value used for monitoring the robot's distance from the end of the hallway.

2.8 How TSL Compares to Other Planning Languages

Early least-commitment plan languages such as SOUP (used in Noah) and TF (developed for the Nonlin system) were capable of conveying some basic facts about the tasks, and the plans for achieving them. Information about:

- the prerequisites for a task
- the required state of the world
- the tasks place in the plan's partial order
- the effects caused by achieving the task

are all somehow represented in the planning language. Additionally, some of the languages were able to differentiate between preconditions that must exist before the plan could be attempted and preconditions that the planner should try and achieve on its own. In TF these were referred to as ***unsupervised*** and ***supervised*** preconditions, respectively. Unsupervised preconditions must be met by some other plan, while supervised preconditions should spawn a new subtask to be achieved — whenever possible.

The plan language used in Deviser made a significant addition to the plan-language literature by adding some references to metric time. Each task in Deviser can have a temporal window associated with it as additional constraints on the place in the final plan that the task may occupy.

TSL allows all of the information above to be clearly specified by user. The inclusion of metric time allows other information, concerning resource management, to also be included. Resource management is a major difficulty in creating workable plans. With a simple representation of time, and how resources will be used over time, it is possible to create plans that not only avoid resource conflicts, but actually make efficient use of the resources. TSL contains the `WITH-RESOURCES` and `WITH-CONSTRAINTS` statements to alert the planner to the pattern of resource usage that can be expected. The state restrictions, discussed in the next chapter, further extend these abilities.

The inclusion of metric time also allows loops to be described and constrained in ways other than counting the number of iterations performed. In TSL it is possible to tie a loop's conclusion to a deadline or a particular resource's consumption. Statements in TSL concerning time, loops, and resources are designed to be easy to write by the system user, yet give the maximum information to the planner. A planner such as Bumpers, given well

specified tasks written in TSL, can perform more complex reasoning on those tasks then has been possible in previous planning systems.

TSL is a user-friendly way of organizing all of the relevant problem information. However, it is not necessarily the best way of organizing that information for use by the planning system The next couple of chapters present the internal representations used by the task scheduler for actually carrying out the planning process. The information used by the planner is gleaned from the task specifications, but it is manipulated through other structures.

```
(TRAVERSE-1F-TO-3F-REGION end-1F-perpendicular-wall (region obs-dir dl)
    (WITH-MONITORS
        (dist-to-wall (SONAR))
        (dist-to-3f (SONAR)))
    (WITH-DECLARATIONS
        (optimal-dist (max 6 (/ (: WIDTH region) 2.0)))
        (d-max 12)
        (initial-position (STATE 'position))
        (min-region-length (: MIN-LENGTH region))
        (max-region-length (: MAX-LENGTH region)))
    (WITH-CONSTRAINTS
        (TRUE (= 0 obs-dir))
        (STATE-REQUIREMENT r12 (>= (- (STATE 'position) initial-odometer)
                                   (- min-region-length d-max)))
        (STATE-REQUIREMENT r12 (=< (- (STATE 'position) initial-odometer)
                                   max-region-length))
        (MAY-START r12 0 (: START r11))
        (STATE-REQUIREMENT r11-s1 (= (STATE 'sensor) 45))
        (STATE-REQUIREMENT r12-s1 (= (STATE 'sensor) 0))
        (IF-SCHEDULED r12 (MAKE-FINISHED r11))
        (DEADLINE r11 dl)
        (DEADLINE r12 dl))
    (WITH-RESOURCES
        (position *varied* overall)
        (sensor 45 discontinuous)
        (sensor 0 discontinuous)
        (drive on discontinuous))
    (WITH-RELATIONS
        (max-lateral-drift (/ max-lateral-drift
                              (* ?drive-speed (drift-speed ?drive-speed)))))
    (PLAN-SCHEMA
        (REPEAT r11
            (WHEN (CHANGE 'lateral-drift (/ optimal-dist 2)))
            (STEP r11-s1
                  (adjust-steering (wall-correction (EVALUATE dist-to-wall)
                                                    optimal-dist)))))
        (REPEAT r12
            (WHEN (CHANGE 'position d-max))
            (STEP r12-s1 (EVALUATE dist-to-3f))
            (WHILE (> dist-to-3f d-max))))
    (UTILITY (TRAVERSE-1-F-UTILITY region landmark dl)))
```

Figure 2.23: The Complete Specification for Traversing a 2-F Region

Chapter 3

Resources, Physics, and Scheduling

The world of robot planning is not a simple place. Plans interact with one another, often adversely. The most common way for plans to interact is through competition and manipulation of resources. Plans often involve the use of physical resources or require a particular state of the world. Unfortunately, plans for tasks being solved in parallel may require the same resources and different world states. These conflicting requirements can happen at all levels of planning.

The resolution of resource and state conflicts is one of the jobs of the task scheduler. In order to accomplish this job the scheduler needs to know certain things about the required resources and world states. The resource information needed by the scheduler and how the scheduler represents this information are the subjects of this chapter.

3.1 Specifying the Physics

3.1.1 The Syntax of Resource Declarations

Section 2.7.6 discussed the construction and declaration of *delay functions*, functions which return the amount of time necessary to move a single resource from one state to another. This was discussed in the context of the `WITH-RELATIONS` statement of the task specification. The `WITH-RELATIONS` statement from one of the example tasks of the previous chapter is

shown in Figure 3.1. The statement presents a resource (the robot's lateral drift) and the relationship for calculating the speed at which the state of that resource occurs.

```
(WITH-RELATIONS
    (max-lateral-drift (/ max-lateral-drift
                          (* ?drive-speed (drift-speed ?drive-speed)))))
```

Figure 3.1: The WITH-RELATIONS Statement for Calculating Lateral Drift

The WITH-RELATIONS statement is used for special purpose resources that occur in a single or very few task specifications. For standard resources (resources that most of the specifications access), the delay functions are defined as part of the physics of the domain. They are declared outside of any single task specification.

```
(RESOURCE-PHYSICS resource
    (MOVE ( s2 s1 time )
      -code- )
    (DELAY ( s2 s1 time )
      -code- )
    (PLAN ( s2 s1 time )
      -code- )
    (VARIABLE-STRATEGY
      (VARIABLE var
         (VALUES {discrete | continuous} values )
         (DEFAULT value )
         (COORDINATION-STRATEGY strategy-1 strategy-2 ... ))
      ...))
```

Figure 3.2: Syntax of the Physics Declaration

The RESOURCE-PHYSICS statement shown in Figure 3.2 is used to describe the physics of a particular resource. Statements describing each resource are given to the planner, along with the task network, as part of the initialization of the planning system. For each resource three things must be known in order for intelligent planning to be done:

Move: Can the resource be moved into a particular state?

Delay: How long would it take to move the resource to that state?

Plan: How is the resource moved to that state (i.e., what is the plan for accomplishing the resource's state transition)?

The MOVE, DELAY, and PLAN fields of the declaration contain functions that answer these questions. Each function takes three arguments:

1. The desired state of the world
2. The "current" state of the world
3. The "current" wall-clock time

In our planning paradigm, a search is performed over the space of totally ordered plans. Since the plans are totally ordered, "current" refers to the present simulated spot in the plan. The functions included in the resource specification strip the appropriate information from the states (i.e., the information in the state descriptions that concern that particular resource) and perform their calculations.

3.1.2 Scheduler Variables

The functions in a resource declaration may reference *scheduler variables*, parameters whose value is determined and set during task scheduling. Scheduler variables are used when the correct value of the parameter depends on the context in which that section of the plan will be executed. Motor speeds are commonly expressed as scheduler variables since the best speed to use is the one that best coordinates all of the tasks being planned for.

Information about scheduler variables is contained in the (optional) VARIABLE-STRATEGY section of the RESOURCE-PHYSICS statement. This declaration includes the values that the variable can be assigned, the preferred value, and the basic coordinating strategy. Scheduler variables are distinguished by a "?" prefix. Guidelines for the assignment of scheduler variable is often encoded into the RESOURCE-PHYSICS declaration. These include maximum and minimum values along with general strategies for coordinating one resource with others.

The range of values that the variable can attain are listed in the VALUES part of the declaration. Discrete values are listed one at a time in separate VALUES statements. A range of values can be listed using the continuous switch on the VALUES statement. The lower and upper bounds are given in the statements. VALUES statements are given in order from lowest to highest — discrete values do not have to be expressed numerically.

There may be several different ways in which resources may be manipulated to achieve the necessary state for a particular plan step. The scheduler variables can control the way in which some of the resources are manipulated, and the rates at which changes in

the resources state are affected. For example, the drive-motor's speed may be controlled by the scheduler variable `?speed`. In this case the rate at which the robot's position is changed depends on the value given to the scheduler variable. If the robot's position must be coordinated with the state of some other resource, the value of the scheduler variable can affect the way in which the resources are coordinated. Scheduler variables are assigned coordination strategies in order to aid in the assignment of their values. The coordination strategy for a scheduler variable can be a combination of:

highest+wait — set this variable to the highest value it can have have, achieve the desired state, and wait for the other resources to reach their states.

lowest+wait — basically the same as highest+wait, except the scheduler variable is set to its lowest acceptable value.

default — ignore the other resources and just set this variable to its default value.

match-last — set the value of the scheduler variable so that the resource reaches its state as close to, but not before, the time the last of the other resources reach theirs.

match-first — set the value of the scheduler variable so that the resource reaches its state as close to, but not later than, the latest of the other resources.

match-best — set the value so as to have this resource reach its state as close as possible to the time the other resources reach theirs.

In the robot-feedback domain the "match" strategies are commonly used. For example, in the robot-feedback problem of having a robot navigate down a long hallway, it is necessary that the robot have its sensor pointed at a specific direction every time the robot's position changes by some pre-specified distance. If the speed of changing the robot's position was controlled by a scheduler variable, then the coordination strategy might check to see if the robot's position could be brought into the desired state in less time than would be required to move the sensor. If the position could be changed faster than the direction of the sensor, then the scheduler variable would be adjusted so that the position change would take the same amount of time as was needed to move the sensor — the `match-first` strategy. If it could not, then the position should probably be changed as fast as possible (since it cannot be changed as fast as is desired). The alternative strategy for the value of `?drive-speed` would be `highest+wait` (see Figure 3.3).

3.2 Resources and State Changes

To be useful, a planner must be able to deal with all sorts of domain-dependent resource issues. Yet the task scheduler runs by heuristics and must (if it is to function well in a

```
(RESOURCE-PHYSICS position
    (MOVE (s2 s1 time)
          t)    ;;; It is always possible to change the position
    (DELAY (s2 s1 time)
                ;;; The delay depends on the amount of
                ;;; change being made in the position, and
                ;;; the speed at which the robot moves
          (/ (abs (- (<o state 'position s1)
                     (<o state 'position s2)))
             ?drive-speed))
    (PLAN (s2 s1 time)
                ;;; The plan for moving the robot depends
                ;;; on the states it is moving from and to.
                ;;; The expression  (<o state  resource state)
                ;;; is used to access the resource's state from
                ;;; the world state
          (:= drive-direction (cond ((< (<o state 'position s1)
                                        (<o state 'position s2))
                                     'forward)
                                    (t 'reverse)))
          (:= drive-speed ,?drive-speed)
          (drive-on ,(abs (- (<o state 'position s1)
                             (<o state 'position s2)))))
    (VARIABLE-STRATEGY
                ;;; This section gives the speeds that the
                ;;; drive speed can be set at, its default
                ;;; and the strategies used to assign its value
          (VARIABLE ?drive-speed
                  (VALUES discrete 0.0)  (VALUES discrete 0.2)
                  (VALUES discrete 0.4)  (VALUES discrete 1.0)
                  (DEFAULT 1.0)
                  (COORDINATION-STRATEGY match-first highest+wait))))
```

Figure 3.3: Resource Declaration for the Robot's Position

variety of domains) keep its heuristics domain-independent. The previous sections have described how resource information is specified by the user. This section presents the structure that the task scheduler uses to organize and manipulate the resource information — the *State Object*. The task scheduler isolates all domain-dependent resource information in the state object, thereby providing a standard interface between the domain-dependent and domain-independent parts of the scheduling process.

A State Object (SO for short) is created for each task that is to be worked on by the scheduling module. The State Object (see Figure 3.4) keeps track of, and is used to manipulate, all of the state information required by the scheduler for that particular task.

The SO serves as the data structure for collecting and organizing all of the information in the **WITH-RESOURCES** statement that is relevant to a particular task. The scheduler does a limited simulation of each schedule that it works on, and it is the SO's job to make sure that the computed world is in the correct state for each step of the simulation. In other words, an SO is created for each task given to the scheduler and that SO contains a copy of the state of the world required for the task to be placed onto the schedule. Additionally, a task's SO can project the state of the world that will exist once the task has been executed.

```
(Object
    PROPERTIES        ---> a list of relevant state properties
    STATE p           ---> the state that property p must be in
    RESTRICTION p     ---> limitations on the state
                            transition of property p
    MOVE    so time   ---> returns true if a
                            transformation is possible
    DELAY   so time   ---> time needed to transform the state
                            defined by so at time t into the
                            required state
    UPDATE  so time   ---> returns the state object that would
                            result from performing the transition
    )
```

Figure 3.4: The Fields of a State Object

State Information involves any aspect of the world that might be changed intentionally by the robot, or might be changed by some outside event, in a predictable manner. Typical aspects that are kept track of by the SO in the mobile-robot domain are:

- The robot's position
- The orientation/availability of the robot's sensors and manipulators
- The speed settings of the robot's motors
- The availability of outside resources to be used by the robot, e.g., tools, raw materials

In other words: if a task is being considered for scheduling next, its SO is capable of reasoning about

- whether the task's resource requirements can be met
- when those resources will be available
- what the effects of the task will be on the rest of the world.

Each task's SO is a computational theory of what the next small stretch of time might possibly look like. The SOs compete against one another to make their task look the most attractive to the scheduler so that their task will be placed next onto the schedule. When the next task is chosen for the partial schedule, its SO computes what the state of the world will be following the task. The remaining tasks' SOs then compete to make the section of time after that look attractive for their task. This continues until all the tasks have been placed onto the schedule, and the state of the world is described at any point in the schedule by the appropriate SO.

For example, suppose there are two tasks A and B needing to be scheduled. A requires the robot to be in Manhattan; B needs the robot in Chicago. If the last task scheduled places the robot in New Haven, the SOs are going to calculate the minimum expected time that would be needed to get the robot from New Haven to their respective cities. The smallest transition delays for traveling from New Haven to Chicago, and from New Haven to Manhattan will be calculated. The results calculated by the SOs will be used by the scheduler to help decide which task should go onto the schedule next.

The SO is also designed to handle restrictions on any or all of these resources. Restrictions limit the way in which these resources can be used. These limitations can take the form of temporal restrictions, availability, and limits on how any resource can be transformed from one state to another (see section 3.2.2).

In summary, the SO is used to provide the scheduler with a uniform mechanism for getting information on any aspect of the robot's world for a given simulation state.

3.2.1 State Delays

The first piece of information that the scheduler requires of a task's SO is whether the robot is directly capable of transforming the world from $state_1$ to $state_2$, and if so, how long such a transition would take.

In order to answer that question a ***transition path*** must be constructed for every resource that is specified in both the states. Note that not all resources need be specified in every state. It is quite possible that a task will demand that the robot be located in Chicago, but not mention, nor care, whether or not its left manipulator is available. In such a case a transition path needs to be calculated only for the robot's position. The manipulator, and all other resources not specified in $state_2$ can, for the moment, be assumed to carry on in their previous conditions.

Each transition path is calculated by a specialized function applicable to the particular resource being worked on. The transition functions can vary in complexity from simple assignment statements to involved planning systems of their own. In each case, if the transition is possible then the functions return estimates on how long such a state change would require. Complications arise when the transition delay is some function of one or more other resources under the robot's control.

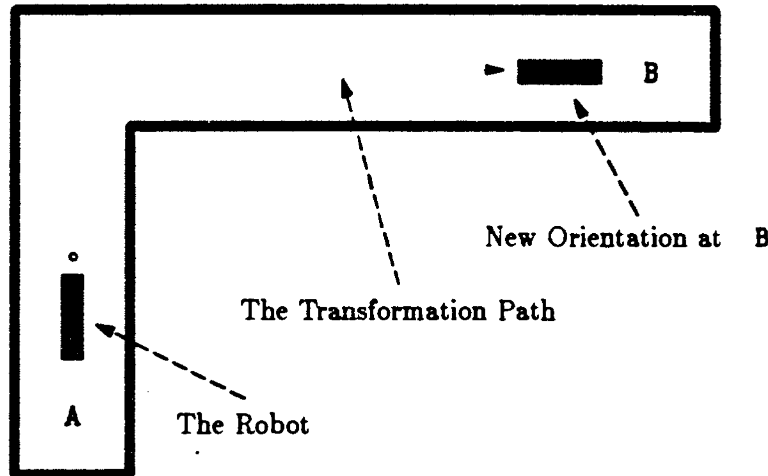


Figure 3.5: Transition Function Side Effects

There are many transition functions that have side effects on aspects of the robot's world aside from their specified resource. An example is the robot's position and orientation (see figure 3.5). The standard transition for moving the robot from A to B will alter the robot's orientation by 90°. If the orientation does not matter then the obvious position transition is clearly the fastest plan. Even if the orientation is important, a position transition which restores the robot's orientation is not necessarily the best idea. If the desired orientation happens to be 90° clockwise of what it was at A then a position transition that restores all of the robot's other aspects to their original values would be very inefficient. For this reason it is critical that the order in which the transition functions are performed be carefully considered — something that must be done by the system designer. The SO functions can be written so that those transition functions that produce side effects are done before the functions on which they have their effects. In cases where both transitions may have side effects, the magnitude of the effects are used by the designer to decide which transition should be attempted first.

A more subtle form of state-transition interaction can also exist. Imagine a robot with two motors: one for the drive, the other to turn the robot's head. If the robot's batteries can supply only a limited amount of power then it is quite possible that, when both motors are running simultaneously, the speed of one is dependent on the speed of the other. If the robot's desired state is to move ten feet and have the head rotate forty degrees, and the motors operate with the speeds shown in table 3.1, then the time it will take to do these two actions is not trivial to calculate.

Motor	Speed Alone	Speed Together
Drive Motor	$2.5 \frac{feet}{second}$	$2 \frac{foot}{second}$
Head Motor	$3 \frac{degrees}{second}$	$2 \frac{degree}{second}$

Table 3.1: Motor Speeds: Alone and in Conjunction

In such a situation the function calculating the transition path for the head must be able to know when the drive motor is on. It must also be able to represent that the drive motor will only be on for at most five seconds during its transition, and thus the head motor will be able to go at full speed during the remainder of its transition. For interactions of this kind the SO must:

1. calculate all of the transition delays for the slowest speeds
2. arrange the transitions in order of increasingly large delays
3. recalculate all but the smallest in this new order

For this example the times for transforming the robot's head and body to the desired state are twenty and five seconds respectively. Since the body-transition is the smallest, it is left as calculated. The head-transition delay is recalculated with the additional information that the head will be moving alone after the first five seconds. In those first five seconds the head can rotate ten degrees; its speed then goes upto $3 \frac{degrees}{second}$ and it can therefore finish the transition in an additional ten seconds. Thus, the SO will report a transition delay of fifteen seconds for getting the robot's body and head into the desired state.

A final complication that arises when calculating the transition delays is when the time it takes to perform a state transition on a resource is not necessarily the amount of time the schedule should be delayed by. For example, the three states shown in table 3.2 represent a problem where if the state changes are calculated one at a time, the transition delays will

appear to be larger than they need to be. If it takes one second to move a motor one step then a delay of five between $state_1$ and $state_2$ is necessary. During that delay the head of the robot could be moved in order to prepare it for the state described in $state_3$. It would be advantageous if the system could somehow realize that the head movement needed to achieve $state_3$ can start any time after $state_1$ is no longer required, and does not have to wait until $state_2$ has been finished.

state	Head Position	Body Position
$state_1$	0	0
$state_2$	—	5
$state_3$	5	5

Table 3.2: Arranging Resources for Efficient State Changes

This problem is solved through the use of ***time-stamps***. A time-stamp is a date attached to each resource in the SO. The date represents the time a specific state was last required of that resource. When delays are calculated for a certain transition path between $state_i$ and $state_j$ the difference between the time associated with $state_i$ and the time-stamp for the resource is subtracted from the transition delay. In this way, assuming the motors can be moved together at their full speed, the delay between $state_1$ and $state_2$ is still calculated to be five, but the delay between $state_2$ and $state_3$ turns out to be zero. Unfortunately, there is no general optimizing strategy if the two motors interfere with one another in the way shown in table 3.1; some search is required to find the optimal arrangement of transition paths.

3.2.2 State Restrictions

A SO is meant to be able to handle and represent arbitrary restrictions on the resources under its control. The restrictions may be time dependent, triggered or removed by a specific task or set of tasks, or controlled by the state of other resources represented in the SO.

For example, to represent the fact that there is an impenetrable traffic jam on the highways leading into New York City every weekday morning the SO could contain a restriction against ground traffic into New York between 6 and 9am. If a state transition

demanded that the robot be moved from Yale to NYU with the transition starting at 7am, then the transition planner would either avoid ground transport altogether or add two hours onto the delay that it would normally calculate, i.e., it would wait until the restriction had expired and then carry on with its normal plan.

If the robot was only capable of operating one of its motors at a time, any task that required a motor movement would put a restriction on all the other motors. It is through the use of restrictions that the SO, and hence the scheduler would be able to recognize that a request for such a robot to pat its head while rubbing its stomach would simply not be feasible.

Restrictions can be used to literally keep the planner from painting itself into a corner. If the robot is given a task to paint the floors of the three rooms in figure 3.6, and then to go out of the house and do the gardening, the SO's restrictions will keep the robot from traveling over wet floors. Restrictions that keep the robot from traveling over newly painted floors do more than just keep the robot from leaving unsightly wheel marks all over the house. Restrictions on travel of this kind make schedules requiring painted floors to dry before painting the next room less attractive to the system then schedules that do not require the drying time. Thus the restrictions aid the scheduler in picking the more efficient schedule for the robot: painting the inner rooms first.

Restrictions are composed of four basic parts:

- Trigger
- Property
- Action
- Release.

They are normally placed inside the `WITH-CONSTRAINTS` statement of the task description. The syntax for a restriction is given in Figure 3.7.

The *trigger* is the test that decides when the restriction should come into effect. In most cases, as in the examples so far presented, the restriction is strongly associated with a particular task. In the floor painting example the trigger is the scheduling of the painting task. In such a case the trigger is an `IF-SCHEDULED` statement wrapped around the body of the restriction. The restriction is activated as soon as the triggering task is put onto a possible schedule.

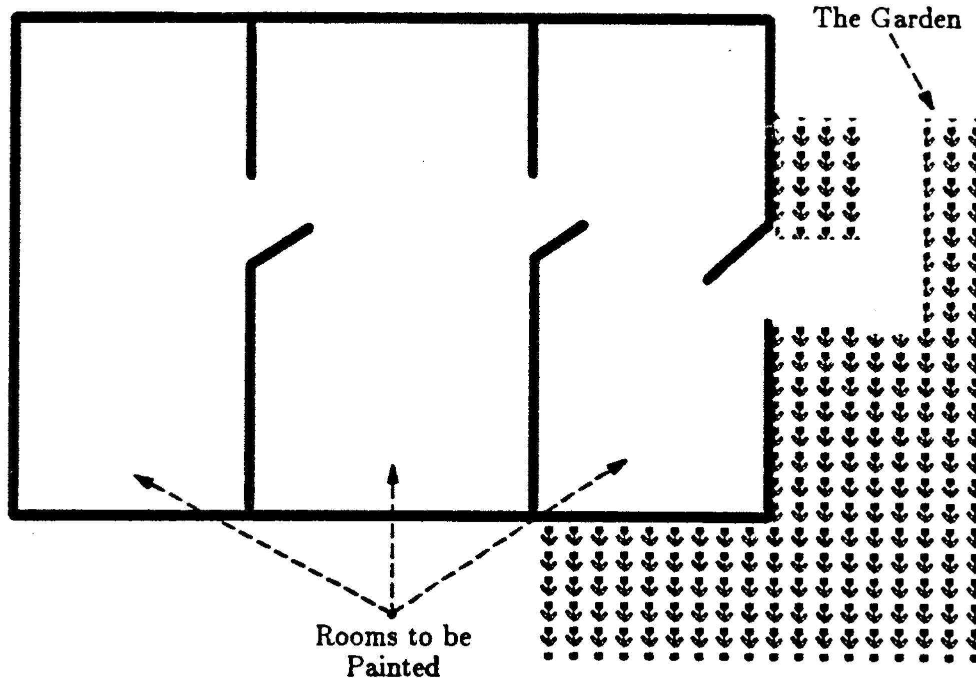


Figure 3.6: Three Rooms to be Painted

```
(SO-RESTRICTION
    property
    (ACTION { FILTER | BEHAVIOR } action)
    { ( { REL-TIME | ABS-TIME } time) | (TASK task) }
    )
```

Figure 3.7: The Syntax of the Restriction Statement

In other cases a task may just create a situation under which the restriction will eventually come into effect. For example, if the robot started up a piece of machinery that produced metal shavings as it operated then the startup task might have had a restriction associated with it that says that the machine would become inoperative after two hours of running unless an empty the shavings bin task was scheduled. In cases such as this, the trigger is a **MAY-START** statement containing the restriction.

The *property* refers to the resource that is linked to the restriction. The property is not necessarily related to the properties handled in the trigger or the resource's action. In the floor-painting example the property controlled by the restriction is the robot's movement. The restriction's property indicates that once the restriction has bee triggered, a check will need to be made whenever a transition path is being computed for that particular resource.

Another example of a restriction property is if the robot is given a supertask move a heavy container from the bottom of a hill to the top. The subtask pick up the box may place a restriction on the robot such that it cannot move up a slope steeper than 5° unless it is in low gear. There are two paths up the hill, one a shallow spiral, the other a direct steep path. The restriction is triggered by the lifting task. Its action at schedule time will be to lock the gearbox into a particular state. The property of the restriction will be the robot's position. The state of the gearbox becomes important when a transition path for the position of the robot is being calculated.

The restriction's *action* can fall into one of two categories:

- A *filter* through which the desired final state of the restriction property must pass.
- A *behavior modifier* which is sent to the transition function to guide its computation towards the desired result.

An example of a filter action would be: If the robot is given the task guard a painting with the restriction stay within ten feet of the painting at all times, and if the next task the system tries to add to the schedule is eat at the restaurant at the top of the Sears Tower, then it would be nice if the restriction fired off a violation before the transition path is even calculated. When the violation is detected the schedule would fail and the scheduler would try to expand another partial schedule. A filter type of restriction action is used in situations where the final state is the only concern, not the transition path used to achieve it. Positional restrictions and resource availability constraints are commonly represented as filter restriction actions.

When there may be many transition paths, only some of which will be acceptable, a behavior-modification restriction is used. In these cases there is not a straightforward test to calculate whether or not a transition is possible. Instead, the transition path (i.e., the plan for transforming the world to the desired state) must be generated; the path generated is influenced by the behavior modification restrictions that are in force.

An example of this type of restriction would be if the robot in figure 3.8 was considering a schedule in which it had just completed the task: fill room B with toxic waste, and was now

in room A and wanted to do a task in room C. The transition path used to move the robot from room A into room C is provided by a route planner accessible to the SO. Assuming that the deposit-toxic-wastes task placed a restriction against traveling through room B, then the restriction must guide the route planner to find an acceptable route. Figure 3.9 shows the restriction used in this case. One strategy to do this would be for the route planner to generate routes until one was produced that met the requirements of the various restrictions that were active. A better strategy is to use the restrictions when creating the transition paths. In this case the restriction causes room B to be removed from the route planner's map. Under this strategy, if a transition function is able to come up with a path then the path will be acceptable. The floor-painting problem mentioned earlier is also an example of a behavior modification restriction.

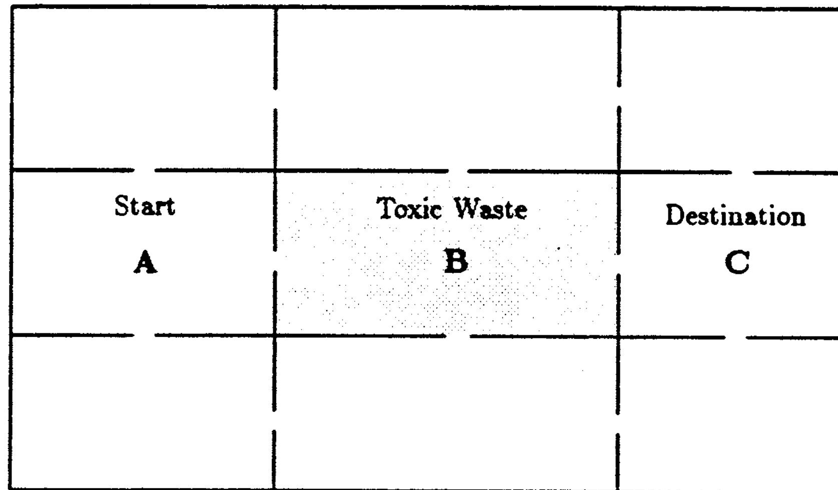


Figure 3.8: Many Paths; Few that are Acceptable

The fourth part of a restriction, the *release*, is simply the event that causes the restriction

```
(IF-SCHEDULED fill-B-with-toxic-waste
      (SO-RESTRICTION
          'position
          (ACTION BEHAVIOR (not (memq 'B route)))
          (REL-TIME 1000)))
```

Figure 3.9: A Restriction Against Traveling Through Toxic Waste

to go away. A release is either a specific task or a particular time. When the task is scheduled or the time is past, the restriction is deleted from the SO. Time-dependent releases can be checked during the generation of a transition path. In the floor-painting example, the restriction against the robot traversing a newly painted floor is in effect from the time the floor is painted until the floor is dry, say four hours later. When the scheduler has scheduled events more than four hours from when the floor is scheduled to be painted, the restriction is deleted. In this way a restriction, for example prohibiting the use of a lathe until noon, can have that time figured into the calculation of the transition delay. Thus, if the scheduler's last task in the prefix is scheduled to finish at 10am and the next task up requires the lathe, then the SO will report that a two hour delay is necessary before that task can be scheduled. This might raise the cost of that particular schedule sufficiently that the scheduler will decide to switch to another; but for completeness' sake it is necessary to represent that the schedule is possible though perhaps inefficient.

Chapter 4

The Scheduling Algorithm

4.1 Introduction

The role of the task scheduler is to convert a partial ordering of tasks into an efficient executable total ordering. The total ordering has associated with it a sequence of temporal intervals; one interval associated with each step in the ordering. This allows the robot to monitor its progress through the plan as well as allowing the total ordering to be integrated with other schedules that are to precede and follow it. In order to ensure that the ordering produced by the scheduler is a good one, the scheduler performs a limited simulation on the ordering. This system simulates tasks and calculates the state that would result from the execution of those tasks. Additionally, this system uses the results from partial simulations to efficiently order tasks that are to be performed later in the schedule. Towards this end the scheduler searches through the space of possible task orderings in an attempt to find a reasonably efficient ordering. Since the number of possible schedules rises exponentially with the number of tasks being scheduled, the system relies on scheduling heuristics [Vere 84], [Miller 83] to trim the search tree to a reasonable size. The search heuristics rely on the following properties that are associated with each task:

- The location and type of resources required for the task
- Partial ordering constraints with respect to other tasks
- The absolute execution window for the task (e.g., do this between 9:00 and 11:00am)
- Metric ordering information with respect to other tasks (e.g., do B between fifteen and twenty minutes after completing A)

The first of these properties is domain-dependent, but can be handled in a standardized manner through the use of state objects. The remaining properties of the tasks, and their relationships with one another, can be used to reduce the total scheduling space in a totally domain-independent way. The reduced search space can then be searched using further domain-independent and dependent heuristics to help guide the search. What these properties are, how the information about them is encoded, and how this is all worked into a coherent search strategy are the subjects of this chapter.

There are several basic data structures used by the scheduling system during the scheduling process. The information needed to do a scheduling is initially provided in each task's description. In particular, this information is contained in the WITH-CONSTRAINTS and PLAN-SCHEMA sections of the description (as described in section 2.7). When a list of tasks to be scheduled is first being processed, the necessary information is skimmed off and encoded in data structures usable by the scheduler. The methods in which this information is encoded are described below.

Relations between tasks are represented as *Prerequisite Objects* (section 4.3). Prerequisite Objects contain ordering information, mandatory delays between tasks, and relative deadlines from one task to another. One Prerequisite Object is used for every two tasks that are related to one another. The prerequisite objects are derived from the PRECEDES and FOLLOWS constraints along with the relative forms of the MAY-START and DEADLINE constraints.

A *Current Status Object*, or CSO (described in section 4.5) is also created for each task. The CSO keeps track of a task's availability for scheduling. It is the depository for preemption and looping information. It also figures prominently in the scheduling of disjunctive tasks; relations that are formed from conditional ordering and evaporative constraints.

A state object (section 3.2) is used to represent and organize information about the resource requirements of a task. One SO is associated with each task, and is passed to the task scheduler with its task.

Also associated with each task is an ***absolute execution window***. This window is an interval of time within which the task must be scheduled. The execution window is formed from the DEADLINE and MAY-START absolute time constraints in the WITH-CONSTRAINTS section of the task's specification.

The State Objects, Prerequisite Objects, CSOs, and absolute execution windows are all loaded into the *Schedule Dependency Array* (the SDA for short). The SDA serves as the

main generator for guiding the schedule search. It is the job of the SDA to produce a list of the next tasks to be scheduled, and to give an initial rating on which would be the best task to work on at each stage of the scheduling process.

A detailed description of the basic scheduling process is presented next. Immediately following the scheduling algorithm are more detailed descriptions of the data structures manipulated by that algorithm. Finally, the methods in which perspective schedules are rated and chosen, are discussed.

4.2 The Scheduling Process

The task scheduler uses a heuristically guided best-first search for creating the plan schedule. For this type of search, all the conceivable schedules can be thought of as being arranged in the form of a tree as in Figure 4.1. Of course to make the schedule search efficient, as little of the search tree as possible is generated.

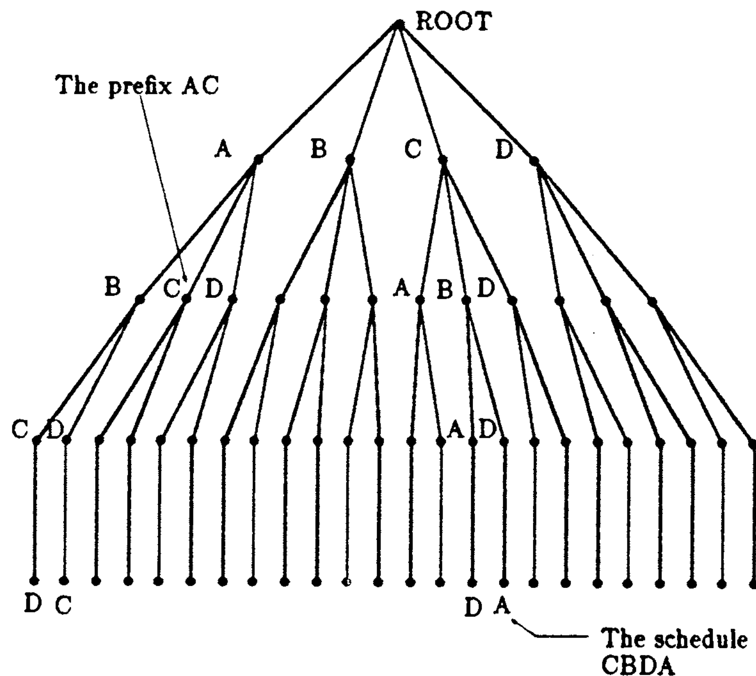


Figure 4.1: A Scheduling Tree for Four Tasks: A,B,C, & D

There are two basic strategies for limiting the search space:

- generating only the feasible schedules
- generating only the promising schedules.

The task scheduler uses a combination of both of these strategies.

Scheduling is done, in this system, by incrementally extending selected schedule *prefixes* until a complete schedule has been found. A prefix of length n is a total ordering of the tasks to be executed; starting with the first task to be done and continuing for a total of n tasks.

At each iteration through the search process the system selects the prefix that looks most promising (see section 4.7). The feasible extensions to that prefix are then generated and the process repeats. If there are no legal extensions to the prefix then that prefix is deleted from the search space. Occasionally, during the generation of the prefix extensions a flaw will be found that may be common to a whole class of prefixes. When this is the case a critic will be generated (see section 4.7.3) and passed back to the search routines instead of the prefix extensions. The critic is then run over all of the prefixes under consideration. Those that share the flaw are detected and removed from the search space.

4.2.1 The Searcher

The task scheduler uses a heuristically guided state-oriented searcher for exploring the space of totally oriented plans. This type of search strategy involves generating partial schedules and saving the state produced by the partial schedule. The partially generated schedules are then rated, and the most promising one is expanded further. The prospects are then rated again, and the most promising schedule prefix is extended. Several different prefixes, in different states of completion, may be being examined at any one time. The quality of the search strategy is judged not only on the schedule it finally selects, but also on the number of schedule expansions it does.

One common situation which could cause a search-based scheduler to perform badly is when it is given a lightly constrained set of tasks where one task is very expensive to perform. In such a situation the scheduler might expand a prefix until it reached the point where it had to schedule the expensive task. This might sufficiently lower the prefix's viability score so that some less developed prefix looks more promising. The new prefix would then be expanded until it also had to add the expensive task. In the worst case this

would continue until almost the full search tree had expanded and the expensive task was the only expansion possibility left for all of the possible prefixes. At this point the system would take the best scoring prefix and expand it with the expensive task, and then continue on. The prefix it finally chooses to make the expansion was probably the prefix it had been initially working with; all the subsequent work was just an exponential sink hole.

To overcome this problem the task scheduler only has a small fraction of the full search space available to it for backing up and improving the score. The limited search space causes the scheduler to follow its original "intuition;" the system cannot vacillate between possible prefixes because it does not have access to the list of alternatives.

The search space is constrained by putting limits on the *bushiness* of the search tree and the number of prefixes that are considered at each iteration. The bushiness, or branching factor, of the tree can be thought of as how many possible extensions there are to a given prefix. In a totally unconstrained scheduling problem there are $N - k$ possible extensions to each prefix where there are a total of N tasks to be scheduled, and the prefix in question is of length k.

The number of prefixes being considered by the search mechanism is the same as the number of leaves in the partially constructed search tree. In a poorly constrained problem of size N the number of leaves approaches $N!$. This performance can be drastically improved by using some sort of *beam search* technique as in [Lowerre 76]. *Beam search* uses an arbitrary cutoff of say ten, so that only the ten most promising prefixes are considered for extension at any time.

Unfortunately, putting limits like these on the search can cause the system to report that no feasible schedule could be found where in fact one might really exist. Limiting the overall number of prefixes being examined is especially bad in this respect, because in large problems the early level prefixes have large numbers of possible extensions, most with about the same score. In this situation, most or all of the limited number of prefixes being considered will have many early ordering decisions in common. If those decisions happen to have been wrong then the search will fail because the system had discarded the workable alternatives early on as not promising enough.

The best strategy appears to be a blend of restrictions on the bushiness of any branch in the search tree along with limits on the total search *front* — the set of prefixes that can be expanded upon. This gives the effect of performing several widely spaced beam searches each having an extra narrow beam. This strategy is preferable to a straight beam search because in task scheduling, if a prefix turns out to be unworkable it is quite likely that a

significantly different prefix should be pursued. Using a normal beam search would require a very large beam width to ensure that a satisfactory answer was eventually found. The task scheduler's search uses the limitations on bushiness and the length of the search front, along with a new system for handling worst-case backtracking. To get the best results, the size of the front must be coordinated with the allowed bushiness of the search space. As the search progresses under a normal tree search algorithm, the number of possible prefixes rises, but at the same time the bushiness decreases. However, the two are not well balanced so that the number of nodes that could be considered peaks at level $N-1$. What is needed is an algorithm that keeps the number of nodes at a smaller and more constant level, yet still keeping a wide variety of possible solutions available to the system. This can be achieved by limiting the bushiness of the tree at any level k to an amount considerably smaller than than the total number of nodes being examined. A good heuristic appears to be to set an overall limit of $max(4, \log N^2)$ prefixes to be available to be examined and a bushiness of $max(2, \log N)$. The prefixes and prefix expansions that are removed from the front by these limitations are place onto a ***backup queue***. There they can be accessed by the system should it prove that no feasible schedule can be extracted from the possibilities availble to it from the search front. The size of the backup queue can be varied to control the search time verses assurance of finding an answer tradeoff. For the experiments performed so far with the system, a backup queue size of $max(4, \log N^2)$ seems sufficient to ensure the construction of a reasonable schedule. The empirical effects of various limits are explored in Chapter Six.

4.2.2 The Schedule Dependency Array

In order to perform a heuristic search there must first exist some coherent method for selectively generating possible extensions to the search tree. It is the SDA's job to produce, upon request, all of the *feasible* daughter nodes to a particular node in the search tree. A feasible node is one that produces a schedule prefix free of domain dependent and independent problems, i.e., one that meets all of the restrictions imposed on the schedule so far. In the event that there is more than one daughter node for the prefix that the SDA is expanding, it falls upon the SDA to perform an initial evaluation upon the possible expansions and to pass on only those that appear promising.

Every prefix has an SDA. When a prefix is expanded, its SDA is copied, updated, and linked to the new prefix formed from the expansion. To find the legal daughter nodes, the SDA must provide some mechanism to distinguish those that are legal from those that are

not. For each possible prefix expansion, the SDA must be able to answer the following questions:

- Does the task being considered still need to be placed on the schedule?
- Is the current time a legal time for a particular task to be schedduled?
- Have all of the prerequisites for the task in question been placed on the schedule?
- Are other tasks being executed in this state with which this task is supposed to be disjunctive?
- Can all of the spatial and resource restrictions be satisfied?

The SDA also indicates whether the particular prefix it is working on leads irrevocably into some sort of dead end. A dead end would be where some deadline violation or resource restriction must occur no matter how the prefix is expanded. The SDA serves as an organizational device from which information on the many possible relational objects can be quickly and efficiently derived.

The SDA can be thought of as a $N \times N$ array where N is the number of tasks being scheduled. Each task has associated with it a column and a row. The elements of the array are filled with the task inter-dependencies. The columns contain all the information about what a particular task is dependent on. The rows give information on what is dependent on that task. Three vectors of length N are also part of the SDA. One vector contains the SOs. Another vector has the tasks' execution windows (see section 4.4). The third vector has each task's CSO, which is used to guide the processing of that task's row in the SDA (see section 4.5).

With the arrangement described above, the SDA can be used to easily calculate what tasks have had all of their dependencies fullfilled, and how much of a delay, if any, will be necessary before they themselves may be executed. A task has all of its dependencies fulfilled if the column associated with it is empty or has the non-empty elements (i.e., the dependencies) in their final stages of counting down their delays. If the dependency is in the countdown stage then the prerequisite has been fulfilled and it is just a matter of some temporal delay being exhausted before the task may be executed. If some element $SDA_{i,j}$ is not empty nor in the countdown stage then that indicates that task j is still dependent and waiting for task i to be put on the schedule before it may follow suit.

When a task is put into the schedule the SDA makes it a simple job to start all of the task's dependencies counting down. If task i is added into the schedule then its CSO is

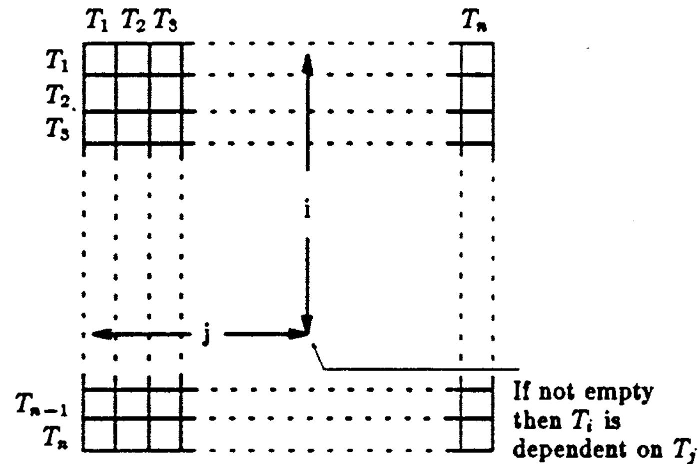


Figure 4.2: How Dependencies Are Organized in the SDA

appropriately altered. The CSO then sends an appropriate message to the task's row in the SDA. The message is passed down all of the dependencies in that row instructing them to go into the countdown state — a state where the delays between tasks start to decay.

When the SDA is being updated to take into account some amount of simulated time that has gone by, the updating process is done row by row. At each row the appropriate CSO is checked to see whether or not the dependencies in that row are in the countdown state. If they are, then an appropriate amount of time is deducted from each of the delays in that row. Relative deadline counters are also decremented. If one of the deadline counters should go negative then a *deadline* critic (see section 4.7.3) is created. All of the execution windows in the execution-window vector are updated when the SDA array is updated. The execution windows can be updated independently of the status in the CSOs.

When the *ready tasks*, those whose dependencies have all been fulfilled, are being searched for, the algorithm is: find all of the columns in the SDA whose elements are empty or in the countdown stage. For each task, whose column meets these requirements, check its SO, found in the vector of SOs, and check to see if there exists a legal state transition to that state required for the task. If a legal transition can be found then that task is passed onto the SDA's *thinning* routines. The transition time for the expansion is matched against that task's closest deadline (both absolute and relative). If the transition time is found to be longer than the time scheduled before the task's deadline would occur, an *impending-deadline* critic is formed.

In large problems with few ordering and/or resource constraints it is possible that there will be a great many *ready tasks*. The information available to the SDA, after calculating the ready tasks, is sufficient to thin down the number of possible expansions passed back to the rest of the scheduler. The SDA selects just the union of the following tasks:

- tasks with other tasks dependent on them
- the two tasks with the nearest deadline
- the two tasks with the smallest state-transition time
- the two tasks with the least time left over when their delay times are subtracted from their nearest deadline.

These particular criteria are used in order to select the tasks that are needed to avoid resource and/or dealine violations. Tasks with dependencies are always included to ensure that a relatively unconstrained task that has heavily constrained dependents is carried along far enough in the search process to at least be evaluated by the final rating heuristics (see section 4.7). Each of the above criterion has its mark placed in a data area for the tasks to which it applies. When a task is incorporated into a schedule prefix, its marks can be examined to ascertain why it was placed at that particular point in the schedule.

The SDA performs one final evaluation of the ready tasks that are left. It is quite possible that the ready task with the nearest deadline has its deadline closer than the time it would take to perform the state transition for some of the other ready tasks. If one of the tasks with a lengthy state transition should be used as the prefix expansion then a deadline violation would arise when the SDA is next updated. In order to avoid this, the SDA evaluates the remaining ready tasks, and calls a *potential-deadline* critic to eliminate those expansions that would cause such a violation.

4.2.3 Determining if a Feasible Schedule Can be found

The SDA can also be used to make an initial check to see if a feasible schedule is even possible, before any scheduling has actually been done. The process for this is similar to the methods used to reduce a determinant in linear algebra.

The initial SDA, that is formed from the original task specifications, must have at least one column with every element empty or in countdown mode. If this were not the case then that would mean that every task in the problem had a prerequisite task in that problem. If every task is dependent on some other task in the problem, then none

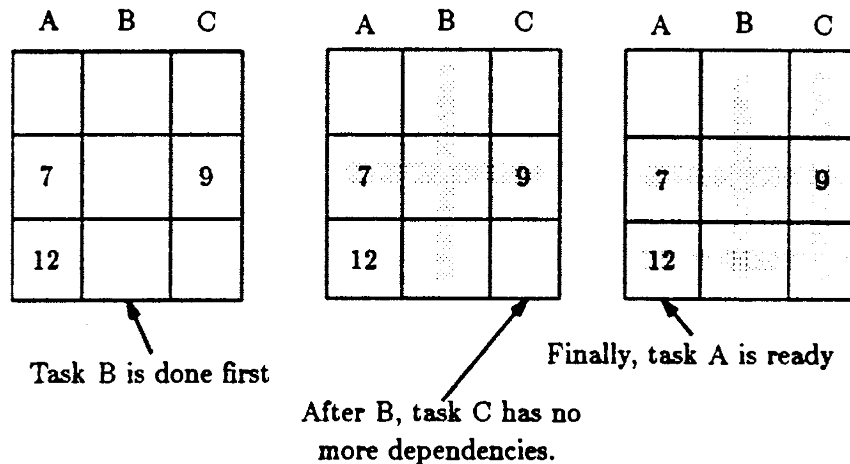


Figure 4.3: Reducing a Three Task SDA for Schedule Validation

of the tasks can ever be done. The column that is empty of prerequisite objects, and the row with the same index, can be deleted from the SDA (see figure 4.3). After the deletion operation one or more columns must become similarly *clear*. At each iteration all of the empty or countdown-only columns are deleted, and with them their associated rows. If the complete SDA (not counting the rows and columns containing the SOs, CSOs, and execution windows) is eventually deleted, then there are no contradictory or mutually exclusive ordering constraints present in the problem.

4.3 Task Orderings and Prerequisites

In real planning situations it is very common for the various tasks to be interrelated in a variety of ways. The most typical of these relations is that one task is required to precede another. For most Noah-type planners [Sacerdoti 77] this is the only type of explicit relation that two tasks can have.

However, when time is brought into the picture, additional inter-task relations can occur. These include:

- delays between tasks
- deadlines between tasks

- disjunctive sets of tasks
- conjunctive sets of tasks.

Delays between tasks are most common when two tasks are the start and finish processes of some higher level task. For example, if $task_1$ is to put a steak into the broiler and $task_2$ is to take the steak out, then $task_2$ must come at least ten minutes after $task_1$.

Deadlines between tasks often crop up in the same situations. If the person designing the task specifications does not want to risk eating a steak that has been burned beyond recognition, then it would be a good idea to put a deadline between $task_1$ and $task_2$ of about fifteen minutes. Deadlines between tasks allow the scheduler to avoid searching through the possible schedules that, in this example, would cause the steak to be removed from the broiler more than fifteen minutes after it went in. Possible schedules that have a larger delay between tasks than that specified by the inter-task deadlines cause deadline violations, and are disqualified immediately. The structure representing inter-task deadlines must be set up so that a deadline violation occurs by the point when the scheduler's time has advanced past the deadline. Waiting until the task with the deadline is put into the schedule to realize that a deadline violation has occurred could be very detrimental to the efficiency of the scheduler as a whole.

In the task scheduler, inter-task delays and deadlines are represented for each pair of tasks by a *Prerequisite Object* (hereafter referred to as a PO). A PO is associated with the dependent task (in the steak cooking example the dependent task would be taking the steak out of the broiler, i.e. $task_2$). Through the machinations of the SDA (see section 4.2.2) the PO is notified when the prerequisite task is scheduled. Once the prerequisite is scheduled, the PO starts counting down its delay and deadline counters — keeping pace with the scheduler's clock. When the delay has counted all the way to zero, the PO sends out a message that, as far as that particular prerequisite task is concerned, the dependent task is ready to be scheduled.

As the scheduler's clock continues to count down, the PO's deadline counter also continues to count down. If the deadline counter reaches zero before the dependent task has been put into the scheduler, then the PO sends a message to the SDA saying that the current schedule has failed.

Of course a PO need not have both a delay and a deadline. Some tasks and/or domains (such as the blocksworld) have no deadlines, just ordering constraints; tasks containing `PRECEDES` and `FOLLOWS` constraints.

Disjunctive task situations (e.g., not eating just before, during, or right after working out at the gym) are handled by having two deactivated POs (one associated with each task) each making its task dependent on the other. Each PO is skimmed from the appropriate IF-SCHEDULED constraint in the task's specification. Each PO has the appropriate ordering delay so that if, for example, eating lunch is put onto the schedule then going to the gym cannot be scheduled to happen less than an hour later. When one of the tasks is put onto the schedule then an appropriate message is sent down that task's row if the SDA; this message causes the correct PO to be activated. This ensures that the two tasks will not be scheduled to come too close together in time.

Conjunctive situations (e.g., schedule all your job interviews within a week of each other) can be handled in a similar manner. In this case, all the POs associated with each task are set up to be triggered by every other task in the conjunctive set. Each of these POs would have an appropriate deadline (in this case a week). Thus once the first of the tasks was scheduled all of the other tasks in the set would be under a deadline to be scheduled quickly.

4.4 Execution Windows

Tasks often have absolute temporal constraints that must be met in addition to the ordering constraints describe in the previous section. Absolute time constraints specified by MAY-START and DEADLINE constraints in the task specification are turned into execution windows for use by the task scheduler.

It is not at all uncommon to have tasks constrained by the particular time at which they are supposed to be executed. Most of the activities in a person's day are constrained in this way. When one thinks of an itinerary, usually a time-mapped schedule like that in Figure 4.4 is pictured. However, such a mapping into time is quite ambiguous. For example, what does 12:00-1:00 lunch really mean? Does lunch start at precisely noon and terminate exactly one hour later? Or perhaps lunch occurs sometime between noon and 1:00pm. If the latter is the case then how long does the actual event of lunch last? In a similar vein, does Figure 4.4 actually suggest that one is to leave work at 5:00 pm, or is that just the earliest time one could leave. Perhaps 5:00 pm is a deadline one must leave the office by; at 5:00 pm vicious guard dogs are released into the building.

What is lacking in the figure is a sense of *slop* or *fuzziness*. A preliminary task description should not specify exactly when anything should happen. *Ideal times*, as suggested in

---

6:45 get up

8:00 go to work

12:00-1:00 lunch

5:00 go home

. . .

Figure 4.4: An Itinerary of a Typical Day

Deviser [Vere 83a], try to specify the exact time at which a task should be done, but by the time other scheduling considerations come into play the ideal times must usually be ignored. What first appears, in isolation, as a good time for an event to take place may not look as attractive when brought into context with the other tasks that must be scheduled. Hence tasks in this system are assigned ***execution windows*** which represent the total interval over which the task may be done. The POs can then be used to control the length of the actual task. The inter-task dependency delays capture the fuzziness of the length of the task, while the execution windows on the endpoints allow the necessary slop for the task's placement in the time line. In this way the start and end points of a task are represented by intervals; the duration of the task is controlled by the countdown and deadline encoded in the PO linking the start and end subtasks.

Unfortunately not all tasks have a continuous interval over which they can be executed. When trying to get an appointment it is not at all uncommon for the window to be described as something like: "Come in between 9am and noon or between 2 and 5pm." However, even in cases such as this there is an interval over which the task must be executed, i.e., between 9am and 5pm. It just so happens that there is a two hour interval during which the task may not be accomplished. There is a fundamental difference between trying to schedule the task at 1pm and trying to put it at 6pm; for the first, there will be an additional hour delay reported by the task's window, for the latter, placement of the task in the schedule at 6pm will cause a deadline violation.

Thus an execution window is more than just a window, it is actually a series of non-overlapping windows in which the task can be done. The execution window is implemented similarly to a PO. The difference is that when the end of a window is reached, the delay until the next window begins is reported, and a deadline violation is not set up. When the scheduler's simulation time has passed the end of all of the windows, a deadline violation

is created if the associated task has yet to be scheduled.

4.5 Current Status Objects

There can be up to N^2 task interdependencies — delays and deadlines, for a set of N tasks. In order to make scheduling a smooth and efficient process, the system requires some method to track which of these dependencies are active and which are not. This is an especially complex problem in the case of disjunctive tasks, since the dependencies may switch about depending on how the schedule is being expanded.

To handle the task interdependency bookkeeping the scheduler uses a set of N *Current Status Objects*, further referred to as CSOs. A CSO is created for each task. The main purpose of a CSO is to alert the scheduler to which of the following states its task is in. These states include:

- *waiting*
- *schedulable*
- *finished*
- *looping*
- *evaporated*
- *failed*

When most tasks first start out they are in a *waiting* state. This means that the task is dependent upon some other task or set of tasks that have yet to be scheduled. When a task's CSO reports that the task is waiting then no further consideration is given to that task on that iteration of the scheduling process. A task's CSO is computed to be in the waiting state during the preprocessing done before the scheduling process ever begins. Such a CSO can only be moved into another state by the tasks upon which it is dependent.

After all of the tasks upon which a particular task is dependent are scheduled then that task's CSO reports that it is now in the *schedulable* state. Schedulable means that a task is not dependent upon any other task directly. It is possible that such a task may still not be worked into the schedule due to a resource conflict or other state restriction. Only tasks in the schedulable state are are checked for state restrictions and transition delays. Those that have no such restrictions will be returned to the scheduling routines as possible

extensions to the partial schedule which is currently being expanded. They are the *ready tasks*.

When one of the *schedulable* tasks is placed onto the current prefix then its state is changed to *finished*. Being in the finished state indicates that this task need not be examined or updated again. When a CSO *becomes finished* it sends out a series of messages to the CSOs that are still in the waiting, schedulable, or looping states informing them of its change in status. Any IF-SCHEDULED constraint that is satisfied by the newly scheduled task takes affect.

If a task has a MAKE-EVAPORATED activated then the task's CSO gets the status *evaporated*. For example, if a message must be sent to Tom there may be several plans for accomplishing it:

1. Go to Tom's house and deliver it to him personally
2. Write and mail him a letter
3. Go to a computer and send him electronic mail

All of these are workable plans, and it may be very difficult if not impossible to tell which will become the most expedient to actually execute when they are being examined at plan expansion time. Only as the task scheduler creates a plan is it possible to determine which plan, for sending a message to Tom, would be the best to use.[1] Therefore all of the plans are submitted to the scheduler with

```
(IF-SCHEDULED ...  (MAKE-EVAPORATED ...))
```

constraints. These constraints specify that if a note should ever come from any of these plans, specifying that it has been incorporated into the schedule, then the other two should evaporate (i.e., their CSOs should report a status of *evaporated*). Upon evaporation any tasks formerly dependent on the evaporated task are released from those dependencies.

Disjunctive sets of tasks, where all of the tasks will eventually be used in the schedule, are implemented in a similar manner. When a task which is the start of one of the disjunct intervals is worked into the schedule, its CSO passes a special message to the CSOs of all tasks which are not to overlap its interval. This message instructs the CSOs to activate the necessary POs in the SDA to ensure that the disjunctive intervals cannot overlap.

[1] See section 6.4 for a complete discussion on the task scheduler's role in the plan selection process.

A status of *looping* has the combined effects of the finished and schedulable statuses. The looping status indicates that the task has been placed onto the schedule at least once but must be placed onto the schedule at least one more time. The POs dependent on a looping task start to countdown much as if the task had been finished, but the looping task remains active to be updated by the SDA as necessary. The looping status figures prominently in the *loop copy* process described in section 5.3. It is the CSO that eventually changes the status of the task from *looping* to *finished* when a sufficient number of iterations have been incorporated into the schedule. Looping is the subject of the next chapter.

Whenever an intertask deadline violation occurs, a message is sent to the task's CSO. In such a case the CSO reports a status of *failed*. This serves as a flag to the scheduling routines and causes the partial schedule under consideration to be discarded. The particular task which caused that failure is kept track of for debugging purposes.

4.6 Scheduling Windows

The output of the scheduler is an *itinerary* where an itinerary is a list of tasks and events, and the windows and states under which they should take place. The states described in the output are those defined by the various task's SOs, as worked out by their state transitions.

The windows produced by the scheduler differ in several aspects from the execution windows described in section 4.4. Most notable among the differences is that the scheduling windows are made up out of three time values rather than two. These three times are:

- The *lower* time bound
- The *safe* time bound
- The *upper* time bound

Each of these times represents a different aspect of the schedule with regard to the execution time of the task. The times can also serve to aid in execution monitoring.

The schedule is guaranteed to succeed if the *lower* bound is followed. This bound is calculated using the minimum time spent for accomplishing state transitions and the minimum execution time for each task upto that point in the schedule. The lower bound refers to the earliest time that can be planned on to start a task, assuming that all the tasks preceding this point have taken as little time as possible.

The *safe* time bound, coming out of a scheduling window for a particular task, represents the last time that the task can be started and have the schedule remain valid. If a task is executed at a time later than the *safe* time then it is impossible for the remainder of the tasks to be executed without a deadline violation occuring.[2] The *safe* time limit is calculated from the total ordering by propagating the *lower* limit backwards from the various deadlines incorporated in the schedule.

Finally, the scheduler also produces a pessimistic view of the execution world. In this case the scheduler uses the *upper* bounds on the tasks' execution times and state transitions. The longest times for a task's execution time and state transition are added onto the minimum of the previous task's *safe* and *upper* bounds.

Just because execution times remain below the *upper* bounds specified in the schedule does not have any bearing on whether or not the execution is proceeding in a manner that concurs with the schedule. *Upper* bounds serve only as warnings as to how long a task or transition could possibly take; only the *safe* limit can be used to monitor whether or not a schedule is remaining valid. Places where the *safe* limit comes before the *upper* time bound indicate places where the schedule is likely to fail. These are the prime spots for the insertion of monitor tasks and contingency plans.

4.7 Schedule Rating Heuristics

While the algorithms for scheduling tasks have been covered in detail, there still remains the question of how the schedule searcher decides which prefix, and which expansion for that prefix, the system should be working on. The answer to that question is that the system maintains a rating of how promising each prefix and expansion is. What that rating is based on and how it is calculated make up the remainder of this section.

4.7.1 Rating the Schedule So Far

At any point in the scheduling process, it is easy to find out how much time and how many tasks have been scheduled for a particular prefix. With these numbers it is trivial to calculate the average time scheduled for each task. Assuming that the average task's time

[2]This of course assumes that one of the tasks remaining to be executed does not take substantially less time to be executed than was marked in the task specification. Similarly for the amount of time spent executing the state transitions.

remains constant throughout the remainder of the scheduling process allows the system to estimate the total time required for the schedule, if the number of tasks to be scheduled is known. Unfortunately, the number of tasks to be scheduled is often not well known when only a part of the schedule has been decided upon. The number of iterations for some loops (as described in the next chapter) may yet to have been calculated. If decisions on exclusive disjunct task sets have not been scheduled then they also can contribute to the task number uncertainty. However, estimates can be made of the number of tasks remaining to be scheduled, and the average task time can be improved upon by sampling the delay times stored in the SDA. These can help to improve the quality of the ***Estimated Total Time***, or ETT for short.

The ETT can give some estimate of the quality of its own value by tracking its change during the scheduling process. If two prefixes' ETTs are the same, but one prefix's ETT has decreased every time its prefix has been expanded, then that prefix should get a higher score (since its ETT has a history of being pessimistic). So the score associated with an ETT is a function of both the value of the ETT and the trend of the ETT over the last several prefix expansions.

The ETT's reliability (and likewise its overall weight in the prefix's rating) depends heavily on the proportion of the ETT that has already been scheduled. In other words, a prefix that is almost a complete schedule should have its ETT play a more important role than a prefix that has only a couple of steps in it.

The score based on the schedule-so-far is a function of how much has been scheduled and how much the system thinks there is yet to schedule. Additionally, the rating is modified by whether the estimates it is based on are believed to be conservative or optimistic. If the estimates are believed to be conservative (based on the trends of the ETT) then the score associated with it would be raised; if optimistic then the score would be lowered. The rating based on these time estimates helps to distinguish markedly different prefixes from one another.

4.7.2 Rating Possible Prefix Expansions

Choosing the best prefix expansion for a particular prefix relies on information that is local to the possible expansions, and therefore mostly provided by the SDA. The particular factors that play a role in the rating of expansions are:

- the expansion's closest deadline

- the expansion's state-transition delay
- the expansion's prerequisite delays
- the tasks that are dependent on the task in the expansion
- the task's place in the overall set of state transitions.

The way that these factors interact with one another can also effect the expansion's score.

If a task is placed onto the schedule at a time such that its prerequisite delays will run out simultaneously with its state-transition delays, then the robot will spend little or no time idle. The task will assuredly be done before any deadline violation can occur. If the task is scheduled so that its prerequisite delays are larger than the transition delays then the robot will spend that difference being idle — a sign of an inefficient schedule. This is wasted time, and is detracted from an expansion's score.

In most scheduling problems a better solution can be found if the scheduler has freedom on how to order the tasks to be performed. One set of factors that limit the scheduler's freedom are the prerequisite constraints. Prerequisite constraints can keep a large percentage of the tasks unavailable for scheduling if their prerequisites have yet to be fulfilled. For creating a near optimal schedule, and for meeting certain deadlines, it is desirable to schedule the prerequisites as early as possible in a schedule. Therefore, the number of tasks for which a particular expansion's task serves as a prerequisite has some effect on that expansion's score.

The relationship between an expansion's state-transition delay and that task's nearest deadline is an important factor in an expansion's score. If the state-transition delay takes up almost all of the time before the task's deadline, then that expansion has a higher priority than an expansion that has a large gap between its nearest deadline and transition delay. The reason for this is quite simple: an expansion that contains little slack between the end of the state-transition and the task's deadline can probably not afford to be delayed any longer. Those expansions with a wide disparity between their deadlines and transition delays have more leeway, and are more likely to be postponable without an adverse effect. Expansions whose state-transition delays take up the exact amount of time before the task's deadline comes up form a ***delay-deadline*** critic, to ensure that they are tried as the next, and usually only (see section 4.7.3), expansion.

State-transition delays for different tasks can fluctuate greatly depending on the order in which the tasks are done. To take this into account the ***resource tour time***, or RTT, is figured into each prefix's overall rating. The RTT can be thought of as the length of a generalized

traveling salesman tour; the time between "cities" is measured as the state-transition delay found while converting from one state to another. Unlike traditional traveling salesman problems, the distances between *cities* are neither commutative nor associative, due to the nature of state transitions. It is possible that not all RTTs are possible due to resource conflicts among the tasks.[8]

An RTT is calculated for each prefix expansion with the constraint that the first "city" to be visited is the state associated with the task to be added to the schedule by that particular expansion. The other stops on the tour are the other ready tasks. Figure 4.5 shows the RTT scores for three schedulable tasks whose only resource is the robot's position. For this simple example each task's RTT reduces to calculating the shortest route for visiting every workstation starting with the one needed for the task. The result is then normalized. The figure gives the RTTs for each of the three tasks.

The RTTs are used for scoring purposes, rather than the individual transition delays, because the RTTs are immune to certain types of situations where scoring only on transition delays would produce a very bad schedule. An example of a situation where RTTs produce better scores that transition delays is shown in Figure 4.6. The figure presents a problem where the robot must visit four workstations: A, B, C, and D. The workstations are positioned so that ordering visits based upon the shortest transition delay will actually produce the most time-consuming schedule. The RTTs suggest a schedule that is considerably more efficient.

4.7.3 Final Scoring, Re-Scoring, and Critics

The factors that go into rating a prefix and expansion have been explored in the sections immediately above, but the question remains: How are these factors combined into a final score? The answer to that question is not as simple as one might hope.

Different scheduling problems have different scoring needs. A relatively lightly constrained problem can have an optimal score produced by paying close attention to state-transition delays, waste time, and getting prerequisite tasks scheduled as quickly as possible. Deadline-intensive sets of tasks need more attention paid to minimizing waste time and the relationship between prefix expansion deadlines and their state-transition delays, in order

[8]A resource conflict among the ready-tasks is not indicative of a serious resource conflict. It is quite likely that a task that was not yet ready, or was thinned from the list of ready-tasks, will supply the necessary resources so that all tasks can be scheduled.

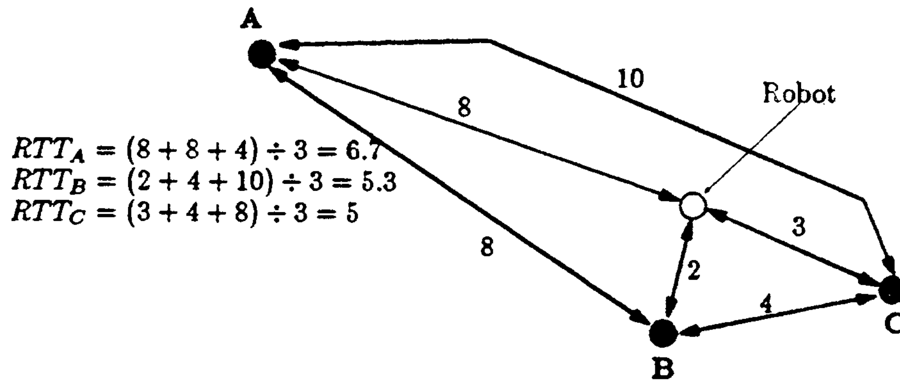


Figure 4.5: Calculating Three RTTs for Three Ready-Tasks

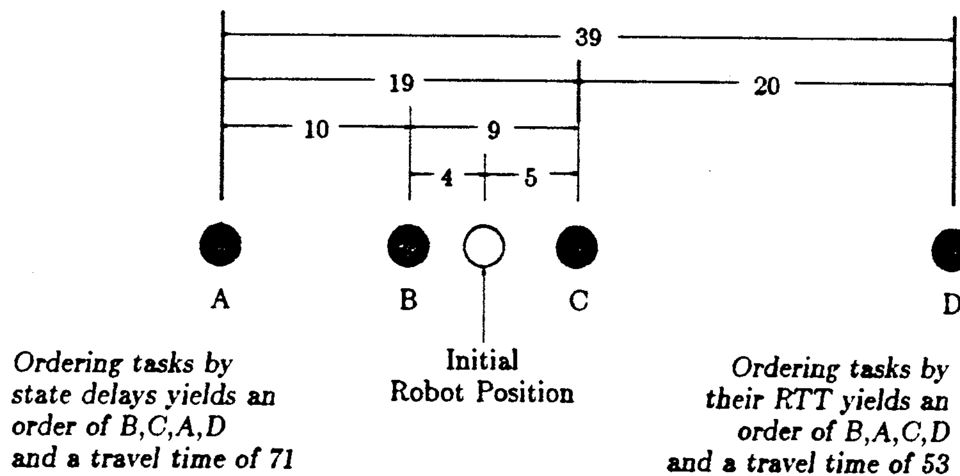


Figure 4.6: Why RTTs Are Used Instead of Transition Delays

to just find an answer. Resource-intensive tasks must pay more attention to domain-specific scoring factors, and must make sure to get prerequisite tasks scheduled as early as possible.

Unfortunately, many problems have part of their schedules being unconstrained, while other parts are resource and/or deadline intensive. Since there is no one scoring system that works for all problems, the task scheduler uses a flexible system which adapts itself to the problem being worked on.

Scoring factors are initially balanced towards producing a maximally efficient schedule.

As the scheduling process proceeds, the system may try to expand a prefix that leads to a dead end. When that dead end in encountered it is analyzed and an appropriate critic is constructed. A dead end can be one of three types:

Update Deadline: a failure that is spotted by the SDA when the time required for the most recently scheduled task is added into the SDA and one of the tasks remaining to be scheduled is forced past its deadline. A *Deadline* critic is created in response to this failure.

Impending Deadline: a failure that is uncovered when the SDA is comparing ready tasks' deadlines to their state and prerequisite delays. An *Impending Deadline* critic is created to handle these situations.

Resource: a failure that results because of there being no legal state transition from the prefix's last state to any of those states of the tasks that are otherwise ready. A *Resource* critic is made up in these circumstances.

These types of dead ends, and their associated critics, are used to reset the balance in the score of schedule prefixes and their possible expansions. The ways in which the scoring are changed are shown in Table 4.1. The critics then go through each of the prefixes and possible expansions in the search front, and rescore them.

Heuristic	Absolute Deadlines	Relative Deadlines	Resource Restrictions
Task Weight	+	-	+
Deadline	+	+	
RTT	+	-	
Domain			+
No Wait		-	-
Waste Time			-

Table 4.1: Scoring Changes Due to Scheduling Dead Ends

Each time a dead end is encountered the scoring is altered. The effects are cumulative, but not long lasting. If several impending deadline failures are encountered, then the scoring is going to be heavily shifted in order to avoid future failures of that type. However, it is quite possible that the part of the schedule that is likely to cause difficulties in that direction has been or will very soon be completely scheduled. In order for the scheduler to produce efficient schedules the scoring strategy must revert back to the efficiency-directed strategey with which the system started out. Scoring changes decay if they are not reinforced. The scheduler is always trying to revert to an efficiency-oriented scheduling system.

In order to reduce the number of failures encountered of the various deadline varieties, two other critics are commonly constructed:

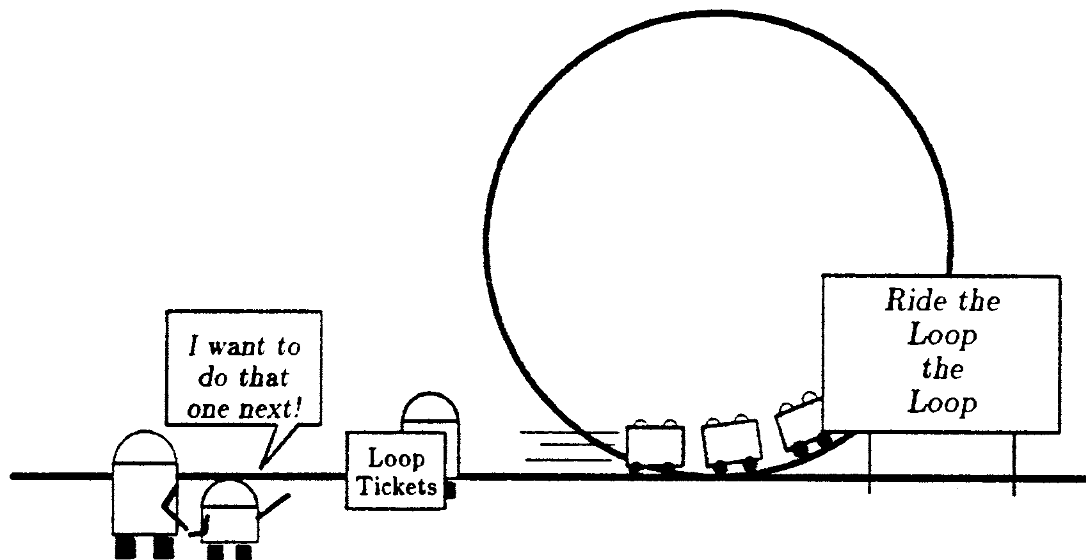
Potential Deadline: a critic used when some of the ready tasks have state transitions that are longer than the nearest deadlines of some of the other ready tasks.

Delay Deadline: a critic that is produced when a ready task's transition delay takes exactly as long as the time scheduled before its nearest deadline.

Neither of these critics causes the mass re-scoring that is initiated by the other critics described above. Instead, these critics eliminate some of the expansions for the prefix. The Potential Deadline Critic removes all of the expansions for a prefix that have state transitions taking longer than the time until the nearest deadline of any of the expansions. This is to avoid the Update Deadline failure that would certainly result if one of the expansions eliminated were to be used. The Delay Deadline critic removes all of the prefix expansions, except for the one that cause the critic's appearance. Since that task's state-transition delay will take up all of the time before its deadline, it must be scheduled next — otherwise it will surely violate its deadline. The other possible expansions are saved for the somewhat rare cases where the scheduling of one of them would cause the task that formed the critic to evaporate.

The flexible scoring system outlined above, when combined with critics and and the structured search provided by the SDA, lead to efficient schedules produced with a minimum of search.

Chapter 5



Scheduling Loops

5.1 Introduction

In many domains some or all of the tasks to be performed by the robot are designed to be repeated several times. The number of times the repetition is to be carried out, and the way that number is specified, varies from problem to problem. The role of the scheduler, when encountering a repeated or *looped* set of tasks, is threefold:

1. Provide a mechanism for specifying and coordinating loops with the other tasks to be scheduled.
2. Work the loop smoothly into the schedule so that the complexity of calculating the schedule does not rise exponentially with the number of iterations to be scheduled.
3. Recognize when the schedule itself is in a loop and modify the output of the scheduler to reflect the observation.

The ability to handle loops is a major factor that allows the task scheduler to produce efficient solutions for some difficult problems; for example:

The task for the robot to accomplish is to paint ten widgets using the widget painting factory shown in Figure 5.1. The time it takes the robot to get from one workstation to another is proportional to the distances between the workstations. The painter has a reservoir that can hold several cans of paint; a can of paint will paint two widgets. At the start the robot is at the painter, which is empty. The robot has a manipulator that can carry one item; the hand is also empty. The robot must stay at the painter while it is painting a widget.

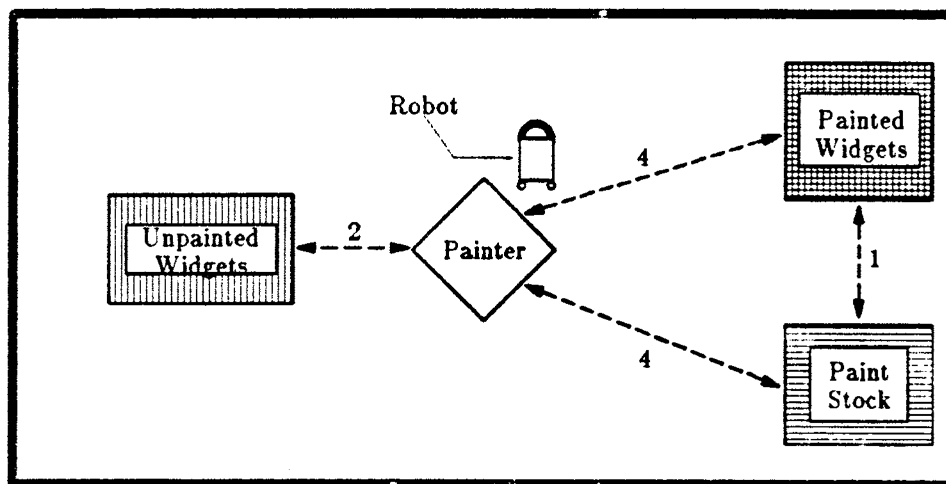


Figure 5.1: The Widget Painting Factory

The partial order for the problem described above is shown in Figure 5.2. The partial order is the type of output that could be expected from a Noah-type planner. The partial order does not contain the information needed providing any type of travel advice for the robot. This particular problem would cause special difficulties for a Noah-like system. The replicate nodes in Noah allow the ten widget-painting episodes to be expressed by only two nodes in the task network (only two nodes are needed for any number of widgets greater than one). However, Noah can only use the replicate nodes when each iteration has identical effects. In this problem, a single can of paint will paint two widgets — some iterations of widget painting require a paint refilling subtask and some do not. It is therefore doubtful that Noah could get the prerequisites correct wile using the replicate nodes.

The task scheduler avoids this problem in two ways. First, loop tasks are handled by the planner one iteration at a time; the planner never sees a thousand widget painting

tasks, but instead, a single painting task — a thousand times over. Secondly, when the schedule starts to repeat[1] the scheduler designs a new loop that can be directly executed by the robot. This condenses the thousands of instructions the robot might have to follow into a simple repeat loop.

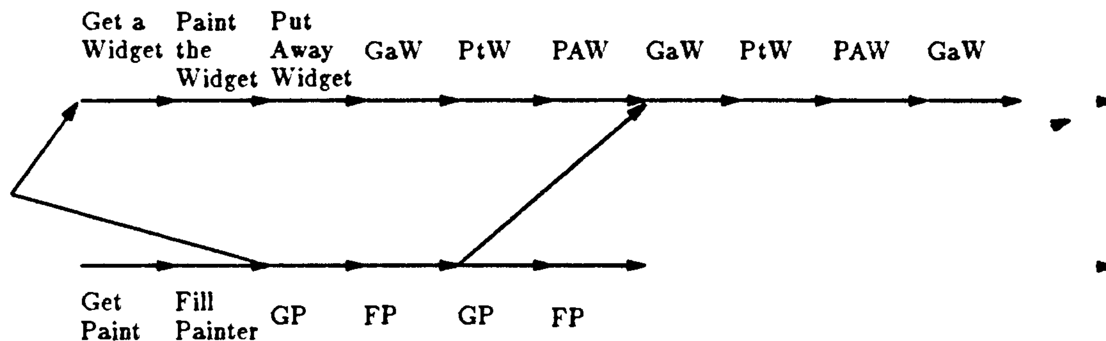


Figure 5.2: The Partial Order for Painting Ten Widgets

This chapter first explains how ordinary loops of just a few iterations are handled by the task scheduler. Later, the methods for recognizing when a schedule is in a loop, and the way in which the robot repeat loops are constructed, are discussed.

5.2 Scheduling Repeated Tasks

In the context of scheduling, a *loop* is defined as:

Loop: A set of tasks which are to be put on the schedule multiple times. Before any task in the set may be put onto the schedule N times all other tasks in the set must be on the schedule at least $N - 1$ times.

Repeated tasks can be broken into three general looping situations:

- Loops that are to be repeated N times (e.g., do ten situps).
- Loops that are to be repeated until event P occurs, where event P is predictable at plan time (e.g., patrol the bank until closing time).

[1]For example, after a few widget paintings and refilling of the paint reservoir the same sequence will be repeated over and over again in the schedule.

- Loops that are to be repeated until event E occurs, where event E must be observed at execution time (e.g., play golf in thunderstorms until lightning strikes you).

The only thing that distinguishes the situations above are their termination conditions. It is necessary that the second two situations, those that are event driven, be transformed, to some degree, into loops that terminate after a pre-specified number of iterations. The scheduler can only make resource and time estimates if it has some idea how many iterations of a loop are to be done. Without these estimates, the scheduler has difficulty choosing times to schedule tasks that are not in the loop. This is especially relevant for the last of the situations shown above, where the loop termination conditions are not well specified until execution time. If the situation is left unchanged then there is no way that the scheduler can know if sufficient time will remain in which to do any of the tasks that remain on the schedule.

All loops are handled by the scheduler in the same way, regardless of their termination conditions. This section covers the machinery that is used to integrate a loop of fixed number of iterations into a schedule. Later in the chapter, examples will be given to show how these methods work on loops with more poorly defined termination conditions.

The major mechanism used for implementing repeated tasks is the CSO. As mentioned in section 4.5, each CSO contains a status flag which indicates to the scheduling routines whether that particular task is ready for scheduling, has been scheduled, or is waiting for a dependency to be scheduled first. Whenever this flag is changed by the scheduler (e.g. from *schedulable* to *finished*), several messages can be passed from one CSO to another along with messages being passed to the various POs. These messages can cause significant changes to be made to the SDA. Most common among these changes are that:

1. The CSO for the scheduled task is changed to reflect its new status.
2. The POs that were dependent on the scheduled task are placed into a countdown state.

Additional changes can be made to the SDA to handle loops. The exact changes that take place depend on whether the loops are:

1. a repeated set of totally ordered tasks, or
2. a repeated, unordered set of tasks.

Whenever a set of tasks are known to form a loop they are run through a special preprocessor before being handed off to the scheduler. If the set of tasks in the loop are totally

ordered, each of the tasks, except for the first, already has associated with it a PO making that task dependent on the task which is supposed to precede it. When preparing a set of tasks to be scheduled as a loop, the tasks are first run through the loop pre-processor which replaces each of those POs with a *Resettable PO*, or RPO for short. An RPO has all of the features of an ordinary PO, but also has some additional abilities. An RPO has a quiescent state which produces no countdown delays or deadlines but does keep its associated task from being scheduled, independent of the statuses shown by the CSOs. In addition, an RPO can be reset to its full countdown state i.e., the state where its delay counter and relative delays are at their initial values. With these modifications the RPOs do not "fade away" after the first iteration of the loop has been completed. After an RPO's task has been placed onto the schedule the RPO goes into its quiescent state. When the task on which the RPO is dependent is placed onto the schedule again, in the next iteration of the loop, then the RPO is reset to its initial values to make sure that its task is placed onto the schedule within the proper relative time frame.

The CSOs are also modified by the pre-processor in order to incorporate the counters needed to keep track of how many iterations of their tasks have been placed onto the schedule. Finally, the loop pre-processor inserts an additional RPO which makes the first task in the loop dependent on the last, thus closing the loop. This RPO is referred to as a *cycle-RPO.* The cycle-RPO is not actually placed into the SDA until the starting task of the loop has been executed for the first time (at the point where the first task's CSO changes status from *schedulable* to *looping*). It then appears in its quiescent state and is reset to an active countdown status when the last task in the loop is placed onto the schedule. It is necessary to keep the cycle-RPO out of the SDA until its task has been executed once, in order to insure that the loop can actually be started.

When the set of tasks in the loop consists only of a single task then the cycle-RPO is attached to that task; the task becomes directly dependent on itself. Everything else remains the same as in the multiple task loop.

The *Widget Painter* problem described above provides a good example of how the scheduling of totally ordered loops is accomplished. When the problem first reaches the scheduler the SDA is in the state shown in Figure 5.3. Two tasks can be put onto the schedule: the robot can either do get paint, or get widget. If the latter is chosen the scheduler will soon hit a dead end; there will be no paint in the painter and the robot's hand will be occupied holding the widget. The system would then backtrack to this point and choose the get paint task. At this point the SDA will be as shown in Figure 5.4; note

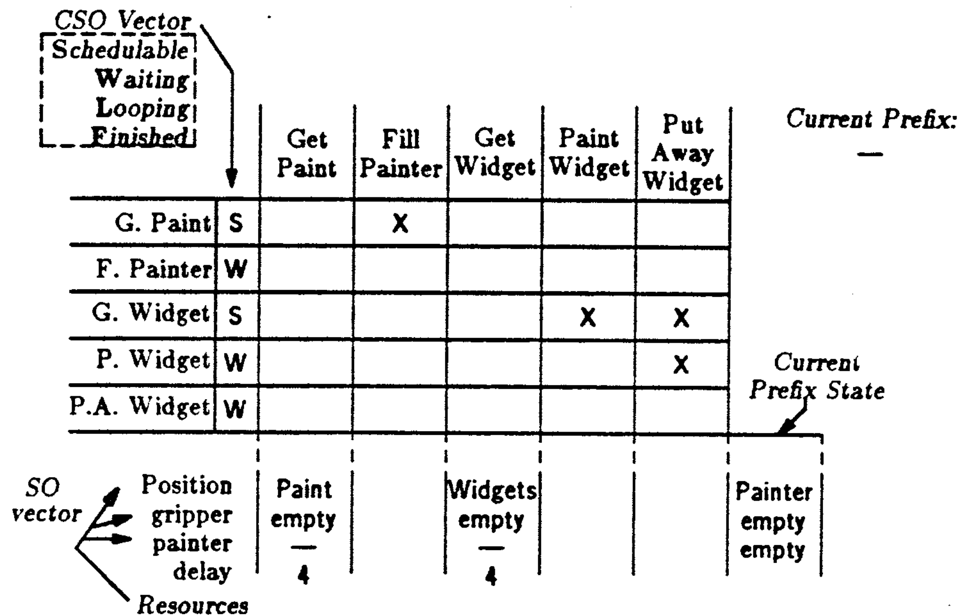


Figure 5.3: The SDA for the Widget Painting Factory

that the cycle-RPO making the get paint task dependent on the fill painter task has become active. Also, the CSO for the get paint task has changed to *looping*. The looping status indicates that the actual status of the task must be determined by checking all of the POs in the task's column of the SDA. If all of the POs are in countdown mode then the task is *schedulable*; if one or more are in the quiescent stage, then the task is *waiting* on some other task in the loop to be put onto the schedule.

In Figure 5.5 the SDA is shown after the fill painter task has been placed onto the schedule for the first time. There is now sufficient paint in the painter to cover two widgets so the scheduler can choose to begin the widget painting loop. Optionally, it could put another can of paint into the painter. However, the widget painting loop will be started since it receives a better score from the heuristic searcher (i.e. it is making more efficient use of the robot's position). Once an unpainted widget is retrieved by the robot neither of the two fill painter tasks can be put onto the schedule[2] until the painted widget is dropped off in the stockroom. When the widget is dropped off at the stockroom, the highest rating

[2]Because no transition path can be found by the paint widget task's SOs that will allow them to be placed onto the schedule while the robot is holding onto a widget.

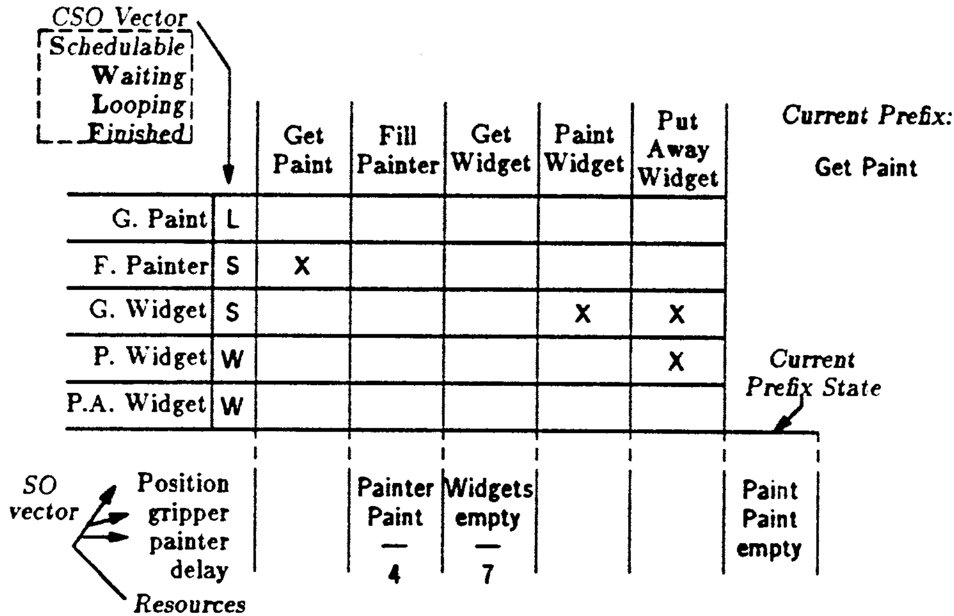


Figure 5.4: The SDA With the Robot Holding a Can of Paint

task that is schedulable will be the get paint task, since the robot will be located so close to the paint stock. This schedule will repeat until five widgets have been scheduled to be painted and all five cans of paint have been scheduled to have been put into the painter. At that point the SDA will appear as shown in Figure 5.6. The scheduler will then continue until the painting of the remaining five widgets have been painted.

The scheduling of unordered loops is done in a similar fashion, but with a few differences. When a loop is unordered it means that the set of tasks that comprise the loop do not have a total ordering among them. While some POs may exist that relate one of the steps in the loop to other steps, in many cases these POs will not be sufficient to define unique first and last tasks in the loop. Without definite first and last tasks there does not exist a task to which the cycle-RPO can be assigned that will close the loop and still guarantee that the tasks will be scheduled in the form of a loop (i.e., that all the tasks in the loop will be scheduled for N iterations before any task is put onto the schedule for $N+1$ time).

This problem is gotten around by having the loop pre-processor add a ***cycle-task*** to the set of loop tasks. A cycle-task is a no-op task to which the cycle-RPO is attached. The cycle-task consumes no resources and does not change the state of the world in any way; it

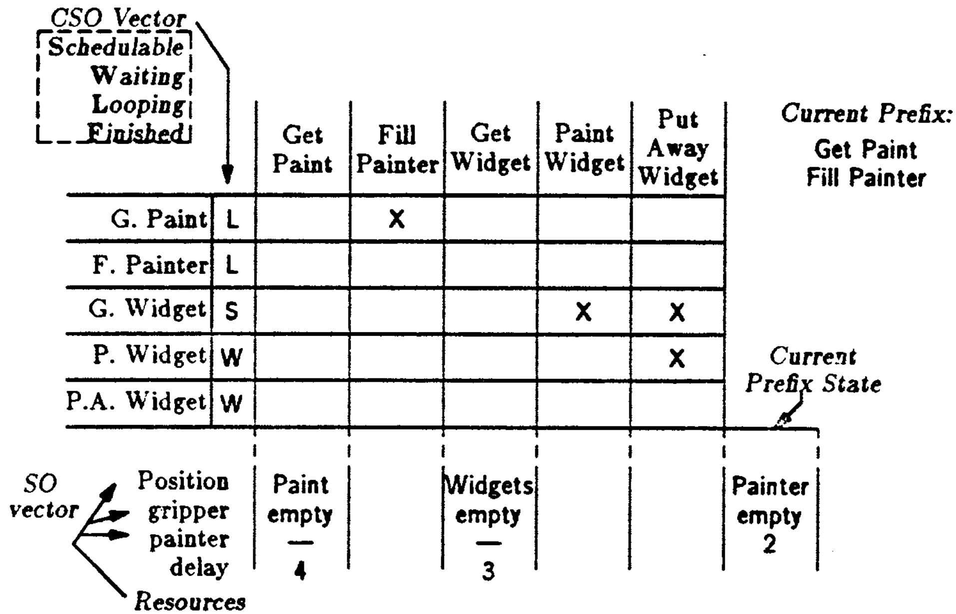


Figure 5.5: The SDA After *Fill Painter* has Been Done Once

serves only to aid in the scheduling process.

The cycle-task is initially made dependent upon all of the other tasks in the loop. As each task is placed onto the schedule its RPO goes into the quiescent state that makes that task dependent on the cycle-task. When all of the tasks in the loop have been put onto the schedule the cycle-task may be placed onto the schedule. Once the cycle-task is on the schedule then the RPOs on all of the other tasks are fullfilled and they may each be placed onto the schedule another time. No task can be placed twice on the schedule before each task in the loop is on once because as soon as a task goes onto the schedule it becomes newly dependent on the cycle-task, but the cycle-task is still dependent on all of the tasks in the loop that have yet to be placed onto the schedule. The series of SDAs shown in Figure 5.7 give the progression of dependencies for a loop consisting of two unconstrained steps.

Through the use of the loop preprocessor and RPOs the scheduler is able to handle loops of any length while searching a schedule space that is at worst proportional to the number of iterations in the loop, rather than exponential. The RPOs make sure that no more than one iteration of the loop is visible to the scheduler at a time. With a sufficiently large

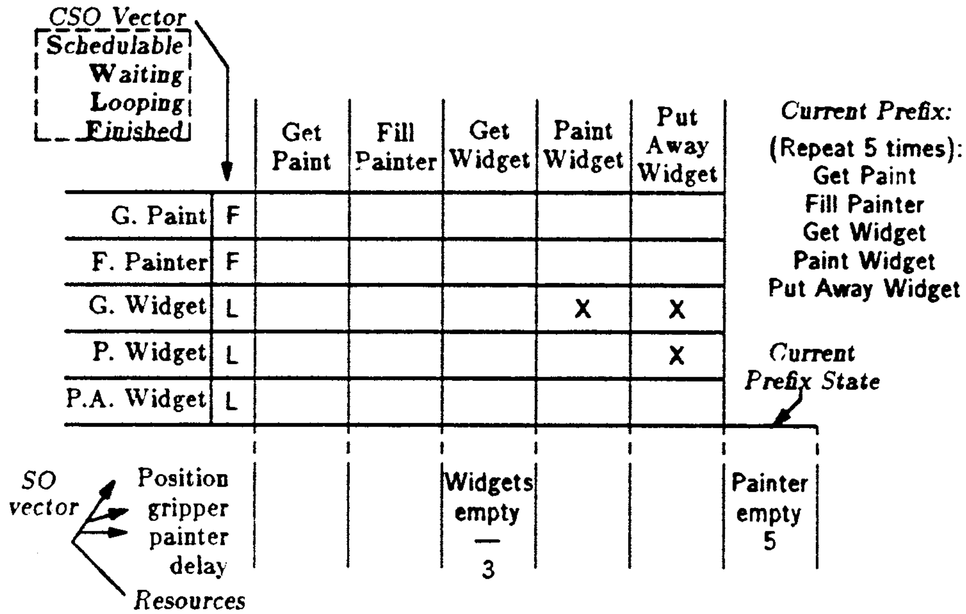


Figure 5.6: SDA When the Painter has All the Paint

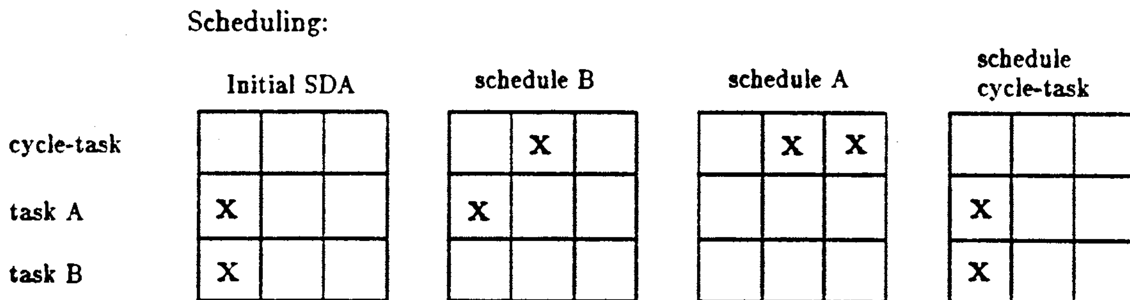


Figure 5.7: The Role of the *Cycle Task* in Unordered Loops

number of iterations this can still be prohibitively expensive to compute. Unfortunately, loops with large numbers of iterations are not at all uncommon. Even worse, loops that have termination conditions that may only be absolutely determined at execution time cannot be handled in this way; for the scheduler to schedule each step of each iteration it must know how many iterations to do. To handle problems of this sort two things are

needed:

1. A way to schedule loops that is more efficient then having the scheduler calculate every step of every iteration
2. A way to make reasonable estimates on how many iterations of a particular loop will be needed in order to have the robot accomplish a particular task.

Solutions to both of these problems are explored in the next section.

5.3 Condensing Loops

There are many problems involving repeated tasks where the exact number of iterations to be done is not well specified in the task description. For example:

- Do situps for ten minutes
- Paint widgets until dinner
- Look right every two feet traveled, until the end of the hallway is reached.

Unfortunately, the scheduling algorithms described so far require the number of iterations to be performed to be known exactly in order for the system to be able to produce an itinerary. However, it is not really necessary for the itinerary that the task scheduler produces to state every instance of every task explicitly. The system needs only to produce a schedule that can easily be executed by the robot. This section is in two parts: the first presents an algorithm for speeding up the scheduling of loops by creating a program loop for the robot to execute that incorporates the part of the schedule that needs to be repeated. The second part of the section discusses the heuristics that are used to make estimates on how many iterations of the repeated tasks (and hence the program loop) will need to be scheduled.

Figure 5.8 shows the schedule that would be produced by *Bumpers* for the *Paint six widgets* example given in the previous section. This is an obviously repetitive schedule and thereby is taking up more memory in the robot and time to compute then may actually be necessary. What is desired is a schedule that looks more like the one shown in Figure 5.9. Note that the schedule in the figure is a combination of the two loops given to the scheduler. It was constructed by noticing a pattern in the schedule as it was being constructed in the normal, linear fashion. To notice and condense a pattern of this sort the scheduler must know three things:

Task:	Robot Status Position:	Painter:	Hand:
Get Paint	paint stock	empty	paint
Fill Painter	painter	2	empty
Get Widget	widget stock	2	widget
Paint Widget	painter	1	widget
Put Away Widget	painted widgets	1	empty
Get Paint	paint stock	1	paint
Fill Painter	painter	3	empty
Get Widget	widget stock	3	widget
Paint Widget	painter	2	widget
Put Away Widget	painted widgets	2	empty
Get Paint	paint stock	2	paint
Fill Painter	painter	4	empty
Get Widget	widget stock	4	widget
Paint Widget	painter	3	widget
Put Away Widget	painted widgets	3	empty
Get Widget	widget stock	3	widget
Paint Widget	painter	2	widget
Put Away Widget	painted widgets	2	empty
Get Widget	widget stock	2	widget
Paint Widget	painter	1	widget
Put Away Widget	painted widgets	1	empty
Get Widget	widget stock	1	widget
Paint Widget	painter	empty	widget
Put Away Widget	painted widgets	empty	empty

Figure 5.8: Schedule for Painting Six Widgets

1. When should the scheduler look for a repeated pattern?

2. What constitutes a pattern that may be repeated?

3. For how long can the pattern be maintained?

Deciding when to check if a schedule is being repetitive depends on the type of loop (ordered or unordered) being worked on. For loops with totally ordered steps a check should be made every time the first step in the loop becomes *schedulable* (except, of course, for the very first time). At the point when the first step becomes *schedulable* for the second time, the entire loop must have already been placed at least once onto the schedule. Therefore, the amount of time needed for an iteration of the loop, as well as the resources needed, can be calculated from what has already been scheduled.

Task:	Robot Status		
	Position:	Painter:	Hand:
Repeat 3 times:			
Get Paint	paint stock	----	paint
Fill Painter	painter	+2	empty
Get Widget	widget stock	----	widget
Paint Widget	painter	-1	widget
Put Away Widget	painted widgets	----	empty
End ----	painted widgets	3	empty
Repeat 3 times:			
Get Widget	widget stock	----	widget
Paint Widget	painter	-1	widget
Put Away Widget	painted widgets	----	empty
End ----	painted widgets	0	empty

Figure 5.9: Schedule for Painting Six Widgets

When a loop is unordered a schedule cannot be considered to be in a repetitive state until all the steps in the loop have been placed onto the schedule two times. When the cycle task becomes schedulable for the second time the schedule can be checked for being in a loop. Unordered loops must be on the schedule twice in order to ensure that their scheduled order is going to remain constant for a while. In an unordered loop the order that the steps are scheduled for one iteration does not necessarily reflect on the ordering for future iterations. Even in situations where the loop is the only task being worked on by the scheduler, orderings can change. For example, in Figure 5.10 a task is shown where three workstations must be visited by the robot repeatedly in a loop. When the robot starts out it is at workstation **A** and therefore visits **A** first. It then goes on to **B** and **C**. At the second iteration the robot is starting out at workstation **C**, and so visits that one first. On the third iteration the robot is starting at **B** but finishes at **A**. So it is not until after three full iterations of the loop that the schedule goes into a repetitive state.

In addition to establishing a point in the scheduling process where the schedule should be checked for repetition; it is necessary to establish the rules for deciding whether or not a part of the schedule is in fact in a repetitive state. This process, referred to as *loop extraction*, is attempted every time a loop check is done. Loop extraction starts by first going through the schedule so far produced, and checking if there is a valid, stable loop that can be excised from the schedule. The criterion being looked for in the schedule by this process are:

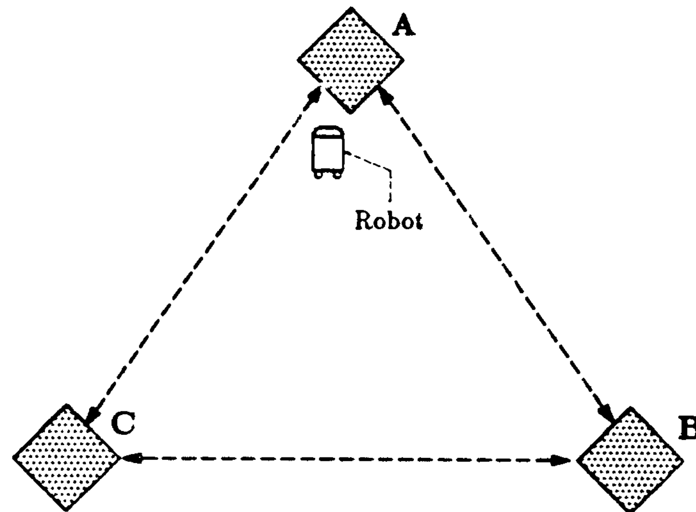


Figure 5.10: Three Unordered Workstations to be Visited

1. No *one-shot* tasks
2. No *loop fragments*
3. Identical state delays closing the loop

The first of these criteria is to ensure that tasks which are not supposed to be repeated do not get inserted into a piece of code which will be repeated. One-shot tasks are just ordinary tasks that are not a part of any loop. If a one-shot task is scheduled to occur between two steps in a loop then that part of the schedule cannot be extracted and used for the purposes of loop condensation. A piece of the schedule containing the scheduling of a one-shot task is unique in the schedule and cannot be repeated.

Loop fragments are steps from a loop that is independent of the cycle-task that prompted the loop extraction attempt. It is not uncommon, in domains such as mobile-robot navigation, for two or more loops to be interleaved with one another. If the various loops do not run at exactly the same pace it is very possible that all the steps in a complete iteration of one loop may be scheduled while only a handful of steps have been scheduled for the other loops. Rewinding the schedule until a full iteration of one loop has been uncovered may therefore only contain a few steps from the other loops. If this portion of the schedule was then repeated some of the loops being scheduled would have only a fraction of their steps repeated over and over again; the rest of their steps being scheduled fewer times, or not at

all. An example of this is shown in Figure 5.11. In the figure three repeated tasks of three steps each are being scheduled. The schedule so far generated shows that `loop1` has been scheduled in total two times, while only two steps of `loop2` and a single step of `loop3` have been placed onto the schedule. This schedule prefix is not sufficiently developed to extract a portion for loop condensation.

```
1. loop1-step1
2. loop1-step2
3. loop2-step1
4. loop1-step3
5. loop3-step1
6. loop1-step1
7. loop2-step2
8. loop1-step2
9. loop1-step3
```

Figure 5.11: Two Iterations of Loop1 Containing Fragments of Loops 2 & 3

During loop extraction the schedule is rolled back task by task. If a one-shot task is encountered then the loop extraction is terminated unsuccessfully. If a step from another loop is encountered then a list of all of the steps in the loop (except for the step encountered) is formed. The schedule unwinding continues. Every time a step from another loop is encountered during the unwinding, it is checked to see if it is any of the lists that have been formed. If it is then it is deleted from those lists. If the step encountered is not on any of the lists, and is not part of the loop that triggered the loop extraction, then a new list is formed. This process continues on until the cycle-task that triggered the loop extraction is encountered again in the schedule. When it is encountered a check is made to make sure none of the loop fragment lists have any steps left in them. If there are still fragments of the other loops then the process continues until the cycle-task is encountered again on the schedule. When all of the lists are empty and the cycle-task is encountered, one of two checks is made. If the loop in question is an unordered loop then a check is made to test whether at least two iterations of the loop have been unwound and that the first and last iterations are identical with respect to the ordering of the loop steps. If they are not then the schedule is unwound further; if they are identical, then that portion of the schedule is

extracted for further processing.

If the loop being unwound is totally ordered then the check that is made involves the cycle-tasks of every loop that has been unwound. In Figure 5.12 two totally ordered loops of three steps each have been unwound as keyed by the cycle-task for **loop1**. In this example, **loop1** has had one iteration unwound while **loop2** has had two. Loop extraction was not successful after either iteration of **loop2** was placed onto the schedule because those iterations contained fragments of **loop1**. The extraction will be successful at the present time iff the maximum PO delays and transition delays for the cycle-tasks for each loop in the unwound schedule (i.e., **loop2** in addition to **loop1**) are identical at the begining point of the unwound schedule (at the point after the scheduling of **one-shot1**) to what they are at the point where the loop extraction was initiated (the point after the scheduling of **loop1-step3**).

---

1. **one-shot1**
2. **loop2-step1**
3. **loop2-step2**
4. **loop1-step1**
5. **loop2-step3**
6. **loop2-step1**
7. **loop1-step2**
8. **loop2-step2**
9. **loop2-step3**
10. **loop1-step3**

Figure 5.12: Loop Extraction Triggered by Cycle-Task of **Loop1**

The state and PO delays must be checked for all of the cycle tasks in the loops that have been unwound in order to ensure that the spacing of a contained loop is not altered through loop extraction and condensation. If two loops were being scheduled, each having only one step, where one loop was spaced every second and the other every five seconds, then the schedule shown in Table 5.1 could result. Without a check on the delays on each cycle-task a loop extraction containing schedule steps 5 and 6 would look perfectly legal. However, such an extraction would alter the timing of one of the tasks drastically. Such

an extraction would cause both tasks to have iterations at one second intervals. By not allowing a loop extraction until the delays match, the extracted loop will contain schedule steps 1 through 6, thus ensuring the proper timing.

Schedule Step	Task	Delay-1	Delay-2
0	one-shot	1	5
1	loop-1	1	4
2	loop-1	1	3
3	loop-1	1	2
4	loop-1	1	1
5	loop-1	1	0
6	loop-2	1	5

Table 5.1: SO and PO Delays During Loop Extraction

Once a successful attempt has been made at loop extraction, much is known about the loop that is being condensed: the tasks in the loop and how much time and resources that are required in order to complete an iteration. Yet the question still exists: *How many times should this part of the schedule be repeated?* For the majority of problems the answer lies in the SDA. Since the time for an iteration of the repeated part of the schedule is known, the number of times it should be repeated is bounded by two possibilities:

1. The maximum number of iterations that should be made by any of its component loops
2. The amount of time that can be scheduled before something "interesting" is scheduled to happen.

The first of these bounds is rather straightforward to calculate. Attached to each task's CSO is a counter which gives the number of iterations that task is to have scheduled. For each step in the extracted schedule fragment a quick check can be made to find the minimum number of iterations to be performed.

The second bound is slightly more complex to calculate. Something "interesting" in the scheduling process can be one of several events:

- An absolute deadline violation occurs
- A relative deadline violation occurs

- A task's window and prerequisite delays decrease below its state-transition delay.

The first two "interesting" events are fairly straightforward; it would be foolish to extend the schedule via loop condensation past another task's deadline. The remaining class of event is more subtle, but just as critical. When a task is schedulable and has a short transition delay and a long prerequisite delay before it can be placed on the schedule, then it makes sense to try and schedule other tasks before it. The reason for this strategy is simply that it is inefficient for a state transition to be performed a considerable period before the task for which the transition was done, can itself be executed. However, if the state-transition delay is longer than the other delays on a task, then that task's possibility of being placed next on the schedule should be considered very carefully. If a prerequisite deadline follows closely after the prerequisite delay (as is often the case) then the optimal point where a task should be scheduled is when the PO delay is equal to that shown for the required state transition — the task should be scheduled so that the robot will arrive in the position necessary for executing the task the moment the task becomes executable. While these *delay-delay* points in the schedule are not always critical, they are important sufficiently often that they should not be scheduled past during loop condensation. Fortunately these points are easy to calculate in most instances.[8] The maximum PO delay for each *schedulable* task is calculated. From these values are subtracted the corresponding state-transition delays. The minimum value calculated gives the time to the next "interesting" time of the delay-delay type. This can then be compared with the earliest relative and absolute deadlines to find the amount of time through which the schedule fragment can be repeated.

5.4 Loop Termination

The loop scheduling algorithms just described handle problems where the exact number of iterations of the loop is well defined. They also work well for problem situations where a set of tasks are to be repeated until some event, under the control of the task scheduler, occurs (e.g., schedule study time repeatedly until the exam is placed onto the schedule). Because loop condensation will halt whenever the optimum scheduling time for a new task rolls around, loops whose termination is keyed to some other task will terminate at the

[8]In some domains it is possible that either the SO and/or PO delays will be a function of the tasks in the part of the schedule which is being repeated. In such instances "interesting" points in the schedule cannot be accurately predicted. For these domains loop condensation cannot be done, and the schedule is produced one task at a time.

proper point in the schedule. However, these algorithms do not do the complete job in problems where a loop's termination is not completely predictable at schedule time.

Problems where loop termination is controlled by an event sensed at runtime are fairly common in most domains dealing with low level control of robots. Since these are the domains that Bumpers is primarily concerned with, some explanation is due on how the system handles them. First, let us review an example problem of this sort that was first brought up in section 1.2.4. The problem being referred to is when a mobile robot is been given the top-level task of rolling down the hallway shown in Figure 5.13 and turning left at the second left-hand corridor. The robot in question has a single steerable distance center that it must use to make steering corrections while navigating the hallway and also for spotting the right hand corridors. If the map is built from earlier observations by the robot then it contains some uncertainty on the length of the hallway before the branching corridors. Because of this uncertainty, and the uncertainty in the robot's ability to measure distances, it is not possible to accurately determine in advance:

- How long it will take the robot to make it to the second corridor
- How many steering corrections will be needed for the robot to traverse the hallway
- When the robot should start looking for the branching corridors

The best that can be hoped for is to get some estimate on the expected values of each of the parameters in question. These estimates can be determined via the spatial representation system that the map is based on (see section 2.3). This system allows calculation of all these parameters along with an indication of their reliability. These estimates are necessary ingredients in the calculation of the termination points for the loops scheduled to solve the hallway navigation task.

Assuming that the estimates, with their error bounds, are accurate, then the lower bound on the length of the hallway is the shortest distance the hallway could be. Traversing a length of hallway equal to the lower bound will ensure that the robot has yet to reach the point where the left-hand corridors branch off. Likewise, traversing a length of hallway equal to the upper bound will ensure that the robot has reached (and probably passed) the branching corridors. The trip down the hallway can therefore be broken into three parts: rolling through the straight part of the hallway, the section where the corridors branch off, and the section past the branching points.

When the robot is at each of the different parts of the trip it should have different activities scheduled:

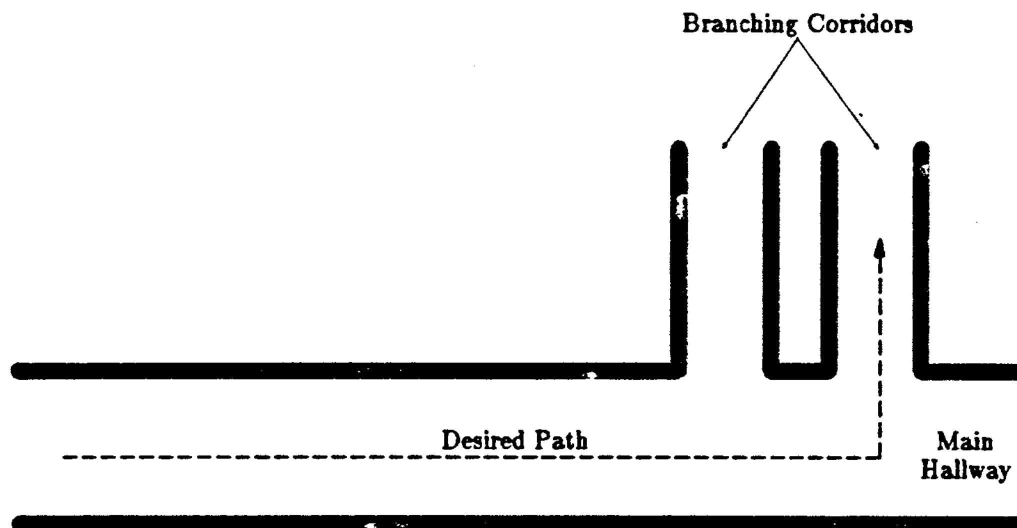


Figure 5.13: Map of Corridors for Mobile Robot to Navigate

1. When the robot is in the straight section of the hallway it should just be looking to the right; gathering sufficient observations so that in can successfully navigate the straight hallway.

2. As the robot approaches the part of the trip where it will be passing the branching corridors it should start making observations to the left so that it may spot exactly where it should begin its left-hand turn.

3. If the robot reaches the part of the hallway where it knows, from its earlier estimates, that it must be past the branching corridors to its left, then it should execute some contingency plan. For the robot to have gone this far without making a left-hand turn means that the original plan has failed in some way. There is no point in continuing on looking both left and right — for that strategy was based upon assumptions that the system based on its internal representation of the area; either the representation, or the robot's model of how it executes actions has now been proven to be wrong. A corrective measure is in order.

In problems such as the hall-walking scenario the problem gives adequate information to determine bounds on the time and/or number of iterations that a portion of the schedule should be repeated. In the hall walking example the first loop (observing the right wall and making steering corrections) should have enough iterations scheduled to move the robot to the earliest point in the hallway where the branching corridors might occur. The second loop (continue to make steering observations and correction additionally, look left occasionally in order to spot the left-hand corridors) should have enough iterations scheduled for the

robot to cover the entire stretch of hallway where the corridors might be located. The code produced for the second loop must have the additional termination condition that the loop is stopped when the right hand corridors are spotted. This condition is specified to the scheduler initially by saying that the look for left-hand corridors task is to be repeated until the observe left-hand corridor event is placed onto the schedule. Since the transition delay for this event will be an interval that corresponds to the time necessary for the robot to cover some or all of the region of the hallway, the correct amount of uncertainty will be propagated through the remainder of the schedule.

Chapter 6

Experiments

In order to test out the theories that have been presented in this dissertation, a task scheduler has been implemented which matches the description given in this dissertation. Many different types of test problems have been run using the task scheduler. The tests on coordinating loops have been detailed throughout this dissertation (see sections 1.2.4, 1.3.2, 2.6.1, 2.7, 3.1.2, 5.3, and 5.4). Other experiments were needed to fully test out the scheduler — a large number of scheduling tests with different types of constraints and different numbers of tasks. Towards this end, hundreds of experiments were performed using randomly generated data. In addition, special scheduling tasks were hand created in order to test out particular aspects of the system that had not been stressed in the other tests. This chapter contains descriptions and results of some of the experiments that have been carried out on the task scheduler.

6.1 Scheduler Implementation

The task scheduler was implemented in Nisp [McDermott 83], a portable dialect of the Lisp programming language. The SOs and POs were all implemented as *instances* of *objects* (see [Goldberg 83] for a good explanation of this type of object-oriented programming). The state objects all have the same referencing protocols, but different instances can perform their calculations in different ways. For example, when calculating whether a state transition was possible, an SO for a task that involved only the position of the robot's head would need only check to make sure there were no restrictions against turning the robot's head to the desired position; a task that required the robot's body to be in a specific location

might have an SO call a route planner to make sure that the robot could reach its desired location from the location of the previous task in the schedule prefix.

For some calculations (such as calculating delay intervals) the SOs and POs have identical referencing protocols — though the calculations performed by the objects may differ markedly. This use of objects allows the SDA to contain both POs and SOs, and to treat the two types of objects identically in most instances. The CSOs can pass messages to a row or column in the SDA and have all of the objects in that row or column perform the appropriate action — whatever it may be.

The CSOs pass messages to the POs and execution windows in order to control their countdown state. The POs and execution windows can also pass messages to the CSOs — when deadline violations crop up. To facilitate the message passing among the objects the CSOs are stored in the same data-structure as the rest of the SDA.

The SDA is implemented as a *dynamic sparse-array.* This means that only those elements of the array that are **not** empty are actually contained in the data structure, but new elements can be added to the array at any time (a nice feature in domains where the task interdependencies can change markedly depending on the state of the schedule prefix). Since the absolute maximum number of interdependencies that can be active at any one time is $(0.5N)^2$, the sparse-array offers a significant improvement in storage and SDA update time over the full N^2 array.

Figure 6.1 shows the heuristic rating function used by this implementation of the task scheduler. All of the terms with a suffix of `-factor` are the dynamic weighting-factors by which the particular properties of a prefix expansion are multiplied. These factors change during the course of the actual task scheduling (the initial values of these factors are given in Table 6.2). The terms in the figure with a prefix of `task-` are properties of the prefix expansion that are used by the rating routine to determine the expansion's score.

Table 6.1 shows the amount of code used in each portion of the scheduler. All of the routines have been compiled. This code includes the libraries of routines for constructing the SOs, POs, and CSOs needed for describing a particular task. The libraries are totally domain-independent. For each of the problems described in this chapter, only a couple hundred lines of code were needed for special state functions, domain-dependent scoring routines, and the task descriptions. The domain-dependent code for a problem is about 5% of the size of the scheduler. New domains and problems can be given to the task scheduler without having to make significant changes or additions to the code.

```
(schedule-default-score-function
     (- (+ (cond ((and (= 0 task-state-delay)
                       (= 0 task-prerequisite-delay))
                  (* zero-state-delay-factor task-rtt))
                 (t 0))
           (* domain-factor task-domain-specific-score)
           (* task-weight-factor task-number-dependents)
           (* -1 rtt-factor task-rtt)
           (* waste-time-factor
              (cond ((> task-prerequisite-delay
                        task-state-delay)
                     (* 2 (- task-state-delay
                             task-prerequisite-delay)))
                    (t 0)))
           (cond ((= task-deadline task-state-delay)
                  9999999)
                 ((> (* 3 approaching-deadline-factor)
                     (- task-deadline task-state-delay))
                  (* approaching-deadline-factor
                     (- task-state-delay task-deadline)))
                 (t (* (sqr approaching-deadline-factor) -3))))
        (* 10  prefix-time-to-finish)))
```

Figure 6.1: The Task Scheduler's Heuristic Rating Function

6.2 Randomly Generated Scheduling Tests

A large number of tests were performed on the scheduler using randomly generated data. The tests were used to tune and check the efficiency of the scheduling heuristics. These problems were all variations on the basic errand-running problem. A number of workstations, at various locations, were generated. A path between those stations was chosen at random. Deadlines were created for visiting each of the workstations. The deadlines were based on the amount of time necessary to travel to each of the workstations in the order defined by the randomly chosen path. A small amount of time was added to each of the deadlines in order to give the scheduler some choice in picking an ordering. On some of the tests, dependencies, delays, and deadlines between the workstation visits (based on the chosen path) were also added. The workstations and their constraints were passed to the task scheduler in order for the scheduler to find an ordering in which to visit the workstations.

By basing the constraints attached to the workstation visits on the randomly chosen path, at least one ordering that met all of the constraints was known to exist — the random ordering on which the constraints were based. The two percent additional time added to

Routines	*Lines of Code*
Utilities	691
Search	340
Internal Data Structures	181
Control & Loops	710
Scoring	301
SDA	572
Prerequisite Objects	94
State Objects	1015
Critics	100
Bumpers Domain	1095
Randomly Generated Tests	173
Baseball Domain	211
Vehicle Domain	246
Forbin Domain	287
Total	6016

Table 6.1: The Size of the Task Scheduler Routines

the deadlines was necessary for giving the scheduler some choice in designing an ordering. If the deadline on each workstation provided just enough time to travel to that workstation then the scheduler would be forced to choose the ordering that was randomly generated — and in choosing that ordering it would only be exercising its *nearest-deadline* heuristic. The small amount of slop time allowed other orderings to be explored making it possible for the scheduler to find a more efficient ordering, or to fail to find an acceptable ordering at all.

The randomly chosen path among the workstations usually bore little resemblance to the shortest tour through those stations. However, since the constraints enforced on the problem were derived from the randomly chosen path, it is quite likely that the shortest tour would not provide an acceptable solution (i.e., the shortest tour will often result in some deadline or precedence violation).

In order to find an acceptable and efficient solution to the test problems it was necessary for all of the scheduler's domain-independent heuristics to play a significant role. In general, the task scheduler's solutions to these problems was not dominated by any single aspect of the problem; no one heuristic directed the scheduler's choices. The state-transition-delay

heuristics and the deadline heuristics had to reach compromises with one another in order for an efficient solution to be found. The heuristics dealing with wasted time and task weight also had a role in the ordering of the workstation visits.

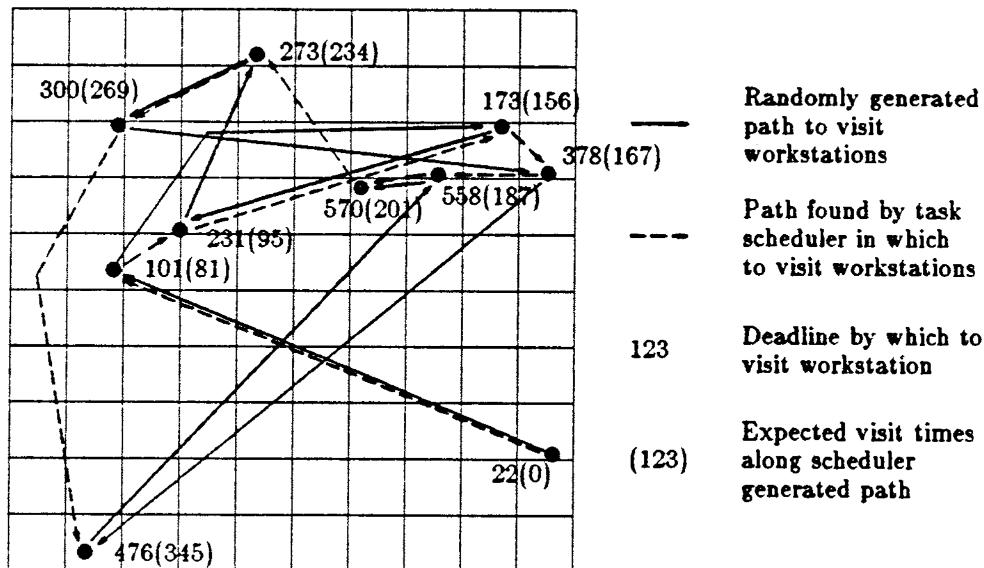


Figure 6.2: A Typical Randomly Generated Workstation Problem

Figure 6.2 shows a typical test problem on which the task scheduler was run. Workstations were randomly placed on a 100 × 100 matrix. The robot was initially positioned at the location of one of the workstations. A path to all the workstations, starting with the workstation at which the robot was located, was randomly generated. Deadlines are set up in accordance with the generated path. The time before a deadline for visiting a particular workstation was based on the amount of travel needed to get the robot to that workstation along the generated path. The robot did not have to spend any time at the workstation, it only had to visit it. To make the problem slightly more difficult for the task scheduler, all of the slop time in the deadlines was moved to the deadlines of first workstations in the randomly generated path. When the scheduler first starts looking at the problem there is large amount of slop on the tasks with the near deadlines. As a solution is developed, the slop time on the tasks with the later deadlines diminishes. In other words, if the scheduler chooses a path identical to that which was randomly generated, the first task will have 20% slop time while the last task will have a deadline equal to the amount of travel time required to reach the workstation.

The generated deadlines and scheduler-calculated visit times for going to each workstation are given in Figure 6.2. The schedule found by the task scheduler is about 30% shorter than that of the randomly generated path.

None of the workstation visits in this problem have any prerequisites so the schedule search tree contains over 10! nodes for the ten workstations in the problem. To find a solution the scheduler must search through a minimum of ten nodes in the search tree. The solution for the problem in Figure 6.2 required a search through thirteen nodes of the search tree. This result is a little better than average. Over several hundred problems of varying sizes the schedules produced by the task scheduler required a search of between N and $5N$ nodes, where N is the number of workstations in the problem. There was no significant relation between the number of workstations in the problem and the number of the nodes searched.

There was a significant relationship between the number of workstations in the problem and the efficiency of the solution found: the larger the problem the more efficient the solution. The solutions found for problems of a hundred workstations were typically between 30% and 40% of the length of the randomly generated path. Problems of ten workstations averaged around 70% of the randomly generated path.

Rating Factor	*Value*
Task Weight	8
Approaching Deadline	14
RTT	11
Domain	13
Zero State Delay	100
Waste Time	8
Decay Amount	1
Decay Delay	2

Table 6.2: Initial Values for Rating Factors and Their Decay Rates

Randomly generated problems were used to come up with the initial values for the weights on the scheduling heuristics (see Table 6.2). The decay factors were also derived and tested on these problems. The scheduler would be run in a debug mode on a test problem. At each iteration its weights would be juggled, by hand, until the proper prefix expansion was chosen. After several problems were run in this way, the scheduler was rerun

on all the problems to eliminate any irregularities and biases that had been introduced. Finally, the scheduler was run in its fully automatic mode on the same and additional problems — in order to make sure that the empirically derived heuristics really worked.

Once the scheduling heuristics were selected, the size of the search front and the branching search-factors had to be chosen. When the factors were too small, the scheduler sometimes fails to find any solution to the problem. When the search factors were assigned larger values, the scheduler explores the search tree in more of a breadth-first manner. The chosen values for the search-factors were the smallest that the scheduler could take and still find solutions for over 99% of the randomly generated problems. The value chosen for the maximum number of prefixes avaiable for expansion was $max(4, \log N^2)$; the maximum branching factor was $max(2, \log N)$. While these factors have been sufficient for the scheduler to find acceptable solutions to all of the planning problems on which it was tried, it should be noted that insufficient tests have been performed on very large problems (e.g., $N > 100$) for it to be certain that these values are optimal. The results for some of the randomly generated workstation problems are shown in Figure 6.3. The Figure shows that the search time for a problem is proportional to the size of the problem being worked on. On average, a portion of the search tree equal to two and three-quarters times the size of the problem will be examined before a solution is chosen by the scheduler.

This performance is not affected appreciably by the number of other constraints placed on the problem (e.g., delays, dependencies, and deadlines). Constraints can cause many of the scheduler's initial attempts to be unsuccessful, but the thwarted attemts serve to narrow the search space considerably.

The actual time required by the system is not quite as well behaved as the number of nodes the system searches. On a Vax 750, fifty-one seconds of CPU time were required to solve an average problem of ten workstations. For twenty workstations, 190 CPU seconds were needed. The performance continued on this slightly less tha N^2 track for the other randomly-generated problems it was tested on. There are at least a couple of possible explanations for this behavior:

- The search through the SDA, while nct N^2 (because of the sparse array), is not necessarily linear. As N grows, and more dependencies occur, a larger part of the full SDA must be examined.
- The current implementation is not very space efficient, and on larger problems quite a bit of time may be spent allocating storage and swapping pages.

If the SDA proves to be the real problem, then some better heuristics may be needed

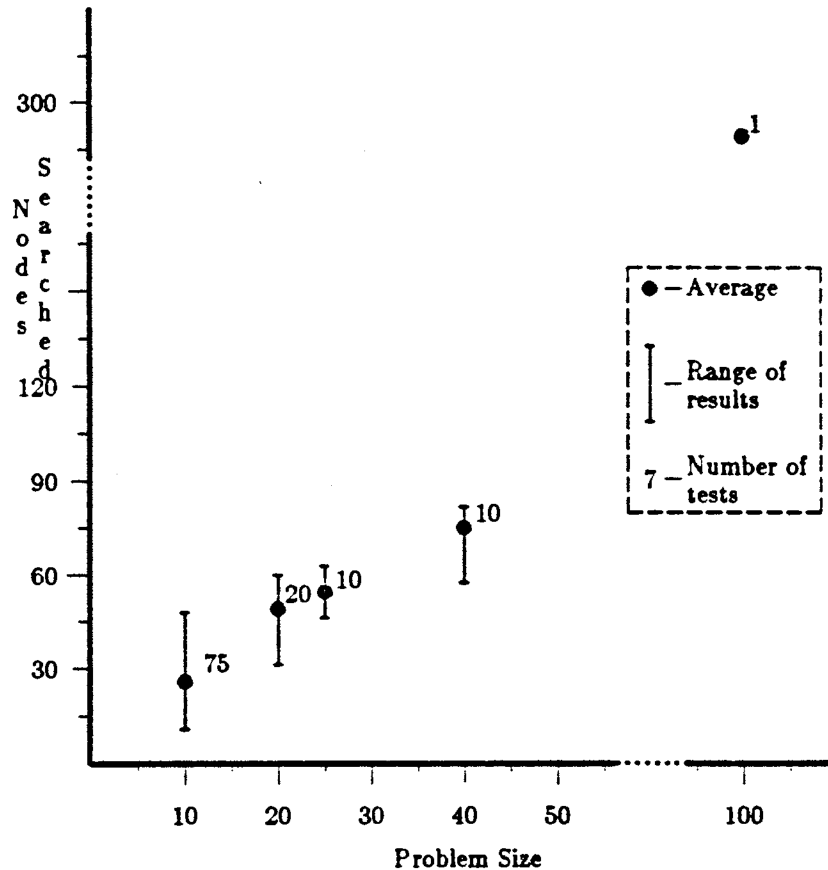


Figure 6.3: Problem Size vs Scheduling-Tree Nodes Searched

for searching and updating the SDA. Currently, every column in the SDA, except for those associated with finished tasks, is examined when the array is being updated. In many domains, the behavior of the POs is sufficiently predictable that the portion of the SDA examined for each prefix expansion could be reduced to those elements in rows whose tasks have been scheduled and columns that have unfinished tasks. The current implementation copies the full SDA at every prefix expansion. This copy includes the SO and CSO vectors. Not all of this information truly needs to be copied, and a future implementation could be more space efficient. If the operating system of the computer is causing the increase in CPU time (through paging), then a larger computer would solve the problem.[1] The CPU

[1] A larger computer would be desirable in any case.

times probably result from a mixture of these causes.

6.3 Additional Scheduling Tests

6.3.1 Unusual State Changes: Vehicles

By isolating the physics of the world in RESOURCE-PHYSICS statements and SOs, the task scheduler is able to handle changes in the physics in a transparent manner. One common change in physics is brought on through the use of vehicles. If a robot can jump into its car any time it wishes then it can accomplish tasks and meet deadlines that would be impossible without the vehicle.

The task scheduler has been given some problems that involve the use of vehicles. A typical problem is one that involves dropping a car off at the garage for repairs. Any tasks that are attempted while the car is being repaired must be done on foot. The robot is given several other tasks to perform. Some have their workstations located near the garage, others are further away. If the robot starts doing the tasks located near the garage it is possible for it to work itself a considerable distance away from the garage by the time the car is ready to be picked up. If it is too far away, a large amount of time can be wasted walking back to the garage. If the robot is too conservative then it will waste considerable time waiting for the mechanics to finish with the car.

The actual tasks given to the scheduler are:

- Get the car muffler replaced at "Glacial Muffler Services" (see the map in Figure 6.4). The robot need only bring the car in and pick it up after it is finished. It will take between 120 and 150 minutes for the garage to do the job. The car must be picked up soon after it is ready, otherwise this disreputable firm will probably cut it up for parts.
- Do the laundry at the Laundr-o-Mat. This involves putting the laundry in the washer (a task that takes two to five minutes), transfering the clothes to the dryer (another task requiring two to five minutes, and one that must follow the first by at least forty minutes, but not more than seventy), and folding the dry clothes (a five to ten minute task taking place at least one hour after the clothes are placed in the dryer, but no more than 90 minutes). If the robot is walking on its own, its speed is slowed by 2mph if it is carrying the laundry.
- Go food shopping at the "Protein Warehouse." This requires between twenty and thirty minutes and slows down the robot's ability to travel by 2mph, unless the robot is using the car.

- Mail some letters at the Post Office — a task that takes between two and five minutes (depending on the size of the line for buying stamps).

- Go home (the same place where the robot started out) within five hours of leaving. All other tasks must be accomplished before this one may be done. The robot's initial walking speed is 5mph and its driving speed is 30mph.

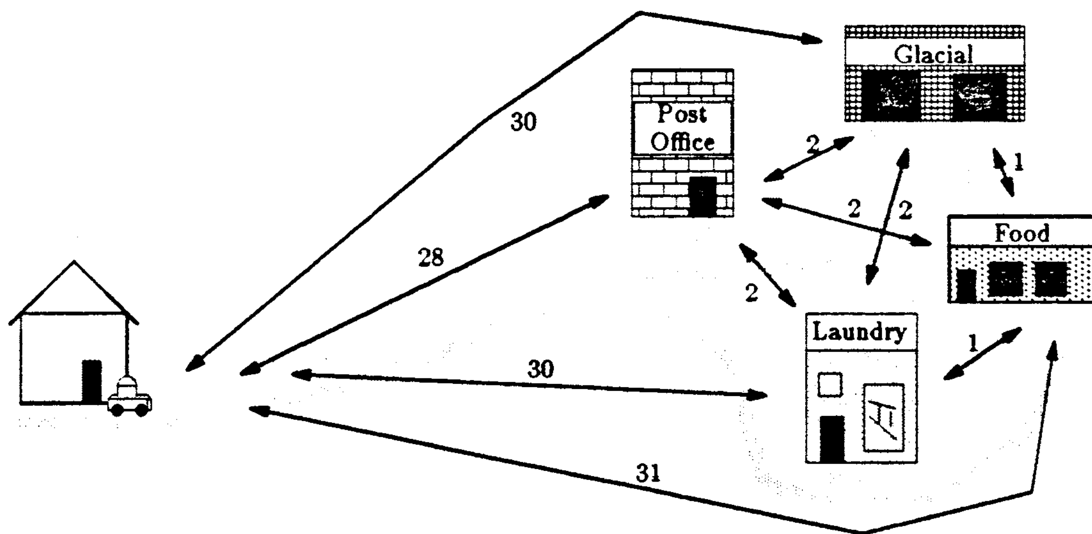


Figure 6.4: Distance Map for Car Repair Problem

This is an unusually hard problem for the scheduler to find a solution. The reason that it is more difficult than most is that the estimates that it gets for the RTT (average route travel time) are less accurate than in most problems. The inaccuracies arise from the different speeds the robot will be traveling at as it switches vehicles or becomes impeded by what it is carrying. Also, the deadlines and delays are quite intricate; it would be difficult to just sit down and come up with an acceptable answer to this problem. While this problem contains many ordering constraints, the scheduling tree still contains several hundred nodes. The task scheduler searched through thirty-five nodes and used 169 CPU seconds. The deadline on the problem was 290 minutes. The scheduler's answer required 254. A complete listing of the program run on this problem is given in Appendix A. The solution it found is presented in Table 6.3.

Location		Speed	Task	Time		
Origin	Destination			Earliest	Latest	Safe
Home	Laundry	30	Drop off laundry	61	61	69
Laundry	Garage	30	Drop off car	66	66	74
Garage	Grocery	5	Get food	98	108	106*
Grocery	Laundry	3	Transfer laundry	123	138	131*
Laundry	Garage	3	Pick up car	187	202	208
Garage	Laundry	30	Pick up laundry	192	207	213
Laundry	Post Office	30	Mail letters	198	216	234
Post Office	Home	30	Go home	254	272	290

*Parts of the plan that could take long enough to violate a deadline.

Table 6.3: Solution to Errand Running Problem Involving Vehicles

6.3.2 Designing Multi-Agent Plans in the Baseball Domain

It was stated early on in this dissertation that plan design was one area in planning research that could benefit from a task scheduler. To test this hypothesis out, experiments were performed with the task scheduler in designing defensive baseball plays for various situations. The design of the double-play is the subject of this section.

In the baseball domain the planner acts in the role of player coordinator or manager. The task the system is given is to design a double-play for a particular hitting situation. For example, the task scheduler is asked to direct the defensive team's movements so that if a ball is bunted up the third-base line, and there is a man on first, a double-play will result. The decisions the scheduler has to make are:

- Which player should field the ball?
- Which out should be tried for first (the batter or the runner)?
- Which player should cover second?
- Which player should cover first?

The problem statement for other hitting situations should be identical except for the hit specification.

The scheduler is given four task specifications from which to create the plan:

1. Field the ball
2. Get the force at first
3. Get the force at second
4. Get the tag at second

The third and fourth tasks both refer to the same out, that of the baserunner. The two specifications are necessary because the out is gotten in different ways depending on whether the batter is gotten out before or after the runner. In plan design, all of the applicable specifications from the task network are given to the scheduler. Part of the design problem is in the choosing of the precise specification to use for a particular task.

Requirements and deadlines are given for each of the specifications. To field the ball a defensive player must achieve the same location as the ball, To get the force at first base a player with the ball must be at first base in under six seconds. The evaporation conditions for the plans are also included:

- If the force at first is scheduled and the force at second has not yet been scheduled, then the force at second evaporates (i.e., has status EVAPORATED).
- If the force at second is scheduled then the tag at second evaporates.

The SOs included in the task specifications for the outs specify only where the ball must be. The SO for the field the ball task specifies only that a player must be wherever the ball is; that location is not specified. The SO given to start out the problem specifies the players and their initial positions, and the initial position of the ball on the ground. Resource functions for the players give delays necessary to move a specific player to a particular position. Delay functions are also given for throwing the ball from one player to another.

The scheduler relies on a simple heuristic for choosing which player should do what. Choose the player that can be in the correct position the soonest. If the ball is hit near the first-baseman, then the plan would call for him to retrieve it. Later in the play, when the plan called for the force at first, the scheduler would select some player to cover first. The player picked would be the first-baseman if he had not been drawn away from the base by fielding the ball. If he had, the plan would call for the pitcher to move to first. The first-baseman would not have to be drawn off far for this to happen; just far enough so that

if the pitcher started moving towards first at the same time the first-baseman was running after the ball, the pitcher would reach first before the first-baseman could return there. The bookkeeping necessary for these calculations was carried out by the time-stamps in the SOs (see section 3.2.1).

The only real ordering decision the scheduler has to make in this problem is whether to get the out at second before or after the out at first. The deadline on the tag at second is shorter than the deadline on the force at second, which in turn is shorter than that of the force at first. Because of the different deadlines, and because the deadline for the tag at second is not distinctly visible to the scheduler until the force at first is scheduled,[2] the force at second is usually performed first. On the occasions where the ball is placed very near first, so that force there is scheduled to come before the one at second, it is possible for the scheduler to make a mistake — try for the tag at second, miss the deadline and have to backup and do the force at second followed by the force at first. However, the scheduling heuristics quickly converge on the proper values to avoid the backup situation. All subsequent scheduling involved the minimum search of three nodes in the search tree.

6.4 Task Selection for the Forbin Planner

The Forbin planner [Firby 85], [Miller 85a] is designed to explore the effects of various forms of temporal reasoning on the planning process. Forbin is a hierarchical planner which consults the task scheduler and a temporal data-base, or *time-map*, when doing plan choice. In Forbin, having many alternative plans for a single task is commonplace; plan choice becomes more important in this system then is common in most other planners.

If there are many possible plan choices, then some mechanism must be used to distinguish which is the best choice. In Forbin, this choice is made by the task scheduler with the help of the temporal data base. First the ***Time-Map Manager***, which controls the temporal data-base, is queried with each of the possible plan choices for the task currently being expanded. For each choice, the TMM returns all of the temporal intervals in which it can find no conflict for executing that plan. Associated with each interval and plan, the TMM attaches a list of all of the constraints that executing that plan, at that time, would add to the state of the world. If there is no time at which a particular plan choice could be successfully executed, the TMM returns no intervals.

[2]The deadline on the tag at second task played a role in the calculation of the force at first's *task-weight* (see section 4.7.3)

The TMM makes all of its predictions based on information formed from the partial order of the ***plan prefix*** (i.e., the plan as it has so far been constructed). Therefore, if the TMM finds a reason for believing that a task would not execute successfully, it is almost certainly correct. However, a recommendation by the TMM to execute a specific plan during a specific temporal interval does not necessarily guarantee success. The TMM is not capable of detecting adverse plan interactions that are dependent on the exact order in which the plan steps are executed. To make the final plan choice, and to guarantee that a choice with a high probability of success does exist, the plans, intervals, and their constraints are sent to the task scheduler.

The scheduler gets, from the time-map, a copy of the plan prefix. It then attempts to create a schedule of plan-steps using the prefix, and one of the plan choices for the task being expanded. Which plan choice is picked depends on which the scheduler says will produce the best schedule overall. Several or all of the choices may be examined before one is finally picked.

The schedule that is produced by the scheduler is consistent with all of the constraints that were already introduced by the plan as previously constructed. Additionally, all of the constraints added by the TMM for the plan and time choice picked by the scheduler are also taken into account. The plan used in the task scheduler's schedule becomes the plan for the task being expanded. It will become incorporated into the larger plan. The particular time-slot chosen, along with the rest of the time and ordering constraints added by the scheduler, will not be enforced, in keeping with the policy of least commitment.

Figure 6.5 shows all the modules in the Forbin system, and the communication paths between the modules. The TE, or ***Task Expander***, is the core of the hierarchical planner. The TE retrieves plan choices from the library and instantiates the chosen plan with the correct values from the planning environment. The ***Task Queue Manager***, TQM for short, keeps track of all of the tasks that need to be expanded. The TQM selects the next task to be expanded and sends full expanded tasks (primitive robot actions) to the robot for final execution.

While there are non-primitive tasks in the TQM, FORBIN:

1. Pops the first task in the TQM and passes it to the TE.

2. The TE finds all the plan descriptions in the plan library which match the task. For each plan it finds, it asks the TMM for a time or times when the plan could be used.

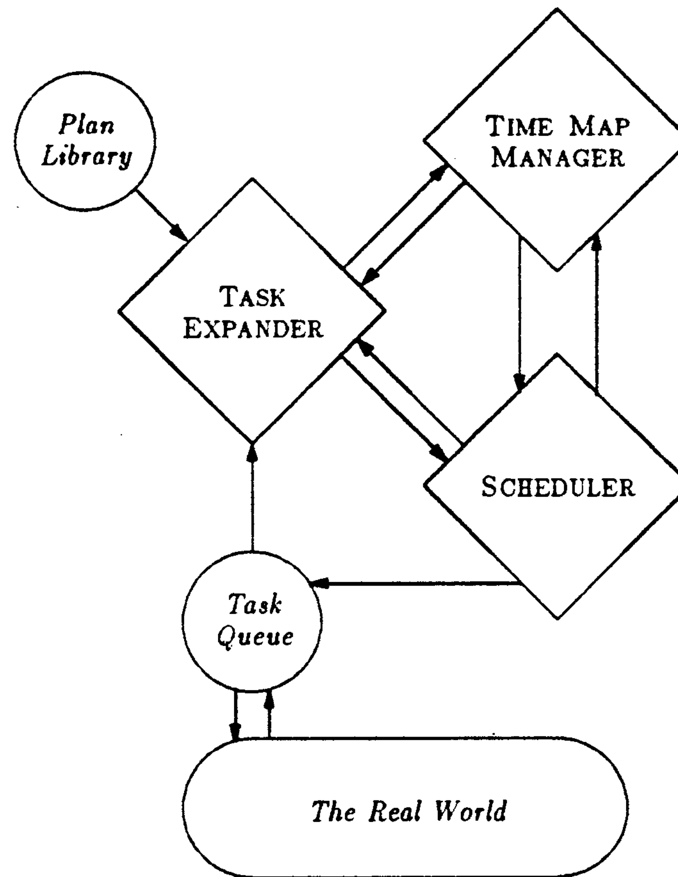


Figure 6.5: Flow of Control in the FORBIN System

3. The TMM derives all of the constraints necessary to make each plan suggested by the TE feasible.

4. The scheduler takes all the plan-constraint sets and finds the one that produces the best schedule when combined with the contents of the rest of the time map.

5. The TQM gets the schedule and the TE is passed the selected plan.

6. The TQM takes the schedule, extracts the new subtasks from it and adds them to its queues. The ordering in the schedule is used to help order the items in the queues.

7. The cycle then repeats until the TQM has no more unexpanded tasks.

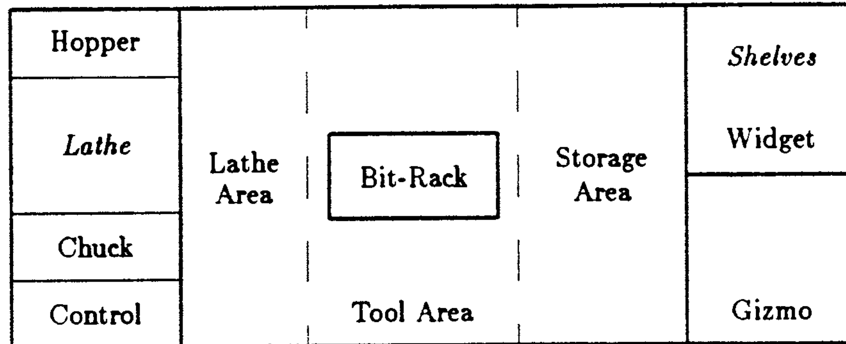


Figure 6.6: Layout of the Forbin Factory

Experiments have been performed, using the Forbin planner, in the domain of a semi-automated robot factory (see Figure 6.6). This factory has machines and tools that can run themselves, but they must be started, stopped, and have raw materials given to them by some outside agent. The agent, in this case, is a simple mobile robot equipped with two grippers. The robot roams around the factory setting up machines and providing them with the necessary raw materials. Its duties also include removing and storing the finished products. The travel times for the robot in the factory are given in Table 6.4. The Forbin planner is designed to control the actions of the mobile robot.

Tasks for the Forbin planner consist of manufacturing *widgets* and *gizmos*. To manufacture either of these products the following steps must be taken:

1. Make sure the correct bit is in the lathe.
2. Start the lathe and let it run until finished (11 units of time for a gizmo and 14 units for a widget).
3. Remove the finished item from the lathe hopper.
4. Place the item on the correct shelf.

There are three plans for making sure that the correct bit is in the lathe:

1. Do nothing (this plan assumes the correct bit is already there).
2. Put the correct bit into an empty lathe (this plan assumes the bit chuck is empty).
3. Take out the old bit and insert the correct bit (this plan assumes the wrong bit is in the lathe).

	Hopper	Chuck	Control	Bit-Rack	Widget	Gizmo
Hopper	0	2	3	3	5	6
Chuck	2	0	1	3	5	5
Control	3	1	0	3	6	5
Bit-Rack	3	3	3	0	3	3
Widget	5	5	6	3	0	2
Gizmo	6	5	5	3	2	0

Table 6.4: Travel Times in the Factory

In a production run of several widgets and gizmos it is quite possible that any of the three plans could be used — depending where in the run a particular widget or gizmo is going to be produced. For example, if ten widgets are going to be produced followed by ten gizmos, and then a new order comes in for a gizmo, any of the plans could apply for the production of the new gizmo. The new order could be tacked onto the end of the run thus requiring plan (1). If the new order is done before anything else then plan (2) could apply. If the gizmo is manufactured in the middle of the widget run then the bits would have to be swapped and plan (3) would be used. When to do the gizmo (and therefore, which plan to use) depends on the deadlines and requirements of all of the jobs; it is a decision made by the task scheduler. When given the tasks: make a gizmo and make a widget before time 35, Forbin comes up with the result shown in Table 6.5. With deadlines, travel time, and several task options, even such a simple task can result in an intricate plan.

Task	Place	Time		
		Travel	Task	Elapsed
(get (bit widget) hand1)	bit-rack	0	1	1
(put (bit widget) hand1)	lathe-chuck	3	1	5
(push (button widget) hand1)	lathe-control	1	1	7
(wait widget)	—	0	11	—
(get (bit gizmo) hand1)	bit-rack	3	1	11
(get (bit widget) hand2)	lathe-chuck	3	1	19[a]
(put (bit gizmo) hand1)	lathe-chuck	0	1	20
(push (button gizmo) hand1)	lathe-control	1	1	22
(wait gizmo)	—	0	14	—
(get widget hand1)	lathe-hopper	3	1	26
(put widget hand1)	(shelf widget)	5	1	32[b]
(get gizmo hand1)	lathe-hopper	5	1	38[c]
(put (bit widget) hand2)	bit-rack	3	1	42
(put gizmo hand1)	(shelf gizmo)	3	1	46[d]

[a] end (wait widget) at 18

[b] end (make widget)

[c] end (wait gizmo) at 36

[d] end (make gizmo)

Table 6.5: Forbin's Solution to the Example Problem

Chapter 7

Summary and Conclusions

The purpose of this research has been to create an automatic planning system for problem solving in domains where interactions between tasks may be a function of the order in which the tasks are executed — domains involving time and travel. Most robot domains contain interactions of this sort. Similar interactions also appear in domains where there are resource limitations and efficiency constraints. The most common form of this problem arises in situations where there are one or more types of robot travel time involved in preparing the robot for the execution of each task.

The solution that has been proposed in this work is to search through the space of totally-ordered plans, in order to detect all of the ordering-dependent interactions. Search through the plan space gives this system the predictive abilities of least-commitment planners. Additionally, because the plans are totally ordered, the system can know the exact state the world will be in when any particular plan step is executed. This allows the system to make plan choices based on more information than is available to least-commitment planners. The search allows the selection of an efficient ordering of task execution that contains a reasonable allocation of resources. This process is called ***task scheduling.***

7.1 Summary of Scheduling and Loop Scheduling Algorithms

The task scheduler produces its schedules by searching through the space of totally ordered plans. Because this search space grows exponentially with the number of tasks being scheduled, heuristics are used to help reduce the search space and direct the search. The task

scheduler uses special internal representations to capture information about task interdependencies and task state requirements. These representations allow a limited simulation of any legal schedule to be preformed. Since the number of legal schedules is potentially exponential on the number of tasks being scheduled, the scheduler performs a heuristic best-first search over the space of legal schedules. This search uses prerequisite ordering constraints, inter-task delays and deadlines, resource limitations and spatial restrictions, and absolute time windows to limit the search space.

Of the possibilities that remain, the system selects the tasks that have dependent tasks, approaching deadlines, minimal resource delays, or minimal differences between their transition delays and deadlines. These final tasks are then rated by the heuristics below:

- Estimates on the amount of time required to execute the completed schedule
- Necessity of scheduling a the task with regard to the relationship between its state-transition delays and its nearest deadline
- Traveling Salesman approximations on the best tour to minimize time required to perform state transitions (the RTT)
- Optimality of scheduling the task with regard to the relationship between its prerequisite delays and its state-transition delays
- Domain dependent heuristics set up for the particular problem being worked on (e.g., minimize run time of `lathe-33`)

As the search progresses it is possible that the scheduler will pursue a schedule that cannot meet all of the constraints of the problem. When this occurs, the class of constraint being violated is noted by the system. The overall scoring function for the system is altered so as to avoid missing this class of constraint in the future; the weights on each of the heuristics are altered to give more emphasis to those that detect the failed class of constraint. Through this dynamic scoring strategy, the heuristic rating function is able to conform to the problem being worked on by the task scheduler. The scores eventually decay back to their initial values so that a single portion of the problem cannot misdirect the scheduler for the remainder of its attempt at finding a solution.

The scheduler constructs a generator (called the SDA), which given an initial state of the world will generate all the legal next tasks that could be placed on the schedule. The scheduler takes the schedule extension which rates best and uses the generator to find the tasks that could follow it. If a schedule leads to a dead end (i.e., where there is no legal extension to the schedule), then the system goes back and picks the next highest

rating partial schedule, and starts extending it. The weights on the scoring heuristics are readjusted to take into account the type of dead end that was encountered.

When a scheduled task, or set of tasks, is to be repeated several times it is given a special CSO which keeps track of the number of times it has been placed onto the schedule. Each iteration of the repeated task is then placed onto the schedule in the normal manner. Occasionally, repeated tasks will cause a large section of the schedule itself to become repetitive. To aid in the efficiency of the scheduling process, the system periodically examines the schedule so far produced, and checks it for being in a repetitive state. If the schedule is found to be repetitive, and in a way that is likely to continue for a while, then that portion of the schedule is condensed into a repeat loop. The termination conditions for the schedule repeat loop depend on the tasks in the loop, and the other tasks that remain to be scheduled.

The schedule space that the system searches over is greatly reduced from that of all possible orderings. The rating heuristics allow the system to pursue only the most likely candidate schedules. A variety of tests show that the task scheduler is able to find a good schedule while searching a portion of the search tree equal to about $2.75N$. The amount of actual computation time required for a problem (for the current implementation) grows at slightly below $O(N^2)$.

7.2 Limitations and Possible Extensions to Task Scheduling

7.2.1 End-Coordinated Planning Problems

One type of timing situation that has not yet been discussed in this dissertation is the situation where two or more tasks are desired to finish simultaneously. For example, imagine a situation where a roast and rice are being cooked for dinner. This situation produces four tasks that can be given to the scheduler:

- `put-roast-in-oven`
- `take-roast-out`
- `start-rice-simmering`
- `turn-off-rice-heat`

One set of consistent constraints (in TSL) are presented in Figure 7.1. These constraints should force the scheduler to have the rice and the roast finish within a minute of one another.

```
(WITH-CONSTRAINTS
              ;;; cook roast at least 2hrs
  (MAY-START take-roast-out 120 put-roast-in-oven)
              ;;; cook rice at least 30min
  (MAY-START turn-off-rice-heat 30 start-rice-simmering)
              ;;; cook roast no more than 2hrs
  (DEADLINE  take-roast-out 120 put-roast-in-oven)
              ;;; cook rice no more than 30min
  (DEADLINE  turn-off-rice-heat 30 start-rice-simmering)
              ;;; finish the roast before the rice
  (PRECEDES take-roast-out turn-off-rice-heat)
              ;;; finish the rice within a minute after the roast
  (DEADLINE turn-off-rice-heat 1 take-roast-out))
```

Figure 7.1: Constraints for Coordinating Dinner

The constraints in Figure 7.1 are sufficient to ensure that any schedule the task scheduler comes up with will produce a well coordinated dinner. Unfortunately, the scheduler, as it is currently implemented, is unlikely to find any schedule that will satisfy all of the constraints; it therefore will not return any schedule at all. This failure of the task scheduler is due to its inability to know when it is okay to waste time. The scheduler will attempt to schedule both the `put-roast-in-oven` and `start-rice-simmering` tasks as early in the schedule as possible. If it starts the rice first, a deadline violation will occur on the `turn-off-rice-heat` task. This violation occurs because one of the task's prerequisites (`take-roast-out` cannot be scheduled within the thirty minutes allowed between the first and second rice tasks. The task scheduler will then backtrack and try an alternative ordering — putting the roast first.

Unfortunately, putting the roast into the oven before starting the rice, while the correct ordering of the tasks, is little more likely to result in a viable schedule. The task scheduler will attempt to schedule the rice as soon after the roast goes in as possible — leading to the same deadline failure as before. All of the alternatives available to the scheduler run into similar problems. All of these problems are the result of the scheduler not knowing how to kill time.

There are at least two ways the task scheduler could be extended in order to be able to handle this kind of planning problem. The first solution is to spot potentially troublesome

situations, and perform a **backwards** search from the trouble spot in order to set up initial time constraints that will result in a viable solution. Troublesome situations of this sort may be spotted by noticing cases where one single task has constraints stemming from two other independent tasks. In the example above the `turn-off-rice-heat` task is constrained by both the `take-roast-out` and `start-rice-simmering` tasks. Those two tasks are independent because neither one has any explicit constraints on the other. Using the basic scheduling algorithm over a reversed SDA[1] produces a backwards schedule that can be used to derive the additional temporal constraints on the tasks to keep them properly coordinated.

A cleaner way to get the task scheduler to solve these problems is to run the partial order of tasks through a time-map system such as that described in [Dean 86]. The time-map makes explicit all of the constraints that are implicit in a set of temporal constraints. The time-map would be able to add the constraint

```
(MAY-START start-rice-simmering 90 put-roast-in-oven)
```

to the constraint set given in Figure 7.1. With the additional constraint the task scheduler would have no problem coming up with the correct solution. Because both the time-map manager and the task scheduler are used in the Forbin planner, it is able to handle end-coordinated planning problems such as the cooking example.

7.2.2 Scheduling Resource-Intensive Tasks

The task scheduler described in this work has been designed to find a legal ordering that minimizes time used for state transitions as well as the overall time required by the plan. Resources that do not figure in the state-transition delay can be represented by this system, and the resulting ordering found by the task scheduler will include proper allocation of the additional resources. However, this system is not designed for, and will not do a good job at, minimizing (or maximizing) the amount of a specific resource expended in a plan for achieving a task.

The type of search being used by the task scheduler has only one dimension on which to back up: the ordering of tasks. The space of resource usage for a particular task can be

[1] If the tasks with constraint-free **rows** are made into ready tasks, rather than the tasks with empty **columns**, then the SDA will present tasks to the scheduler in the **reverse** order that they would be presented in the normal scheduling algorithm.

explored when the task is first being considered for scheduling, but no further search can be done in that area once the task had been put onto the schedule prefix. Even if there are several different state transitions that could be made involving the same ordering, only one state transition can be chosen, and that decision is permanent as long as the prefix exists. The task scheduler relies on the state objects to do its resource reasoning; the state objects must make the correct decision first time, every time. In most domains this is easy to do because resources are seldom very tightly constrained. When resources are tightly constrained there is usually one obviously correct thing to do, and that is what the state objects will usually do. If the scheduler selects a correct ordering with an incompatible resource schedule it will never be able to correct its mistake. That dimension of the search is confined to the domain-dependent rating functions and the state-transition routines.

The resource-dominated tasks that the scheduler may fail on all share certain similarities. In section 1.2.2 a planning problem was discussed where there were tasks that consumed money (e.g., errands such as shopping) and other tasks that produced money (e.g., withdrawing cash from the bank). If the costs and withdrawals were specified for each task then the scheduler could find a viable schedule even if there was just enough money to go around. However, if the amounts were not specified (e.g., the bank task just said to withdraw some amount of money, the amount (within limits) being left up to the scheduler) then it would be quite possible for the system not to be able to produce a schedule where all of the monetary constraints were met. Scheduling an investment portfolio (even if the properties of the stocks were well known) would be even more difficult.

It would be possible to make a more thorough analysis of the resources needed by all the tasks being scheduled. Such analyses are the subject of much current OR research. [Bell 83] is a good example of how complex the analysis must be even for one single very specific problem. Methods of doing resource flow analysis as a part of a general problem solving system, and integrating those analyses with the problem solving and scheduling algorithms, are subjects for future research.

7.2.3 Using the Task Scheduler for Planning

The task scheduler has been used in two planning systems: Bumpers and Forbin. The Bumpers system was built around the task scheduler and makes the best use of it. The mobile-robot domain in which Bumpers plans demands detailed plans that are fully computed. Many repetitions of the feedback loops designed by the task scheduler can successfuly be used by a robot during the execution of a task. In Bumpers, it is as expensive to plan

out one iteration of the loop as it is a thousand. Therefore, since at least one iteration will always be executed, Bumpers should, and does, plan out the steps for the robot to execute in tremendous detail.

It is not necessarily true that the same level of detailed planning should be performed in all domains. The Forbin domain (the semi-automated factory) is probably one domain where extensive planning is not only unnecessary, but potentially harmful.

The plans produced by the Forbin system made almost total use of the robot's time, thereby accomplishing the tasks in as little overall time as possible. Unfortunately, because of the efficient use of the robot's time, the plans produced by Forbin are very inflexible. If a task is added to the queue once planning has begun, or if a mechanical failure causes an unpredictable delay, then large parts of the plan will need to be reworked. Replanning is further complicated by the way the scheduler integrates the subplans into the overall plan. The solution shown in Table 6.5 is a good example of this; there is no good dividing line between where the widget-making task ends and the gizmo-making task begins. A large portion of the schedule is involved with the completion of both tasks.

It would be better if the Forbin system left the plan in various stages of detail.[2] Tasks that needed to be done soon, in order to meet deadlines, should be planned in extensive detail; those tasks with no immediate deadlines could be left less expanded. Unfortunately there is no obvious way to decide between the tasks needing immediate attention and those that do not. In the Forbin factory it is possible that an order will come in with a deadline that is later than the deadlines for all other orders. Should that order be placed into the back of the queue and not be examined by the task scheduler until other orders have been expanded, scheduled, and processed? Maybe, but not necessarily. If that order is for the making of a widget, and there is an order with a near deadline for making widgets, and all of the other orders are for gizmos, then it is possible that if the two widget orders are not combined a deadline violation might result. This violation would occur from the additional time needed to switch the set-up on the lathe from widget to gizmo, and then back again.

Passing all of the tasks to the scheduler, but just not expanding some until their execution times draw near, is not sufficient. In the problem above, the scheduler will not realize that the widget tasks must be combined until the widget task has been sufficiently expanded so as to expose the fact that the widget set-up on the lathe is required for part of the task. On the other hand, the widget-making task may not be properly expanded

[2]The possible improvements to the Forbin system discussed in this section originated in discussions with Jim Firby

unless the scheduler is called at each expansion for each task — a strategy that is known to produce efficient, but overly intricate solutions.

An alternative strategy, that is just in its exploratory stages, is to set up partitions between sets of tasks, and maintain the partitions (performing scheduling and task expansions just within that set) unless a deadline violation arises. If such a violation does arise then the sets of tasks that adjoin the one with the conflict can be collapsed together, and optimized by the scheduler. The initial sets of tasks, and their order, can be assigned by a task scheduling over the top-level set of tasks.

Partitioning the tasks into separate *packets* allows the scheduler to take radically different perspectives in scheduling each packet. The adaptive scoring used by the scheduler has a certain lag time. For example, once the scoring has edged over to giving high priority to reducing state-transition delays, several tasks will be scheduled before the scoring could become deadline-oriented — even if a deadline-oriented scoring would produce the most efficient schedules. If the deadline-oriented scoring is the only way the scheduler could find the correct answer then the system will either do considerable backtracking or, if the backup stacks run out of room, find no answer at all. Packets could greatly ease this situation. At the start of each packet the task scheduler would be starting out with no scoring bias, and would be able to shift to the appropriate scoring mode very quickly. Thus if one packet had several independent tasks all with the same deadline, then the scheduler could move its scoring bias towards reducing state-transition delays without fear that the bias would later have to be radically switched.

This strategy should yield plans divisible into packets where each packet is highly optimized. Corrections to one packet can then be made without adversely affecting all of the other tasks achieved by the plan. This strategy should also be more amenable to last-minute additions and deletions of tasks from the queue of things to do.

7.2.4 Why the Scheduler Chooses the Schedule it Does

The task scheduler, as currently implemented, annotates the schedule with the reasons why each task made it through the ready-task thinning heuristics (see section 4.2.2). These annotations give some insight into why one task was picked over another at a particular point in the schedule. The annotations are inadequate for explaining the larger view of the schedule — why this schedule looked better than radically different alternatives. Such information would be useful for redesigning the tasks to be more efficient. For example, in

the Forbin-factory domain it could be beneficial to know why one schedule won out over the alternatives. Such information might be sufficient to help reorganize the arrangement of machines to create a more efficient factory.

Of more direct use to planning would be a method where by the task scheduler gave insight to the overall significance of its choices. For example, does the schedule have the widget production come before that of the gizmo because:

- The widget coming first is necessary for creating a feasible schedule.
- The widget coming first produces a much more efficient schedule than if its production went elsewhere.
- The widget came first in the schedule because it had to go somewhere, and this was the first place that was tried.

The answer may not always be so cut and dried, but some indication would be useful in deciding which ordering decisions should be enforced (i.e., put directly into the time-map) and which might be able to be relaxed. In domains where the scheduler is dealing with events over large expanses of time, this information could play an additional role. If the scheduler is given the problem of planning out the next year of my life then there are several tasks which must be handled immediately (e.g., getting the dissertation to the registrar, eating lunch) while others will happen in the far future (e.g., planting a garden, going on a summer vacation). In fact the distant tasks are not sufficiently expanded, and the future is too uncertain, for any detailed scheduling on them to be performed. By noticing why the scheduler is incorporating a task into the plan-schedule the scheduler can decide not to decide where to schedule that task.

All the necessary information is available to the system when it creates a schedule. For some of it an analysis of the trends of the schedule search might be necessary. Exactly what information is needed, and how it should organized and returned form the scheduler, are worthy projects for future research.

A task scheduler allows a deeper reasoning about a planning problem then has been possible with previous automatic-planning systems. With a task scheduler, problems involving time, multiple agents, multiple plans for the same task, loops, and continuous resources, all become solvable. The experiments in this dissertation have shown that a planning module with these capabilities can be created that will efficiently search the space of totally-ordered plans, and find a viable schedule.

References

[Allen 81] Allen, James, *Maintaining knowledge about temporal intervals,* Technical Report TR86, U. of Rochester Department of Computer Science, 1981.

[Allen 83] Allen, James, *Maintaining knowledge about temporal intervals,* Comm. ACM, 26/11 (1983), pp. 832–843.

[Bell 83] Bell, W.J., Dalberto, L.M., Fisher, M.L., Greenfield, A.J., Jaikumar, R., Kedia, P., Mack, R.G., Prutzman, P.J., *Improving the Distribution of Industrial Gases with an On-line Computerized Routing and Scheduling Optimizer,* Interfaces, 13 December (1983), pp. 4–23.

[Brooks 82a] Brooks, R.A., *Symbolic Error Analysis and Robot Planning,* International Journal of Robotics Research, 1/4 (1982), pp. 29–68.

[Brooks 82b] Brooks, Rodney, A., Solving the Find-Path Problem by Good Representation of Freespace, *Proceedings of the National Conference on Artificial Intelligence,* AAAI, Pittsburgh, Pennsylvania, August 1982, pp. 381–386.

[Cheeseman 84] Cheeseman, P., A Representation of Time for Automatic Planning in Complex Domains, *Proceedings of the 1984 International Computers in Engineering Conference,* ASME, LasVegas, Nevada, August 1984, pp. 352.

[Coles 75] L.S.Coles, A.M.Robb, P.L.Sinclair, M.H.Smith, & R.R.Sobek, Decision Analysis for an Experimental Robot with Unreliable Sensors, *Proceedings of the Fourth International Joint Conference on Artificial Intelligence,* IJCAI, August 1975, pp. 749–754.

[Davis 84] Davis, E., *Representing and Acquiring Geographic Knowledge,* Technical Report 292, Yale University Computer Science Department, 1984.

[Dean 83] Dean, Thomas, *Time Map Maintenance,* Technical Report 289, Yale University Computer Science Department, 1983.

[Dean 85] Dean, T., Miller, D., Firby, R.J., *Hierarchical Planning: What's It Good For?,* 1985.

[Dean 86] Dean, T., *Temporal Imagery: An Approach to Reasoning about Time for Planning and Problem Solving,* 1986. Forthcoming PhD thesis.

[Drumheller 84] Drumheller, M., *Robot Localization Using Range Data,* Technical Report, Dept of Mechanical Engineering, MIT, 1984. S.B. Thesis.

[Drummond 85] Drummond, Mark, Refining and Extending the Procedural Net, *Proceedings of the Ninth International Joint Conference on Artificial Intelligence,* IJCAI, AAAI, Los Angeles, CA, August 1985.

[Ellis 85] Ellis, J.R., *Bulldog: A Compiler for VLIW Architectures*, Technical Report 364, Yale University Computer Science Department, February 1985.

[Ernst 69] Ernst, George W. and Newell, Allen, *GPS: A Case Study in Generality and Problem Solving*, Academic Press, 1969.

[Feldman 75] Feldman, Jerome A. & Sproull, Robert F., *Decision Theory and Artificial Intelligence II: The Hungry Monkey*, Technical Report, University of Rochester Department of Computer Science, 1975.

[Fikes 71] Fikes, Richard and Nilsson, Nils J., *STRIPS: A new approach to the application of theorem proving to problem solving*, Artificial Intelligence, 2 (1971), pp. 189–208.

[Finkel 74] Finkel, R., Taylor, R., Bolles, R., Paul, R., Feldman, J., *AL, A Programming System for Automation*, Memo AIM-177, Stanford University Artificial Intelligence Laboratory, 1974.

[Firby 85] Firby, R.J., Dean, T., Miller, D., Efficient Robot Planning with Deadlines and Travel Time, *Proc. of the 6th Int. Symp. on Robotics and Automation*, IASTED, Santa Barbara, CA, May 1985, pp. 97–101.

[Fisher 76] Fisher, M.L., Jaikumar, R., *A Generalized Assignment Heuristics for Vehicle Routing*, Research Report 79-10-04, Dept. of Decision Sciences, The Wharton School, Univ. of Pennsylvania, August 1976.

[Fisher 81] Fisher,J.A., *Trace Scheduling: A Technique for Global Microcode Compaction*, IEEE Transactions on Computers, C-30/7 July (1981), pp. 478–490.

[Fox 83] Fox, Mark S., *Constraint-Directed Search: A Case Study of Job-Shop Scheduling*, Technical Report CMU-RI-TR-83-22, CMU Robotics Institute, December 1983.

[Fox 85] Fox, B.R., Kempf, K.G., Opportunistic Scheduling for Robotic Assembly, *Proceedings of the 1985 International Conference on Robotics and Automation*, IEEE, St Louis, Mo., March 1985, pp. 880–886.

[Gevarter 85] Gevarter, W.B., Nachysheim, P.R., Stutz, J.C., and Banda, C.P., An Expert System for the Planning and Scheduling of Astronomical Observations, *Proceedings of the 1985 International Conference on Robotics and Automation*, IEEE, St Louis, Mo., March 1985, pp. 221–226.

[Gini 83] Gini, M. and Gini, G., Twoards Automatic Error Recovery in Robot Programs, *Proceedings of the Eighth International Joint Conference on Artificial Intelligence*, IJCAI, Karlsruhe, West Germany, August 1983, pp. 821–823.

[Goldberg 83] Goldberg, A., Robson, D., *Smalltalk-80: the Language and Its Implementation*, Addison-Wesley, 1983.

[Graham 77] Graham, R.L., Lawler, E.L., Lenstra, J.K., Kan, A.H.G, Optimization and Approximation in Deterministic Sequencing and Scheduling: a Survey, *Proc. of Discrete Optimization 1977,* Vancouver, B.C., Canada, August 1977.

[Hayes 84] Hayes, Patrick, Liquids, Hobbs, J. ed., *Formal Theories of the Commonsense World,* Lawrence Erlbaum Associates, 1984.

[Hendrix 73] Hendrix, Gary G., *Modeling Simultaneous Actions and Continuous Processes,* Artificial Intelligence, 4 (1973), pp. 145–180.

[Jacobs 73] Jacobs, W., & Kiefer, M., Robot Decisions Based on Maximizing Utility, *Proceedings of the Third International Joint Conference on Artificial Intelligence,* IJCAI, Stanford, Ca, U.S.A, August 1973, pp. 402–411.

[Kafura 77] Kafura, D. G., Shen, V. Y, *Task Scheduling on a System with Independent Memories,* SIAM J. Comput., 6/1 (1977), pp. 167–187.

[Kirkpatrick 79] Kirkpatrick, D., Efficient Computation of Continuous Skeletons, *Proceedings of the 20th Symposium on Foundations of Computer Science,* 1979, pp. 18–27.

[Lenstra 77] Lenstra, J.K., Kan, A.H.G., Brucker, P., *Complexity of Machine Scheduling Problems,* Annals of Discrete Mathematics, 1 (1977), pp. 343–362.

[Lowerre 76] Lowerre, B., *The HARPY Speech Recognition System,* Ph.D. Thesis, Carnegie-Mellon University, 1976.

[McCalla 82] McCalla, G.I., Reid, L., and Schneider, P.F., *Plan Creation, Plan Execution and Knowledge Acquisition in a Dynamic Microworld,* International Journal of Man-Machine Studies, 16 (1982), pp. 89–112.

[McCarthy 58] McCarthy, John, Programs with common sense, *Proceedings of the Symposium on the Mechanization of Thought Processes,* National Physiology Laboratory, 1958.

[McCarthy 69] McCarthy, John and Hayes, Patrick J., Some Philosophical Problems from the Standpoint of Artificial Intelligence, Meltzer, B. and Michie, D. eds., *Machine Intelligence 4,* Edinburgh University Press, 1969.

[McDermott 82] McDermott, Drew V., *A temporal logic for reasoning about processes and plans,* Cognitive Science, 6 (1982), pp. 101–155.

[McDermott 83] McDermott, Drew, *The NISP Manual,* Technical Report 274, Yale University Computer Science Department, June 1983.

[McDermott 84] McDermott, Drew V. and Davis, Ernest, *Planning routes through uncertain territory,* Artificial Intelligence, 22 (1984), pp. 107–156.

[Miller 83] Miller, David, *Scheduling Heuristics for Problem Solvers,* Technical Report 264, Yale University Department of Computer Science, 1983.

[Miller 84] Miller, David, Two Dimensional Mobile Robot Positioning, *Proceedings of the Ninth William T. Pecora Memorial Remote Sensing Symposium,* IEEE, USGS, NASA, ASP, Sioux Falls, South Dakota, October 1984, pp. 362–369.

[Miller 85a] Miller, D., Firby, R.J., Dean, T., Deadlines, Travel Time, and Robot Problem Solving, *Proceedings of the Ninth International Joint Conference on Artificial Intelligence,* IJCAI, AAAI, Los Angeles, CA, August 1985, pp. 1052–1054.

[Miller 85b] Miller, David, A Spatial Representation System for Mobile Robots, *Proceedings of the 1985 International Conference on Robotics and Automation,* IEEE, St Louis, Mo., March 1985, pp. 122–127.

[Munson 70] Munson, John H., *A Cost-Effectiveness Basis for Robot Problem Solving and Execution,* Technical Note 29, Stanford Research Institute, 1970.

[Munson 71] Munson, John H., Robot Planning, Execution, and Monitoring in an Uncertain World, *Proceedings of the Second International Joint Conference on Artificial Intelligence,* IJCAI, Somewhere over the Horizon, August 1971, pp. 338–349.

[O'Dunlaing 83] O'Dunlaing, C., Sharir, M., Yap, C., Retraction: A New Approach to Motion Planning, *Proceedings of the Fifteenth Annual ACM Symposium on Theory of Computing,* ACM, Boston, Massachusetts, April 1983, pp. 207–220.

[O'Rourke 84] O'Rourke, Joseph, Convex Hulls, Voronoi Diagrams, and Terrain Navigation, *Proceedings of the Ninth William T. Pecora Memorial Remote Sensing Symposium,* IEEE, USGS, NASA, ASP, Sioux Falls, South Dakota, October 1984, pp. 358–361.

[Ramamritham 84] Ramamritham, K., Stankovic, J.A., Dynamic Task Scheduling in Distributed Hard Real-Time Systems, *Proc. of 4th International Conference on Distributed Computer Systems,* May 1984.

[Roach 85] Roach, J., Boaz, M., Coordinating the Motions of Robot Arms in a Common Workspace, *Proceedings of the 1985 International Conference on Robotics and Automation,* IEEE, St Louis, Mo., March 1985, pp. 494–499.

[Rowat 79] Rowat, P.F., *Representing Spatial Experience and Solving Spatial Problems in a Simulated Robot Environment,* Technical Report, University Of British Columbia, 1979.

[Sacerdoti 74] Sacerdoti, Earl D., *Planning in a Heirarchy of Abstraction Spaces,* Artificial Intelligence, 5 (1974), pp. 115–135.

[Sacerdoti 77] Sacerdoti, Earl, *A Structure for Plans and Behavior,* American Elsevier Publishing Company, Inc., 1977.

[Shamos 75] Shamos, M., Hoey, D., Closest Point Problems, *Proceedings of the 16th Symposium on Foundations of Computer Science,* 1975, pp. 151–162.

[Smith 83] Smith, S.F., *Exploiting Temporal Knowledge to Organize Constraints,* Technical Report CMU-RI-TR-83-12, CMU Robotics Institute, 1983.

[Smith 85] Smith, S.F., Ow, P.S., The Use of Multiple Problem Decompositions in Time Constrained Planning Tasks, *Proceedings of the Ninth International Joint Conference on Artificial Intelligence,* IJCAI, AAAI, Los Angeles, CA, August 1985.

[Sussman 75] Sussman, Gerald J., *A Computer Model of Skill Acquisition,* American Elsevier Publishing Company, Inc., 1975.

[Tate 75] Tate, Austin, *Using Goal Structure to Direct Search in a Problem Solver,* Ph.D. Thesis, University of Edinburgh, 1975.

[Tate 76] Tate, Austin, *Project Planning Using a Hierarchic Non-Linear Planner,* Research Report 25, Dept. of Artificial Intelligence, University of Edinburgh, 1976.

[Tate 77] Tate, Austin, Generating Project Networks, *Proc. of the 5th Int. Joint Conf. on Artificial Intelligence,* IJCAI, Cambridge, Ma, U.S.A, August 1977, pp. 888–893.

[Taylor 76] Taylor, Russell H., *A Synthesis of Manipulator Control Programs from Task-Level Specifications,* Ph.D. Thesis, Stanford University, July 1976.

[Vere 83a] Vere, Steven A., *Planning in Time: Windows and Durations for Activites and Goals,* IEEE Transactions on Pattern Analysis and Machine Intelligence, PAMI-5/3 May (1983), pp. 246–267.

[Vere 83b] Vere, Steven, *Planning in Time: Windows and Durations for Activities and Goals,* IEEE Trans. on Pattern Analysis and Machine Intelligence, PAMI-5/3 (1983), pp. 246–267.

[Vere 84] Vere, Steven, *Temporal Scope of Assertions and Window Cutoff,* 1984. JPL, AI Research Group Memo.

[Ward 82] Ward, B., McCalla, G., Error Detection and Recovery in a Dynamic Planning Environment, *Proceedings of the National Conference on Artificial Intelligence,* AAAI, Pittsburgh, Pennsylvania, August 1982, pp. 172–175.

Appendix A

Example Task-Scheduling Run

This appendix contains an annotated program run of the task scheduler on the errand running problem described in section 6.3.1. Presented below are the actual scheduler definitions of the tasks. For not particularly interesting historical reasons, an internal task description is called a *worm*. A worm consists of:

1. A task name
2. A domain specific utilitiy function (none are provided in this example)
3. An execution window
4. An interval describing how long the task will take to execute
5. A state object
6. A current status object
7. A list of prerequisite objects.

```
(:= home1
    (make WORM 'go-home ()
              (pre:make-standard-window 0 290 'go-home)
              (make T-INTERVAL 0 0)
              (so:places 'home () ())
              (stat:make-standard-status 'go-home)
              (list (pre:make-standard-order-ptask 'pick-up-car)
                    (pre:make-standard-order-ptask 'pick-up-laundry)
                    (pre:make-standard-order-ptask 'mail-letters)
                    (pre:make-standard-order-ptask 'food-shopping))))

(:= car1
    (make WORM 'drop-off-car ()
```

```
                (pre:make-standard-window 0 3000 'drop-off-car)
                (make T-INTERVAL 1 1)
                (so:places 'garage () '(car))
                (stat:make-standard-status 'drop-off-car)
                ()))

(:= car2
    (make WORM 'pick-up-car ()
                (pre:make-standard-window 0 3000 'pick-up-car)
                (make T-INTERVAL 1 1)
                (so:places 'garage '(car) ())
                (stat:make-standard-status 'drop-off-car)
                (list (pre:make-bounded-ptask 'drop-off-car 'pick-up-car 120 150))))

(:= laundry1
    (make WORM 'drop-off-laundry ()
                (pre:make-standard-window 0 3000 'drop-off-laundry)
                (make T-INTERVAL 1 1)
                (so:places 'laundr-o-mat () '(laundry))
                (stat:make-standard-status 'drop-off-laundry)
                ()))

(:= laundry2
    (make WORM 'transfer-laundry ()
                (pre:make-standard-window 0 3000 'transfer-laundry)
                (make T-INTERVAL 5 10)
                (so:places 'laundr-o-mat () ())
                (stat:make-standard-status 'transfer-laundry)
                (list (pre:make-bounded-ptask
                         'drop-off-laundry 'transfer-laundry 40 70))))

(:= laundry3
    (make WORM 'pick-up-laundry ()
                (pre:make-standard-window 0 3000 'pick-up-laundry)
                (make T-INTERVAL 1 1)
                (so:places 'laundr-o-mat '(laundry) ())
                (stat:make-standard-status 'pick-up-laundry)
                (list (pre:make-bounded-ptask
                         'transfer-laundry 'pick-up-laundry 60 90))))
(:= food1
    (make WORM 'food-shopping ()
                (pre:make-standard-window 0 3000 'food-shopping)
                (make T-INTERVAL 20 30)
                (so:places 'grocery '(food) ())
                (stat:make-standard-status 'food-shopping)
                ()))
(:= po1
    (make WORM 'mail-letters ()
                (pre:make-standard-window 0 3000 'mail-letters)
                (make T-INTERVAL 2 5)
                (so:places 'po () ())
                (stat:make-standard-status 'mail-letters)
                ()))

(proc VOID vehicle-test ()
   (schd:fsearch (list car1 car2 laundry1 laundry2 laundry3 food1 home1 po1)
                 (make T-INTERVAL 0 0)
```

```
                (so:places-init )))
```

What follows is some of the debugging output from the scheduler. The run is quite long; parts of extreme interest are highlighted by bars of stars (e.g., ****). Additional annotations appear in this type face; output from the system appears in this type face.

```
-> (:+ s2 (vehicle-test))
------------Weights------------  ;;; These are the initial scoring weights
The task weight factor is: 8
The approaching deadline factor is: 14
The RTT factor is: 11
The domain heuristics factor is: 13
The zero state delay factor is: 100
The waste time factor is: 8
----------End--Weights---------

Starting a SCHD:SEARCH at time 0 0 with tasks:
WORM: drop-off-car has the following properties:
     has an initial mandatory delay of 0
     and a deadline of 3000
     The execution time is in the range of 1 1
     The current status is: waiting
     -------------Start STATE OBJECT-------------
     The robot-loc is in state: garage
     The possessions is in state: nil
     The car-loc is in state: 'garage
     -------------End of STATE OBJECT-------------
WORM: pick-up-car has the following properties:
     It must be done at least
        120 after drop-off-car is finished
     has an initial mandatory delay of 0
     and a deadline of 3000
     The execution time is in the range of 1 1
     The current status is: waiting
     -------------Start STATE OBJECT-------------
     The robot-loc is in state: garage
     The possessions is in state: (car)
     The car-loc is in state: 'robot
     -------------End of STATE OBJECT-------------
WORM: drop-off-laundry has the following properties:
     has an initial mandatory delay of 0
     and a deadline of 3000
     The execution time is in the range of 1 1
     The current status is: waiting
     -------------Start STATE OBJECT-------------
     The robot-loc is in state: laundr-o-mat
     The possessions is in state: nil
     -------------End of STATE OBJECT-------------
WORM: transfer-laundry has the following properties:
     It must be done at least
        40 after drop-off-laundry is finished
     has an initial mandatory delay of 0
     and a deadline of 3000
     The execution time is in the range of 5 10
```

```
     The current status is: waiting
     -------------Start STATE OBJECT-------------
     The robot-loc is in state: laundr-o-mat
     The possessions is in state: nil
     -------------End of STATE OBJECT-------------
WORM: pick-up-laundry has the following properties:
     It must be done at least
        60 after transfer-laundry is finished
     has an initial mandatory delay of 0
     and a deadline of 3000
     The execution time is in the range of 1 1
     The current status is: waiting
     -------------Start STATE OBJECT-------------
     The robot-loc is in state: laundr-o-mat
     The possessions is in state: (laundry)
     -------------End of STATE OBJECT-------------
WORM: food-shopping has the following properties:
     has an initial mandatory delay of 0
     and a deadline of 3000
     The execution time is in the range of 20 30
     The current status is: waiting
     -------------Start STATE OBJECT-------------
     The robot-loc is in state: grocery
     The possessions is in state: (food)
     -------------End of STATE OBJECT-------------
WORM: go-home has the following properties:
     It must be done at least          ;;; a -1 indicates a PRECEDES constraint
        -1 after pick-up-car is finished
        -1 after pick-up-laundry is finished
        -1 after mail-letters is finished
        -1 after food-shopping is finished
     has an initial mandatory delay of 0
     and a deadline of 290
     The execution time is in the range of 0 0
     The current status is: waiting
     -------------Start STATE OBJECT-------------
     The robot-loc is in state: home
     The possessions is in state: nil
     -------------End of STATE OBJECT-------------
WORM: mail-letters has the following properties:
     has an initial mandatory delay of 0
     and a deadline of 3000
     The execution time is in the range of 2 5
     The current status is: waiting
     -------------Start STATE OBJECT-------------
     The robot-loc is in state: po
     The possessions is in state: nil
     -------------End of STATE OBJECT-------------
     -------------Start STATE OBJECT-------------
     The robot-loc is in state: home
     The car-loc is in state: robot
     The car-loc restriction:          ;;; this restriction keeps the car
        car-unavailable                ;;; from being used when it is in the shop
     is currently idle, but will be activated by the task: drop-off-car
     and terminated by the task: pick-up-car.

     The possessions is in state: (car laundry)
```

```
     The speed is in state: 30
     -------------End of STATE OBJECT-------------

The status of the Schedule Dependency Array is:
     task                          status          countdown
     drop-off-car                  schedulable     nil
     pick-up-car                   waiting         nil
     drop-off-laundry              schedulable     nil
     transfer-laundry              waiting         nil
     pick-up-laundry               waiting         nil
     food-shopping                 schedulable     nil
     go-home                       waiting         nil
     mail-letters                  schedulable     nil
End status SDA

*********************************************************************
```

Below are the ready-tasks — the tasks whose prerequisites have been met. Each task is followed by four numbers: The madatory delay left from the prerequisites, the state-transition delay, the time till the task's closest deadline, and the RTT over the ready-tasks. After the numbers is a list of the reasons why that task made it through the ready-task filters.

```
*********************************************************************

The tasks (drop-off-laundry 0 60 3000 100 (best-dd)
           drop-off-car 0 60 3000 100 (min-state-change)
           food-shopping 0 62 3000 120 (best-dd closest-dl)
           mail-letters 0 56 3000 98 (lowest-rtt min-state-change closest-dl))
     are ready to be scheduled.
```

I1I ;;; The iteration number

```
The status of the Schedule Dependency Array is:
     task                          status          countdown
     drop-off-car                  finished        t
     pick-up-car                   schedulable     nil
     drop-off-laundry              schedulable     nil
     transfer-laundry              waiting         nil
     pick-up-laundry               waiting         nil
     food-shopping                 schedulable     nil
     go-home                       waiting         nil
     mail-letters                  schedulable     nil
End status SDA
The tasks (drop-off-laundry 0 40 2939 93 nil
           food-shopping 0 20 2939 106 (min-state-change)
           mail-letters 0 40 2939 93 (best-dd closest-dl)
           pick-up-car 120 0 150 80 (lowest-rtt best-dd min-state-change closest-dl))
    are ready to be scheduled.

                    ---Creating New Environment---:
     The environment at time 61 61 has position data
     -------------Start STATE OBJECT-------------
     The speed is in state: 3
     The robot-loc is in state: garage
     The possessions is in state: (laundry)
     The car-loc is in state: 'garage
     The car-loc restriction:
```

```
        car-unavailable
    is active and will be terminated by the task: pick-up-car.

    -------------End of STATE OBJECT-------------
The status of the Schedule Dependency Array is:
    task                            status          countdown
    drop-off-car                    finished        t
    pick-up-car                     schedulable     nil
    drop-off-laundry                schedulable     nil
    transfer-laundry                waiting         nil
    pick-up-laundry                 waiting         nil
    food-shopping                   schedulable     nil
    go-home                         waiting         nil
    mail-letters                    schedulable     nil
End status SDA
    The schedule so far is:
        Move robot          (move robot from location home to garage at 30 mph
                                            (then drop off car))
        Reasons: (min-state-change)
      drop-off-car                      61 61  0

I2I
```

****

After scheduling the car to be dropped off first, the scheduler now examines a different branch in the tree: starting the laundry first. The scheduler periodically switches between branches when one looks more attractive then what it had been working on. This does not imply the old branch has failed, only that it is not as promising as it first seemed.

****

```
The status of the Schedule Dependency Array is:
    task                            status          countdown
    drop-off-car                    schedulable     nil
    pick-up-car                     waiting         nil
    drop-off-laundry                finished        t
    transfer-laundry                schedulable     nil
    pick-up-laundry                 waiting         nil
    food-shopping                   schedulable     nil
    go-home                         waiting         nil
    mail-letters                    schedulable     nil
End status SDA
The tasks (drop-off-car 0 4 2939 81 nil
           food-shopping 0 2 2939 100 (min-state-change)
           mail-letters 0 4 2939 81 (best-dd closest-dl)
           transfer-laundry 40 0 70 80 (lowest-rtt best-dd min-state-change closest-dl))
   are ready to be scheduled.

                ---Creating New Environment---:
    The environment at time 61 61 has position data
    -------------Start STATE OBJECT-------------
    The speed is in state: 30
    The car-loc is in state: robot
    The car-loc restriction:
        car-unavailable
    is active and will be terminated by the task: pick-up-car.
```

```
     The robot-loc is in state: laundr-o-mat
     The possessions is in state: (car)
     -------------End of STATE OBJECT-------------
The status of the Schedule Dependency Array is:
     task                          status        countdown
     drop-off-car                  schedulable   nil
     pick-up-car                   waiting       nil
     drop-off-laundry              finished      t
     transfer-laundry              schedulable   nil
     pick-up-laundry               waiting       nil
     food-shopping                 schedulable   nil
     go-home                       waiting       nil
     mail-letters                  schedulable   nil
End status SDA
     The schedule so far is:
          Move robot          (move robot from location home to laundr-o-mat at 30 mph
                                             (then drop off laundry))
          Reasons: (best-dd)
        drop-off-laundry              61 61  0

I3I

The status of the Schedule Dependency Array is:
     task                          status        countdown
     drop-off-car                  finished      t
     pick-up-car                   schedulable   nil
     drop-off-laundry              finished      t
     transfer-laundry              schedulable   nil
     pick-up-laundry               waiting       nil
     food-shopping                 schedulable   nil
     go-home                       waiting       nil
     mail-letters                  schedulable   nil
End status SDA
The tasks (mail-letters 0 24 2934 108 nil
           food-shopping 0 12 2934 104 (min-state-change)
           pick-up-car 120 0 150 80 (lowest-rtt best-dd min-state-change closest-dl)
           transfer-laundry 35 24 65 88 (best-dd closest-dl))
    are ready to be scheduled.
```

****

Sometimes a task will have delays that would cause it (if scheduled) to create a deadline violation for some other task. In this particular case there is a mandatory 120 minute delay on the pick-up-car task, yet only 65 minutes before a deadline on the transfer laundry task. There is no reason to even create a branch in the schedule tree for the pick-up-car task.

****

```
Making Branches
Critically removing pick-up-car
For task transfer-laundry
                      ---Creating New Environment---:
    The environment at time 66 66 has position data
    -------------Start STATE OBJECT-------------
    The speed is in state: 5
```

```
    The robot-loc is in state: garage
    The possessions is in state: nil
    The car-loc is in state: 'garage
    The car-loc restriction:
        car-unavailable
    is active and will be terminated by the task: pick-up-car.

    -------------End of STATE OBJECT-------------
The status of the Schedule Dependency Array is:
    task                          status          countdown
    drop-off-car                  finished        t
    pick-up-car                   schedulable     nil
    drop-off-laundry              finished        t
    transfer-laundry              schedulable     nil
    pick-up-laundry               waiting         nil
    food-shopping                 schedulable     nil
    go-home                       waiting         nil
    mail-letters                  schedulable     nil
End status SDA
    The schedule so far is:
        Move robot          (move robot from location home to laundr-o-mat at 30 mph
                                           (then drop off laundry))
        Reasons: (best-dd)
      drop-off-laundry                 61 61  0
        Move robot          (move robot from location laundr-o-mat to garage at 30 mph
                                           (then drop off car))
      drop-off-car                     66 66  0

I4I

The status of the Schedule Dependency Array is:
    task                          status          countdown
    drop-off-car                  schedulable     nil
    pick-up-car                   waiting         nil
    drop-off-laundry              finished        t
    transfer-laundry              schedulable     nil
    pick-up-laundry               waiting         nil
    food-shopping                 schedulable     nil
    go-home                       waiting         nil
    mail-letters                  finished        t
End status SDA
The tasks (drop-off-car 0 4 2933 62 (lowest-rtt min-state-change)
           food-shopping 0 4 2933 92 (best-dd closest-dl)
           transfer-laundry 34 4 64 62 (best-dd min-state-change closest-dl))
    are ready to be scheduled.

                    ---Creating New Environment---:
    The environment at time 67 70 has position data
    -------------Start STATE OBJECT-------------
    The car-loc is in state: robot
    The car-loc restriction:
        car-unavailable
    is active and will be terminated by the task: pick-up-car.

    The speed is in state: 30
    The robot-loc is in state: po
    The possessions is in state: (car)
```

```
     -------------End of STATE OBJECT-------------
The status of the Schedule Dependency Array is:
     task                          status        countdown
     drop-off-car                  schedulable   nil
     pick-up-car                   waiting       nil
     drop-off-laundry              finished      t
     transfer-laundry              schedulable   nil
     pick-up-laundry               waiting       nil
     food-shopping                 schedulable   nil
     go-home                       waiting       nil
     mail-letters                  finished      t
End status SDA
     The schedule so far is:
          Move robot          (move robot from location home to laundr-o-mat at 30 mph
                                   (then drop off laundry))
          Reasons: (best-dd)
        drop-off-laundry                  61 61  0
          Move robot          (move robot from location laundr-o-mat to po at 30 mph
                                   (the destination))
          Reasons: (best-dd closest-dl)
        mail-letters                      67 70  0

I5I

The status of the Schedule Dependency Array is:
     task                          status        countdown
     drop-off-car                  finished      t
     pick-up-car                   schedulable   nil
     drop-off-laundry              finished      t
     transfer-laundry              schedulable   nil
     pick-up-laundry               waiting       nil
     food-shopping                 schedulable   nil
     go-home                       waiting       nil
     mail-letters                  finished      t
End status SDA
The tasks (food-shopping 0 12 2928 96 (min-state-change)
           pick-up-car 120 0 150 60 (lowest-rtt best-dd min-state-change closest-dl)
           transfer-laundry 29 24 59 72 (best-dd closest-dl))
    are ready to be scheduled.

Making Branches
Critically removing pick-up-car
For task transfer-laundry
                        ---Creating New Environment---:
     The environment at time 72 75 has position data
     -------------Start STATE OBJECT-------------
     The speed is in state: 5
     The robot-loc is in state: garage
     The possessions is in state: nil
     The car-loc is in state: 'garage
     The car-loc restriction:
        car-unavailable
     is active and will be terminated by the task: pick-up-car.

     -------------End of STATE OBJECT-------------
The status of the Schedule Dependency Array is:
     task                          status        countdown
```

```
    drop-off-car                finished      t
    pick-up-car                 schedulable   nil
    drop-off-laundry            finished      t
    transfer-laundry            schedulable   nil
    pick-up-laundry             waiting       nil
    food-shopping               schedulable   nil
    go-home                     waiting       nil
    mail-letters                finished      t
End status SDA
    The schedule so far is:
        Move robot          (move robot from location home to laundr-o-mat at 30 mph
                                   (then drop off laundry))
        Reasons: (best-dd)
      drop-off-laundry               61 61  0
        Move robot          (move robot from location laundr-o-mat to po at 30 mph
                                   (the destination))
        Reasons: (best-dd closest-dl)
      mail-letters                   67 70  0
        Move robot          (move robot from location po to garage at 30 mph
                                   (then drop off car))
        Reasons: (lowest-rtt min-state-change)
      drop-off-car                   72 75  0

I6I

The status of the Schedule Dependency Array is:
    task                        status        countdown
    drop-off-car                schedulable   nil
    pick-up-car                 waiting       nil
    drop-off-laundry            finished      t
    transfer-laundry            finished      t
    pick-up-laundry             schedulable   nil
    food-shopping               schedulable   nil
    go-home                     waiting       nil
    mail-letters                finished      t
End status SDA
The tasks (drop-off-car 0 4 2894 62 (best-dd)
           food-shopping 0 2 2894 91 (min-state-change closest-dl)
           pick-up-laundry 60 0 90 60 (lowest-rtt best-dd min-state-change closest-dl))
   are ready to be scheduled.

                 ---Creating New Environment---:
    The environment at time 106 114 has position data
    -------------Start STATE OBJECT-------------
    The speed is in state: 30
    The car-loc is in state: robot
    The car-loc restriction:
        car-unavailable
    is active and will be terminated by the task: pick-up-car.

    The robot-loc is in state: laundr-o-mat
    The possessions is in state: (car)
    -------------End of STATE OBJECT-------------
The status of the Schedule Dependency Array is:
    task                        status        countdown
    drop-off-car                schedulable   nil
    pick-up-car                 waiting       nil
```

```
    drop-off-laundry              finished      t
    transfer-laundry              finished      t
    pick-up-laundry               schedulable   nil
    food-shopping                 schedulable   nil
    go-home                       waiting       nil
    mail-letters                  finished      t
End status SDA
    The schedule so far is:
        Move robot          (move robot from location home to laundr-o-mat at 30 mph
                                           (then drop off laundry))
        Reasons: (best-dd)
      drop-off-laundry                 61 61  0
        Move robot          (move robot from location laundr-o-mat to po at 30 mph
                                           (the destination))
        Reasons: (best-dd closest-dl)
      mail-letters                     67 70  0
        Move robot          (move robot from location po to laundr-o-mat at 30 mph
                                           (the destination))
        Reasons: (best-dd min-state-change closest-dl)
      transfer-laundry                 106 114  0

I7I

The status of the Schedule Dependency Array is:
    task                          status        countdown
    drop-off-car                  finished      t
    pick-up-car                   schedulable   nil
    drop-off-laundry              finished      t
    transfer-laundry              finished      t
    pick-up-laundry               schedulable   nil
    food-shopping                 schedulable   nil
    go-home                       waiting       nil
    mail-letters                  finished      t
End status SDA
The tasks (food-shopping 0 12 2889 96 (min-state-change)
           pick-up-car 120 0 150 60 (lowest-rtt best-dd min-state-change closest-dl)
           pick-up-laundry 55 24 85 72 (best-dd closest-dl))
    are ready to be scheduled.

Making Branches
Critically removing pick-up-car
For task pick-up-laundry
                    ---Creating New Environment---:
    The environment at time 111 119 has position data
    -------------Start STATE OBJECT-------------
    The speed is in state: 5
    The robot-loc is in state: garage
    The possessions is in state: nil
    The car-loc is in state: 'garage
    The car-loc restriction:
        car-unavailable
    is active and will be terminated by the task: pick-up-car.

    -------------End of STATE OBJECT-------------
The status of the Schedule Dependency Array is:
    task                          status        countdown
    drop-off-car                  finished      t
```

```
    pick-up-car                   schedulable    nil
    drop-off-laundry              finished       t
    transfer-laundry              finished       t
    pick-up-laundry               schedulable    nil
    food-shopping                 schedulable    nil
    go-home                       waiting        nil
    mail-letters                  finished       t
End status SDA
    The schedule so far is:
        Move robot          (move robot from location home to laundr-o-mat at 30 mph
                                      (then drop off laundry))
        Reasons: (best-dd)
      drop-off-laundry                61 61  0
        Move robot          (move robot from location laundr-o-mat to po at 30 mph
                                      (the destination))
        Reasons: (best-dd closest-dl)
      mail-letters                    67 70  0
        Move robot          (move robot from location po to laundr-o-mat at 30 mph
                                      (the destination))
        Reasons: (best-dd min-state-change closest-dl)
      transfer-laundry                106 114  0
        Move robot          (move robot from location laundr-o-mat to garage at 30 mph
                                      (then drop off car))
        Reasons: (best-dd)
      drop-off-car                    111 119  0

ISI

The status of the Schedule Dependency Array is:
    task                          status         countdown
    drop-off-car                  schedulable    nil
    pick-up-car                   waiting        nil
    drop-off-laundry              finished       t
    transfer-laundry              finished       t
    pick-up-laundry               schedulable    nil
    food-shopping                 finished       t
    go-home                       waiting        nil
    mail-letters                  finished       t
End status SDA
The tasks (pick-up-laundry 38 2 68 122 (lowest-rtt)
           drop-off-car 0 2 2872 122 nil)
    are ready to be scheduled.

                    ---Creating New Environment---:
    The environment at time 128 146 has position data
    -------------Start STATE OBJECT-------------
    The car-loc is in state: robot
    The car-loc restriction:
       car-unavailable
    is active and will be terminated by the task: pick-up-car.

    The speed is in state: 30
    The robot-loc is in state: grocery
    The possessions is in state: (car food)
    -------------End of STATE OBJECT-------------
The status of the Schedule Dependency Array is:
    task                          status         countdown
```

```
     drop-off-car                   schedulable    nil
     pick-up-car                    waiting        nil
     drop-off-laundry               finished       t
     transfer-laundry               finished       t
     pick-up-laundry                schedulable    nil
     food-shopping                  finished       t
     go-home                        waiting        nil
     mail-letters                   finished       t
End status SDA
     The schedule so far is:
          Move robot          (move robot from location home to laundr-o-mat at 30 mph
                                           (then drop off laundry))
          Reasons: (best-dd)
        drop-off-laundry               61 61  0
          Move robot          (move robot from location laundr-o-mat to po at 30 mph
                                           (the destination))
          Reasons: (best-dd closest-dl)
        mail-letters                   67 70  0
          Move robot          (move robot from location po to laundr-o-mat at 30 mph
                                           (the destination))
          Reasons: (best-dd min-state-change closest-dl)
        transfer-laundry               106 114  0
          Move robot          (move robot from location laundr-o-mat to grocery at 30 mph
                                           (then pickup food))
          Reasons: (min-state-change closest-dl)
        food-shopping                  128 146  0

I9I

The status of the Schedule Dependency Array is:
     task                           status         countdown
     drop-off-car                   finished       t
     pick-up-car                    schedulable    nil
     drop-off-laundry               finished       t
     transfer-laundry               finished       t
     pick-up-laundry                schedulable    nil
     food-shopping                  finished       t
     go-home                        waiting        nil
     mail-letters                   finished       t
End status SDA
The tasks (pick-up-laundry 35 40 65 160 nil
           pick-up-car 120 0 150 120 (lowest-rtt))
    are ready to be scheduled.

Making Branches
Critically removing pick-up-car
For task pick-up-laundry
                    ---Creating New Environment---:
     The environment at time 131 149 has position data
     -------------Start STATE OBJECT-------------
     The speed is in state: 3
     The robot-loc is in state: garage
     The possessions is in state: (food)
     The car-loc is in state: 'garage
     The car-loc restriction:
         car-unavailable
     is active and will be terminated by the task: pick-up-car.
```

```
    -------------End of STATE OBJECT-------------
The status of the Schedule Dependency Array is:
    task                           status         countdown
    drop-off-car                   finished       t
    pick-up-car                    schedulable    nil
    drop-off-laundry               finished       t
    transfer-laundry               finished       t
    pick-up-laundry                schedulable    nil
    food-shopping                  finished       t
    go-home                        waiting        nil
    mail-letters                   finished       t
End status SDA
    The schedule so far is:
        Move robot          (move robot from location home to laundr-o-mat at 30 mph
                                       (then drop off laundry))
        Reasons: (best-dd)
      drop-off-laundry                61 61  0
        Move robot          (move robot from location laundr-o-mat to po at 30 mph
                                       (the destination))
        Reasons: (best-dd closest-dl)
      mail-letters                    67 70  0
        Move robot          (move robot from location po to laundr-o-mat at 30 mph
                                       (the destination))
        Reasons: (best-dd min-state-change closest-dl)
      transfer-laundry                106 114  0
        Move robot          (move robot from location laundr-o-mat to grocery at 30 mph
                                       (then pickup food))
        Reasons: (min-state-change closest-dl)
      food-shopping                   128 146  0
        Move robot          (move robot from location grocery to garage at 30 mph
                                       (then drop off car))
      drop-off-car                    131 149  0

I10I

The status of the Schedule Dependency Array is:
    task                           status         countdown
    drop-off-car                   schedulable    nil
    pick-up-car                    waiting        nil
    drop-off-laundry               finished       t
    transfer-laundry               finished       t
    pick-up-laundry                finished       t
    food-shopping                  finished       t
    go-home                        waiting        nil
    mail-letters                   finished       t
End status SDA
The task (drop-off-car 0 4 2833 1 (only-choice)) is ready to be scheduled.

                  ---Creating New Environment---:
    The environment at time 167 185 has position data
    -------------Start STATE OBJECT-------------
    The speed is in state: 30
    The car-loc is in state: robot
    The car-loc restriction:
        car-unavailable
    is active and will be terminated by the task: pick-up-car.
```

```
     The robot-loc is in state: laundr-o-mat
     The possessions is in state: (food car laundry)
     -------------End of STATE OBJECT-------------
The status of the Schedule Dependency Array is:
     task                          status          countdown
     drop-off-car                  schedulable     nil
     pick-up-car                   waiting         nil
     drop-off-laundry              finished        t
     transfer-laundry              finished        t
     pick-up-laundry               finished        t
     food-shopping                 finished        t
     go-home                       waiting         nil
     mail-letters                  finished        t
End status SDA
     The schedule so far is:
          Move robot          (move robot from location home to laundr-o-mat at 30 mph
                                             (then drop off laundry))
          Reasons: (best-dd)
        drop-off-laundry                61 61  0
          Move robot          (move robot from location laundr-o-mat to po at 30 mph
                                             (the destination))
          Reasons: (best-dd closest-dl)
        mail-letters                    67 70  0
          Move robot          (move robot from location po to laundr-o-mat at 30 mph
                                             (the destination))
          Reasons: (best-dd min-state-change closest-dl)
        transfer-laundry                106 114  0
          Move robot          (move robot from location laundr-o-mat to grocery at 30 mph
                                             (then pickup food))
          Reasons: (min-state-change closest-dl)
        food-shopping                   128 146  0
          Move robot          (move robot from location grocery to laundr-o-mat at 30 mph
                                             (then pickup laundry))
          Reasons: (lowest-rtt)
        pick-up-laundry                 167 185  0

I11I

The status of the Schedule Dependency Array is:
     task                          status          countdown
     drop-off-car                  finished        t
     pick-up-car                   schedulable     nil
     drop-off-laundry              finished        t
     transfer-laundry              finished        t
     pick-up-laundry               finished        t
     food-shopping                 finished        t
     go-home                       waiting         nil
     mail-letters                  finished        t
End status SDA
The task (pick-up-car 120 0 150 2 (only-choice)) is ready to be scheduled.

                    ---Creating New Environment---:
     The environment at time 172 190 has position data
     -------------Start STATE OBJECT-------------
     The speed is in state: 1
     The robot-loc is in state: garage
```

```
     The possessions is in state: (laundry food)
     The car-loc is in state: 'garage
     The car-loc restriction:
         car-unavailable
     is active and will be terminated by the task: pick-up-car.

     -------------End of STATE OBJECT-------------
The status of the Schedule Dependency Array is:
     task                          status        countdown
     drop-off-car                  finished      t
     pick-up-car                   schedulable   nil
     drop-off-laundry              finished      t
     transfer-laundry              finished      t
     pick-up-laundry               finished      t
     food-shopping                 finished      t
     go-home                       waiting       nil
     mail-letters                  finished      t
End status SDA
     The schedule so far is:
          Move robot          (move robot from location home to laundr-o-mat at 30 mph
                                              (then drop off laundry))
          Reasons: (best-dd)
        drop-off-laundry                61 61  0
          Move robot          (move robot from location laundr-o-mat to po at 30 mph
                                              (the destination))
          Reasons: (best-dd closest-dl)
        mail-letters                    67 70  0
          Move robot          (move robot from location po to laundr-o-mat at 30 mph
                                              (the destination))
          Reasons: (best-dd min-state-change closest-dl)
        transfer-laundry                106 114  0
          Move robot          (move robot from location laundr-o-mat to grocery at 30 mph
                                              (then pickup food))
          Reasons: (min-state-change closest-dl)
        food-shopping                   128 146  0
          Move robot          (move robot from location grocery to laundr-o-mat at 30 mph
                                              (then pickup laundry))
          Reasons: (lowest-rtt)
        pick-up-laundry                 167 185  0
          Move robot          (move robot from location laundr-o-mat to garage at 30 mph
                                              (then drop off car))
          Reasons: (only-choice)
        drop-off-car                    172 190  0

I12I

The status of the Schedule Dependency Array is:
     task                          status        countdown
     drop-off-car                  finished      t
     pick-up-car                   schedulable   nil
     drop-off-laundry              finished      t
     transfer-laundry              finished      t
     pick-up-laundry               schedulable   nil
     food-shopping                 finished      t
     go-home                       waiting       nil
     mail-letters                  finished      t
End status SDA
```

```
The tasks (pick-up-laundry 23 20 53 140 (lowest-rtt)
           pick-up-car 88 20 118 140 nil)
    are ready to be scheduled.

Making Branches
Critically removing pick-up-car
For task pick-up-laundry
                    ---Creating New Environment---:
     The environment at time 143 161 has position data
     -------------Start STATE OBJECT-------------
     The car-loc is in state: 'garage
     The car-loc restriction:
         car-unavailable
     is active and will be terminated by the task: pick-up-car.

     The speed is in state: 3
     The robot-loc is in state: grocery
     The possessions is in state: (food)
     -------------End of STATE OBJECT-------------
The status of the Schedule Dependency Array is:
     task                          status         countdown
     drop-off-car                  finished       t
     pick-up-car                   schedulable    nil
     drop-off-laundry              finished       t
     transfer-laundry              finished       t
     pick-up-laundry               schedulable    nil
     food-shopping                 finished       t
     go-home                       waiting        nil
     mail-letters                  finished       t
End status SDA
     The schedule so far is:
          Move robot          (move robot from location home to laundr-o-mat at 30 mph
                                              (then drop off laundry))
          Reasons: (best-dd)
        drop-off-laundry                61 61  0
          Move robot          (move robot from location laundr-o-mat to po at 30 mph
                                              (the destination))
          Reasons: (best-dd closest-dl)
        mail-letters                    67 70  0
          Move robot          (move robot from location po to laundr-o-mat at 30 mph
                                              (the destination))
          Reasons: (best-dd min-state-change closest-dl)
        transfer-laundry                106 114  0
          Move robot          (move robot from location laundr-o-mat to garage at 30 mph
                                              (then drop off car))
          Reasons: (best-dd)
        drop-off-car                    111 119  0
          Move robot          (move robot from location garage to grocery at 5 mph
                                              (then pickup food))
          Reasons: (min-state-change)
        food-shopping                   143 161  0

I13I

The status of the Schedule Dependency Array is:
     task                          status         countdown
     drop-off-car                  finished       t
```

```
    pick-up-car                  schedulable   nil
    drop-off-laundry             finished      t
    transfer-laundry             finished      t
    pick-up-laundry              schedulable   nil
    food-shopping                schedulable   nil
    go-home                      waiting       nil
    mail-letters                 finished      t
End status SDA
The tasks (food-shopping 0 12 2894 96 (min-state-change)
           pick-up-car 86 24 116 72 (best-dd closest-dl)
           pick-up-laundry 60 0 90 60 (lowest-rtt best-dd min-state-change closest-dl))
    are ready to be scheduled.

                  ---Creating New Environment---:
    The environment at time 106 114 has position data
    -------------Start STATE OBJECT-------------
    The car-loc is in state: 'garage
    The car-loc restriction:
        car-unavailable
    is active and will be terminated by the task: pick-up-car.

    The speed is in state: 5
    The robot-loc is in state: laundr-o-mat
    The possessions is in state: nil
    -------------End of STATE OBJECT-------------
The status of the Schedule Dependency Array is:
    task                         status        countdown
    drop-off-car                 finished      t
    pick-up-car                  schedulable   nil
    drop-off-laundry             finished      t
    transfer-laundry             finished      t
    pick-up-laundry              schedulable   nil
    food-shopping                schedulable   nil
    go-home                      waiting       nil
    mail-letters                 finished      t
End status SDA
    The schedule so far is:
        Move robot          (move robot from location home to laundr-o-mat at 30 mph
                                            (then drop off laundry))
        Reasons: (best-dd)
      drop-off-laundry                61 61  0
        Move robot          (move robot from location laundr-o-mat to po at 30 mph
                                            (the destination))
        Reasons: (best-dd closest-dl)
      mail-letters                    67 70  0
        Move robot          (move robot from location po to garage at 30 mph
                                            (then drop off car))
        Reasons: (lowest-rtt min-state-change)
      drop-off-car                    72 75  0
        Move robot          (move robot from location garage to laundr-o-mat at 5 mph
                                            (the destination))
        Reasons: (best-dd closest-dl)
      transfer-laundry                106 114  0

I14I

The status of the Schedule Dependency Array is:
```

```
     task                          status        countdown
     drop-off-car                  finished      t
     pick-up-car                   schedulable   nil
     drop-off-laundry              finished      t
     transfer-laundry              finished      t
     pick-up-laundry               schedulable   nil
     food-shopping                 finished      t
     go-home                       waiting       nil
     mail-letters                  finished      t
End status SDA
The tasks (pick-up-laundry 28 20 58 140 (lowest-rtt)
           pick-up-car 54 20 84 140 nil)
    are ready to be scheduled.

                ---Creating New Environment---:
     The environment at time 138 156 has position data
     -------------Start STATE OBJECT-------------
     The speed is in state: 3
     The car-loc is in state: 'garage
     The car-loc restriction:
        car-unavailable
     is active and will be terminated by the task: pick-up-car.

     The robot-loc is in state: grocery
     The possessions is in state: (food)
     -------------End of STATE OBJECT-------------
The status of the Schedule Dependency Array is:
     task                          status        countdown
     drop-off-car                  finished      t
     pick-up-car                   schedulable   nil
     drop-off-laundry              finished      t
     transfer-laundry              finished      t
     pick-up-laundry               schedulable   nil
     food-shopping                 finished      t
     go-home                       waiting       nil
     mail-letters                  finished      t
End status SDA
     The schedule so far is:
        Move robot          (move robot from location home to laundr-o-mat at 30 mph
                                           (then drop off laundry))
        Reasons: (best-dd)
      drop-off-laundry               61 61  0
        Move robot          (move robot from location laundr-o-mat to po at 30 mph
                                           (the destination))
        Reasons: (best-dd closest-dl)
      mail-letters                   67 70  0
        Move robot          (move robot from location po to garage at 30 mph
                                           (then drop off car))
        Reasons: (lowest-rtt min-state-change)
      drop-off-car                   72 75  0
        Move robot          (move robot from location garage to laundr-o-mat at 5 mph
                                           (the destination))
        Reasons: (best-dd closest-dl)
      transfer-laundry               106 114  0
        Move robot          (move robot from location laundr-o-mat to grocery at 5 mph
                                           (then pickup food))
        Reasons: (min-state-change)
```

```
        food-shopping                  136 156  0

I15I

The status of the Schedule Dependency Array is:
     task                          status        countdown
     drop-off-car                  finished      t
     pick-up-car                   schedulable   nil
     drop-off-laundry              finished      t
     transfer-laundry              finished      t
     pick-up-laundry               finished      t
     food-shopping                 schedulable   nil
     go-home                       waiting       nil
     mail-letters                  finished      t
End status SDA
The tasks (food-shopping 0 20 2833 80 (lowest-rtt)
           pick-up-car 25 40 55 100 nil)
    are ready to be scheduled.

                  ---Creating New Environment---:
     The environment at time 167 175 has position data
     -------------Start STATE OBJECT-------------
     The speed is in state: 3
     The car-loc is in state: 'garage
     The car-loc restriction:
         car-unavailable
     is active and will be terminated by the task: pick-up-car.

     The robot-loc is in state: laundr-o-mat
     The possessions is in state: (laundry)
     -------------End of STATE OBJECT-------------
The status of the Schedule Dependency Array is:
     task                          status        countdown
     drop-off-car                  finished      t
     pick-up-car                   schedulable   nil
     drop-off-laundry              finished      t
     transfer-laundry              finished      t
     pick-up-laundry               finished      t
     food-shopping                 schedulable   nil
     go-home                       waiting       nil
     mail-letters                  finished      t
End status SDA
     The schedule so far is:
          Move robot          (move robot from location home to laundr-o-mat at 30 mph
                                              (then drop off laundry))
          Reasons: (best-dd)
        drop-off-laundry                 61 61  0
          Move robot          (move robot from location laundr-o-mat to po at 30 mph
                                              (the destination))
          Reasons: (best-dd closest-dl)
        mail-letters                     67 70  0
          Move robot          (move robot from location po to garage at 30 mph
                                              (then drop off car))
          Reasons: (lowest-rtt min-state-change)
        drop-off-car                     72 75  0
          Move robot          (move robot from location garage to laundr-o-mat at 5 mph
                                              (the destination))
```

```
        Reasons: (best-dd closest-dl)
      transfer-laundry                 106 114  0
        Move robot          (the robot should pickup laundry)
        Reasons: (lowest-rtt best-dd min-state-change closest-dl)
      pick-up-laundry                  167 175  0

I16I

The status of the Schedule Dependency Array is:
     task                          status        countdown
     drop-off-car                  finished       t
     pick-up-car                   schedulable    nil
     drop-off-laundry              finished       t
     transfer-laundry              schedulable    nil
     pick-up-laundry               waiting        nil
     food-shopping                 finished       t
     go-home                       waiting        nil
     mail-letters                  finished       t
End status SDA
The tasks (transfer-laundry 0 20 27 140 (lowest-rtt)
          pick-up-car 88 20 118 140 nil)
    are ready to be scheduled.

Making Branches
For task transfer-laundry
Critically removing pick-up-car
                    ---Creating New Environment---:
     The environment at time 104 117 has position data
     -------------Start STATE OBJECT-------------
     The car-loc is in state: 'garage
     The car-loc restriction:
         car-unavailable
     is active and will be terminated by the task: pick-up-car.

     The speed is in state: 3
     The robot-loc is in state: grocery
     The possessions is in state: (food)
     -------------End of STATE OBJECT-------------
The status of the Schedule Dependency Array is:
     task                          status        countdown
     drop-off-car                  finished       t
     pick-up-car                   schedulable    nil
     drop-off-laundry              finished       t
     transfer-laundry              schedulable    nil
     pick-up-laundry               waiting        nil
     food-shopping                 finished       t
     go-home                       waiting        nil
     mail-letters                  finished       t
End status SDA
     The schedule so far is:
          Move robot          (move robot from location home to laundr-o-mat at 30 mph
                                   (then drop off laundry))
        Reasons: (best-dd)
      drop-off-laundry                  61 61  0
        Move robot          (move robot from location laundr-o-mat to po at 30 mph
                                   (the destination))
        Reasons: (best-dd closest-dl)
```

```
      mail-letters                   67 70  0
        Move robot           (move robot from location po to garage at 30 mph
                                            (then drop off car))
        Reasons: (lowest-rtt min-state-change)
      drop-off-car                   72 75  0
        Move robot           (move robot from location garage to grocery at 5 mph
                                            (then pickup food))
        Reasons: (min-state-change)
      food-shopping                  104 117  0

I17I

The status of the Schedule Dependency Array is:
     task                          status         countdown
     drop-off-car                  finished       t
     pick-up-car                   schedulable    nil
     drop-off-laundry              finished       t
     transfer-laundry              finished       t
     pick-up-laundry               finished       t
     food-shopping                 schedulable    nil
     go-home                       waiting        nil
     mail-letters                  finished       t
End status SDA
The tasks (food-shopping 0 20 2833 80 (lowest-rtt)
           pick-up-car 64 40 94 100 nil)
    are ready to be scheduled.

                ---Creating New Environment---:
     The environment at time 167 175 has position data
     -------------Start STATE OBJECT-------------
     The car-loc is in state: 'garage
     The car-loc restriction:
         car-unavailable
     is active and will be terminated by the task: pick-up-car.

     The speed is in state: 3
     The robot-loc is in state: laundr-o-mat
     The possessions is in state: (laundry)
     -------------End of STATE OBJECT-------------
The status of the Schedule Dependency Array is:
     task                          status         countdown
     drop-off-car                  finished       t
     pick-up-car                   schedulable    nil
     drop-off-laundry              finished       t
     transfer-laundry              finished       t
     pick-up-laundry               finished       t
     food-shopping                 schedulable    nil
     go-home                       waiting        nil
     mail-letters                  finished       t
End status SDA
     The schedule so far is:
        Move robot           (move robot from location home to laundr-o-mat at 30 mph
                                            (then drop off laundry))
        Reasons: (best-dd)
      drop-off-laundry               61 61  0
        Move robot           (move robot from location laundr-o-mat to po at 30 mph
                                            (the destination))
```

```
  Reasons: (best-dd closest-dl)
mail-letters                      67 70  0
  Move robot            (move robot from location po to laundr-o-mat at 30 mph
                                 (the destination))
  Reasons: (best-dd min-state-change closest-dl)
transfer-laundry                  106 114  0
  Move robot            (move robot from location laundr-o-mat to garage at 30 mph
                                 (then drop off car))
  Reasons: (best-dd)
drop-off-car                      111 119  0
  Move robot            (move robot from location garage to laundr-o-mat at 5 mph
                                 (then pickup laundry))
  Reasons: (best-dd closest-dl)
pick-up-laundry                   167 175  0

I18I

The status of the Schedule Dependency Array is:
     task                          status         countdown
     drop-off-car                  finished       t
     pick-up-car                   schedulable    nil
     drop-off-laundry              finished       t
     transfer-laundry              finished       t
     pick-up-laundry               schedulable    nil
     food-shopping                 schedulable    nil
     go-home                       waiting        nil
     mail-letters                  schedulable    nil
End status SDA
The tasks (mail-letters 0 24 2894 108 nil
          food-shopping 0 12 2894 104 (min-state-change)
          pick-up-car 80 24 110 88 (best-dd closest-dl)
          pick-up-laundry 60 0 90 80 (lowest-rtt best-dd min-state-change closest-dl))
    are ready to be scheduled.

                 ---Creating New Environment---:
     The environment at time 106 111 has position data
     -------------Start STATE OBJECT-------------
     The car-loc is in state: 'garage
     The car-loc restriction:
         car-unavailable
     is active and will be terminated by the task: pick-up-car.

     The speed is in state: 5
     The robot-loc is in state: laundr-o-mat
     The possessions is in state: nil
     -------------End of STATE OBJECT-------------
The status of the Schedule Dependency Array is:
     task                          status         countdown
     drop-off-car                  finished       t
     pick-up-car                   schedulable    nil
     drop-off-laundry              finished       t
     transfer-laundry              finished       t
     pick-up-laundry               schedulable    nil
     food-shopping                 schedulable    nil
     go-home                       waiting        nil
     mail-letters                  schedulable    nil
End status SDA
```

```
     The schedule so far is:
          Move robot          (move robot from location home to laundr-o-mat at 30 mph
                                              (then drop off laundry))
          Reasons: (best-dd)
        drop-off-laundry                61 61  0
          Move robot          (move robot from location laundr-o-mat to garage at 30 mph
                                              (then drop off car))
        drop-off-car                    66 66  0
          Move robot          (move robot from location garage to laundr-o-mat at 5 mph
                                              (the destination))
          Reasons: (best-dd closest-dl)
        transfer-laundry                106 111  0

I19I

The status of the Schedule Dependency Array is:
     task                          status         countdown
     drop-off-car                  finished       t
     pick-up-car                   schedulable    nil
     drop-off-laundry              finished       t
     transfer-laundry              finished       t
     pick-up-laundry               schedulable    nil
     food-shopping                 finished       t
     go-home                       waiting        nil
     mail-letters                  schedulable    nil
End status SDA
The tasks (mail-letters 0 40 2862 140 nil
           pick-up-car 48 20 78 130 (lowest-rtt best-dd min-state-change closest-dl)
           pick-up-laundry 28 20 58 130 (best-dd min-state-change closest-dl))
    are ready to be scheduled.

                ---Creating New Environment---:
     The environment at time 138 153 has position data
     -------------Start STATE OBJECT-------------
     The speed is in state: 3
     The car-loc is in state: 'garage
     The car-loc restriction:
         car-unavailable
     is active and will be terminated by the task: pick-up-car.

     The robot-loc is in state: grocery
     The possessions is in state: (food)
     -------------End of STATE OBJECT-------------
The status of the Schedule Dependency Array is:
     task                          status         countdown
     drop-off-car                  finished       t
     pick-up-car                   schedulable    nil
     drop-off-laundry              finished       t
     transfer-laundry              finished       t
     pick-up-laundry               schedulable    nil
     food-shopping                 finished       t
     go-home                       waiting        nil
     mail-letters                  schedulable    nil
End status SDA
     The schedule so far is:
          Move robot          (move robot from location home to laundr-o-mat at 30 mph
                                              (then drop off laundry))
```

```
    Reasons: (best-dd)
  drop-off-laundry                61 61  0
    Move robot          (move robot from location laundr-o-mat to garage at 30 mph
                                (then drop off car))
  drop-off-car                    66 66  0
    Move robot          (move robot from location garage to laundr-o-mat at 5 mph
                                (the destination))
    Reasons: (best-dd closest-dl)
  transfer-laundry                106 111  0
    Move robot          (move robot from location laundr-o-mat to grocery at 5 mph
                                (then pickup food))
    Reasons: (min-state-change)
  food-shopping                   138 153  0

I20I

The status of the Schedule Dependency Array is:
     task                         status         countdown
     drop-off-car                 finished       t
     pick-up-car                  schedulable    nil
     drop-off-laundry             finished       t
     transfer-laundry             finished       t
     pick-up-laundry              schedulable    nil
     food-shopping                schedulable    nil
     go-home                      waiting        nil
     mail-letters                 finished       t
End status SDA
The tasks (food-shopping 0 24 2868 102 nil
           pick-up-car 54 24 84 72 (lowest-rtt best-dd min-state-change closest-dl)
           pick-up-laundry 34 24 64 72 (best-dd min-state-change closest-dl))
    are ready to be scheduled.

                  ---Creating New Environment---:
     The environment at time 132 140 has position data
     -------------Start STATE OBJECT-------------
     The speed is in state: 5
     The car-loc is in state: 'garage
     The car-loc restriction:
         car-unavailable
     is active and will be terminated by the task: pick-up-car.

     The robot-loc is in state: po
     The possessions is in state: nil
     -------------End of STATE OBJECT-------------
The status of the Schedule Dependency Array is:
     task                         status         countdown
     drop-off-car                 finished       t
     pick-up-car                  schedulable    nil
     drop-off-laundry             finished       t
     transfer-laundry             finished       t
     pick-up-laundry              schedulable    nil
     food-shopping                schedulable    nil
     go-home                      waiting        nil
     mail-letters                 finished       t
End status SDA
     The schedule so far is:
          Move robot          (move robot from location home to laundr-o-mat at 30 mph
```

```
                                       (then drop off laundry))
    Reasons: (best-dd)
  drop-off-laundry              61 61  0
    Move robot          (move robot from location laundr-o-mat to garage at 30 mph
                                       (then drop off car))
  drop-off-car                  66 66  0
    Move robot          (move robot from location garage to laundr-o-mat at 5 mph
                                       (the destination))
    Reasons: (best-dd closest-dl)
  transfer-laundry              106 111  0
    Move robot          (move robot from location laundr-o-mat to po at 5 mph
                                       (the destination))
  mail-letters                  132 140  0

I21I

The status of the Schedule Dependency Array is:
     task                          status         countdown
     drop-off-car                  finished       t
     pick-up-car                   schedulable    nil
     drop-off-laundry              finished       t
     transfer-laundry              finished       t
     pick-up-laundry               finished       t
     food-shopping                 schedulable    nil
     go-home                       waiting        nil
     mail-letters                  finished       t
End status SDA
The tasks (food-shopping 0 20 2833 80 (lowest-rtt)
          pick-up-car 19 40 49 100 nil)
    are ready to be scheduled.

                ---Creating New Environment---:
     The environment at time 167 175 has position data
     -------------Start STATE OBJECT-------------
     The car-loc is in state: 'garage
     The car-loc restriction:
         car-unavailable
     is active and will be terminated by the task: pick-up-car.

     The speed is in state: 3
     The robot-loc is in state: laundr-o-mat
     The possessions is in state: (laundry)
     -------------End of STATE OBJECT-------------
The status of the Schedule Dependency Array is:
     task                          status         countdown
     drop-off-car                  finished       t
     pick-up-car                   schedulable    nil
     drop-off-laundry              finished       t
     transfer-laundry              finished       t
     pick-up-laundry               finished       t
     food-shopping                 schedulable    nil
     go-home                       waiting        nil
     mail-letters                  finished       t
End status SDA
     The schedule so far is:
          Move robot          (move robot from location home to laundr-o-mat at 30 mph
                                             (then drop off laundry))
```

```
        Reasons: (best-dd)
      drop-off-laundry                61 61  0
        Move robot          (move robot from location laundr-o-mat to garage at 30 mph
                                            (then drop off car))
      drop-off-car                    66 66  0
        Move robot          (move robot from location garage to laundr-o-mat at 5 mph
                                            (the destination))
        Reasons: (best-dd closest-dl)
      transfer-laundry                106 111  0
        Move robot          (move robot from location laundr-o-mat to po at 5 mph
                                            (the destination))
      mail-letters                    132 140  0
        Move robot          (move robot from location po to laundr-o-mat at 5 mph
                                            (then pickup laundry))
        Reasons: (best-dd min-state-change closest-dl)
      pick-up-laundry                 167 175  0

I22I

The status of the Schedule Dependency Array is:
     task                          status        countdown
     drop-off-car                  finished       t
     pick-up-car                   schedulable    nil
     drop-off-laundry              finished       t
     transfer-laundry              finished       t
     pick-up-laundry               schedulable    nil
     food-shopping                 finished       t
     go-home                       waiting        nil
     mail-letters                  finished       t
End status SDA
The tasks (pick-up-laundry 0 20 20 140 (lowest-rtt)
           pick-up-car 10 20 40 140 nil)
    are ready to be scheduled.

                     ---Creating New Environment---:
     The environment at time 176 194 has position data
     -------------Start STATE OBJECT-------------
     The car-loc is in state: 'garage
     The car-loc restriction:
         car-unavailable
     is active and will be terminated by the task: pick-up-car.

     The speed is in state: 3
     The robot-loc is in state: grocery
     The possessions is in state: (food)
     -------------End of STATE OBJECT-------------
The status of the Schedule Dependency Array is:
     task                          status        countdown
     drop-off-car                  finished       t
     pick-up-car                   schedulable    nil
     drop-off-laundry              finished       t
     transfer-laundry              finished       t
     pick-up-laundry               schedulable    nil
     food-shopping                 finished       t
     go-home                       waiting        nil
     mail-letters                  finished       t
End status SDA
```

```
     The schedule so far is:
          Move robot          (move robot from location home to laundr-o-mat at 30 mph
                                              (then drop off laundry))
          Reasons: (best-dd)
        drop-off-laundry                61 61  0
          Move robot          (move robot from location laundr-o-mat to garage at 30 mph
                                              (then drop off car))
        drop-off-car                    66 66  0
          Move robot          (move robot from location garage to laundr-o-mat at 5 mph
                                              (the destination))
          Reasons: (best-dd closest-dl)
        transfer-laundry                106 111  0
          Move robot          (move robot from location laundr-o-mat to po at 5 mph
                                              (the destination))
        mail-letters                    132 140  0
          Move robot          (move robot from location po to grocery at 5 mph
                                              (then pickup food))
        food-shopping                   176 194  0

I23I

The status of the Schedule Dependency Array is:
     task                          status         countdown
     drop-off-car                  finished       t
     pick-up-car                   schedulable    nil
     drop-off-laundry              finished       t
     transfer-laundry              schedulable    nil
     pick-up-laundry               waiting        nil
     food-shopping                 finished       t
     go-home                       waiting        nil
     mail-letters                  schedulable    nil
End status SDA
The tasks (mail-letters 0 40 2902 140 nil
           pick-up-car 82 20 118 130 (lowest-rtt best-dd min-state-change closest-dl)
           transfer-laundry 3 20 33 130 (best-dd min-state-change closest-dl))
    are ready to be scheduled.

Making Branches
Critically removing mail-letters
Critically removing pick-up-car
For task transfer-laundry
                    ---Creating New Environment---:
     The environment at time 98 108 has position data
     -------------Start STATE OBJECT-------------
     The car-loc is in state: 'garage
     The car-loc restriction:
         car-unavailable
     is active and will be terminated by the task: pick-up-car.

     The speed is in state: 3
     The robot-loc is in state: grocery
     The possessions is in state: (food)
     -------------End of STATE OBJECT-------------
The status of the Schedule Dependency Array is:
     task                          status         countdown
     drop-off-car                  finished       t
     pick-up-car                   schedulable    nil
```

```
    drop-off-laundry              finished       t
    transfer-laundry              schedulable    nil
    pick-up-laundry               waiting        nil
    food-shopping                 finished       t
    go-home                       waiting        nil
    mail-letters                  schedulable    nil
End status SDA
    The schedule so far is:
        Move robot          (move robot from location home to laundr-o-mat at 30 mph
                                        (then drop off laundry))
        Reasons: (best-dd)
      drop-off-laundry               61 61  0
        Move robot          (move robot from location laundr-o-mat to garage at 30 mph
                                        (then drop off car))
      drop-off-car                   66 66  0
        Move robot          (move robot from location garage to grocery at 5 mph
                                        (then pickup food))
        Reasons: (min-state-change)
      food-shopping                  98 108  0

I24I

The status of the Schedule Dependency Array is:
    task                          status         countdown
    drop-off-car                  finished       t
    pick-up-car                   schedulable    nil
    drop-off-laundry              finished       t
    transfer-laundry              finished       t
    pick-up-laundry               finished       t
    food-shopping                 finished       t
    go-home                       waiting        nil
    mail-letters                  finished       t
End status SDA
```

****

Unfortunately this is an infeasible schedule. Going shopping at this point makes it impossible to get back to the garage before the car has been sold for parts. This impending deadline violation causes the schedule to be abandoned and the scoring weights to be changed.

****

```
!!!Impending deadline violation on task pick-up-car
P-delay 24              ;;; Prerequisite delay
S-delay 60              ;;; State transition delay
Deadline 54

$$$$$$$$$Deadline Violation$$$$$$$
Task pick-up-car has busted its deadline.
$$$$$$End Deadline Violation$$$$$$
                    !!!!Failed Environment!!!!
                    The following prefix deadends
    The schedule so far is:
        Move robot          (move robot from location home to laundr-o-mat at 30 mph
                                        (then drop off laundry))
        Reasons: (best-dd)
      drop-off-laundry               61 61  0
        Move robot          (move robot from location laundr-o-mat to po at 30 mph
```

```
                                           (the destination))
      Reasons: (best-dd closest-dl)
    mail-letters                    67 70  0
      Move robot           (move robot from location po to laundr-o-mat at 30 mph
                                           (the destination))
      Reasons: (best-dd min-state-change closest-dl)
    transfer-laundry                106 114  0
      Move robot           (move robot from location laundr-o-mat to garage at 30 mph
                                           (then drop off car))
      Reasons: (best-dd)
    drop-off-car                    111 119  0
      Move robot           (move robot from location garage to laundr-o-mat at 5 mph
                                           (then pickup laundry))
      Reasons: (best-dd closest-dl)
    pick-up-laundry                 167 175  0
      Move robot           (move robot from location laundr-o-mat to grocery at 3 mph
                                           (then pickup food))
      Reasons: (lowest-rtt)
    food-shopping                   207 225  0

                  !!!!End Failed Environment!!!!
I25I

------------Weights------------
The task weight factor is: 8
The approaching deadline factor is: 15
The RTT factor is: 11
The domain heuristics factor is: 13
The zero state delay factor is: 100
The waste time factor is: 8
----------End--Weights---------
The status of the Schedule Dependency Array is:
     task                          status          countdown
     drop-off-car                  finished        t
     pick-up-car                   finished        t
     drop-off-laundry              finished        t
     transfer-laundry              finished        t
     pick-up-laundry               finished        t
     food-shopping                 schedulable     nil
     go-home                       waiting         nil
     mail-letters                  finished        t
End status SDA
The task (food-shopping 0 2 2792 6 (only-choice)) is ready to be scheduled.

                    ---Creating New Environment---:
     The environment at time 208 216 has position data
     -------------Start STATE OBJECT-------------
     The speed is in state: 30
     The robot-loc is in state: garage
     The possessions is in state: (laundry car)
     The car-loc is in state: 'robot
     -------------End of STATE OBJECT-------------
The status of the Schedule Dependency Array is:
     task                          status          countdown
     drop-off-car                  finished        t
     pick-up-car                   finished        t
     drop-off-laundry              finished        t
```

```
    transfer-laundry              finished       t
    pick-up-laundry               finished       t
    food-shopping                 schedulable    nil
    go-home                       waiting        nil
    mail-letters                  finished       t
End status SDA
    The schedule so far is:
        Move robot          (move robot from location home to laundr-o-mat at 30 mph
                                 (then drop off laundry))
        Reasons: (best-dd)
      drop-off-laundry               61 61  0
        Move robot          (move robot from location laundr-o-mat to po at 30 mph
                                 (the destination))
        Reasons: (best-dd closest-dl)
      mail-letters                   67 70  0
        Move robot          (move robot from location po to garage at 30 mph
                                 (then drop off car))
        Reasons: (lowest-rtt min-state-change)
      drop-off-car                   72 75  0
        Move robot          (move robot from location garage to laundr-o-mat at 5 mph
                                 (the destination))
        Reasons: (best-dd closest-dl)
      transfer-laundry               106 114  0
        Move robot          (the robot should pickup laundry)
        Reasons: (lowest-rtt best-dd min-state-change closest-dl)
      pick-up-laundry                167 175  0
        Move robot          (move robot from location laundr-o-mat to garage at 3 mph
                                 (then pickup car))
      pick-up-car                    208 216  0

I26I
.
------------Weights------------
The task weight factor is: 8
The approaching deadline factor is: 15
The RTT factor is: 11
The domain heuristics factor is: 13
The zero state delay factor is: 100
The waste time factor is: 8
----------End--Weights---------
The status of the Schedule Dependency Array is:
    task                          status         countdown
    drop-off-car                  finished       t
    pick-up-car                   finished       t
    drop-off-laundry              finished       t
    transfer-laundry              finished       t
    pick-up-laundry               finished       t
    food-shopping                 finished       t
    go-home                       schedulable    nil
    mail-letters                  finished       t
End status SDA
```

****

Backing up and picking up the car first is not sufficient. The trouble with this schedule is deeper than that. The weights will be reinforced and more backtracking performed.

****

```
!!!Impending deadline violation on task go-home
P-delay 0
S-delay 62
Deadline 60

$$$$$$$$$Deadline Violation$$$$$$$
Task go-home has busted its deadline.
$$$$$$End Deadline Violation$$$$$$
                    !!!!Failed Environment!!!!
                    The following prefix deadends
     The schedule so far is:
          Move robot          (move robot from location home to laundr-o-mat at 30 mph
                                              (then drop off laundry))
          Reasons: (best-dd)
        drop-off-laundry                61 61  0
          Move robot          (move robot from location laundr-o-mat to po at 30 mph
                                              (the destination))
          Reasons: (best-dd closest-dl)
        mail-letters                    67 70  0
          Move robot          (move robot from location po to garage at 30 mph
                                              (then drop off car))
          Reasons: (lowest-rtt min-state-change)
        drop-off-car                    72 75  0
          Move robot          (move robot from location garage to laundr-o-mat at 5 mph
                                              (the destination))
          Reasons: (best-dd closest-dl)
        transfer-laundry                106 114  0
          Move robot          (the robot should pickup laundry)
          Reasons: (lowest-rtt best-dd min-state-change closest-dl)
        pick-up-laundry                 167 175  0
          Move robot          (move robot from location laundr-o-mat to garage at 3 mph
                                              (then pickup car))
        pick-up-car                     208 216  0
          Move robot          (move robot from location garage to grocery at 30 mph
                                              (then pickup food))
          Reasons: (only-choice)
        food-shopping                   230 248  0

                    !!!!End Failed Environment!!!!
I27I

------------Weights------------
The task weight factor is: 8
The approaching deadline factor is: 16
The RTT factor is: 11
The domain heuristics factor is: 13
The zero state delay factor is: 100
The waste time factor is: 8
----------End--Weights---------

*****************************************************************
```

Now the scheduler notices that its clothes may get seriously wrinkled if its schedule is not rearranged. TTwo failures in a row are serious business. The changes in the weights reflect that.

```
*****************************************************************
```

```
------PREREQUISITE DEADLINE FAILURE------
Task pick-up-laundry was not executed soon enough after task transfer-laundry.
----------------------------------.
                    !!!!Failed Environment!!!!
                    The following prefix blows a deadline
     The schedule so far is:
          Move robot          (move robot from location home to laundr-o-mat at 30 mph
                                              (then drop off laundry))
          Reasons: (best-dd)
        drop-off-laundry                61 61  0
          Move robot          (move robot from location laundr-o-mat to garage at 30 mph
                                              (then drop off car))
        drop-off-car                    66 66  0
          Move robot          (move robot from location garage to laundr-o-mat at 5 mph
                                              (the destination))
          Reasons: (best-dd closest-dl)
        transfer-laundry                106 111  0
          Move robot          (move robot from location laundr-o-mat to po at 5 mph
                                              (the destination))
        mail-letters                    132 140  0
          Move robot          (move robot from location po to grocery at 5 mph
                                              (then pickup food))
        food-shopping                   176 194  0
          Move robot          (move robot from location grocery to laundr-o-mat at 3 mph
                                              (then pickup laundry))
          Reasons: (perfect-dd perfect-dd perfect-dd perfect-dd lowest-rtt)
        pick-up-laundry                 197 215  0

                    !!!!End Failed Environment!!!!..
I28I
..
------------Weights------------
The task weight factor is: 7
The approaching deadline factor is: 24
The RTT factor is: 10
The domain heuristics factor is: 13
The zero state delay factor is: 99
The waste time factor is: 8
----------End--Weights---------
The status of the Schedule Dependency Array is:
     task                          status         countdown
     drop-off-car                  finished       t
     pick-up-car                   schedulable    nil
     drop-off-laundry              finished       t
     transfer-laundry              finished       t
     pick-up-laundry               finished       t
     food-shopping                 finished       t
     go-home                       waiting        nil
     mail-letters                  finished       t
End status SDA
!!!Impending deadline violation on task pick-up-car
P-delay 0
S-delay 60
Deadline 15

*******************************************************************
```

Now we get a straight deadline failure. All of the deadlines

```
   become critical at the same point in this problem (this is
   what makes it so difficult). The scheduler backs up again and
   rebalances its weights.
*********************************************************************

$$$$$$$$$Deadline Violation$$$$$$$
Task pick-up-car has busted its deadline.
$$$$$$End Deadline Violation$$$$$$
                    !!!!Failed Environment!!!!
                    The following prefix deadends
     The schedule so far is:
          Move robot          (move robot from location home to laundr-o-mat at 30 mph
                                              (then drop off laundry))
          Reasons: (best-dd)
        drop-off-laundry              61 61  0
          Move robot          (move robot from location laundr-o-mat to po at 30 mph
                                              (the destination))
          Reasons: (best-dd closest-dl)
        mail-letters                  67 70  0
          Move robot          (move robot from location po to garage at 30 mph
                                              (then drop off car))
          Reasons: (lowest-rtt min-state-change)
        drop-off-car                  72 75  0
          Move robot          (move robot from location garage to laundr-o-mat at 5 mph
                                              (the destination))
          Reasons: (best-dd closest-dl)
        transfer-laundry              106 114  0
          Move robot          (the robot should pickup laundry)
          Reasons: (lowest-rtt best-dd min-state-change closest-dl)
        pick-up-laundry               167 175  0
          Move robot          (move robot from location laundr-o-mat to grocery at 3 mph
                                              (then pickup food))
          Reasons: (lowest-rtt)
        food-shopping                 207 225  0

                    !!!!End Failed Environment!!!!
I29I

-------------Weights-------------
The task weight factor is: 8
The approaching deadline factor is: 25
The RTT factor is: 11
The domain heuristics factor is: 13
The zero state delay factor is: 100
The waste time factor is: 8
-----------End--Weights----------
The status of the Schedule Dependency Array is:
     task                         status          countdown
     drop-off-car                 finished        t
     pick-up-car                  finished        t
     drop-off-laundry             finished        t
     transfer-laundry             finished        t
     pick-up-laundry              finished        t
     food-shopping                schedulable     nil
     go-home                      waiting         nil
     mail-letters                 finished        t
End status SDA
```

```
The task (food-shopping 0 2 2768 6 (only-choice)) is ready to be scheduled.

                    ---Creating New Environment---:
     The environment at time 232 240 has position data
     -------------Start STATE OBJECT-------------
     The speed is in state: 30
     The robot-loc is in state: garage
     The possessions is in state: (laundry car)
     The car-loc is in state: 'robot
     -------------End of STATE OBJECT-------------
The status of the Schedule Dependency Array is:
     task                          status         countdown
     drop-off-car                  finished       t
     pick-up-car                   finished       t
     drop-off-laundry              finished       t
     transfer-laundry              finished       t
     pick-up-laundry               finished       t
     food-shopping                 schedulable    nil
     go-home                       waiting        nil
     mail-letters                  finished       t
End status SDA
     The schedule so far is:
          Move robot          (move robot from location home to laundr-o-mat at 30 mph
                                             (then drop off laundry))
          Reasons: (best-dd)
        drop-off-laundry                61 61  0
          Move robot          (move robot from location laundr-o-mat to po at 30 mph
                                             (the destination))
          Reasons: (best-dd closest-dl)
        mail-letters                    67 70  0
          Move robot          (move robot from location po to laundr-o-mat at 30 mph
                                             (the destination))
          Reasons: (best-dd min-state-change closest-dl)
        transfer-laundry                106 114  0
          Move robot          (move robot from location laundr-o-mat to garage at 30 mph
                                             (then drop off car))
          Reasons: (best-dd)
        drop-off-car                    111 119  0
          Move robot          (move robot from location garage to laundr-o-mat at 5 mph
                                             (then pickup laundry))
          Reasons: (best-dd closest-dl)
        pick-up-laundry                 167 175  0
          Move robot          (move robot from location laundr-o-mat to garage at 3 mph
                                             (then pickup car))
        pick-up-car                     232 240  0

I30I

------------Weights------------
The task weight factor is: 8
The approaching deadline factor is: 25
The RTT factor is: 11
The domain heuristics factor is: 13
The zero state delay factor is: 100
The waste time factor is: 8
----------End--Weights---------
The status of the Schedule Dependency Array is:
```

```
     task                         status        countdown
     drop-off-car                 finished      t
     pick-up-car                  finished      t
     drop-off-laundry             finished      t
     transfer-laundry             finished      t
     pick-up-laundry              finished      t
     food-shopping                finished      t
     go-home                      schedulable   nil
     mail-letters                 finished      t
End status SDA
!!!Impending deadline violation on task go-home
P-delay 0
S-delay 62
Deadline 36

*******************************************************************
   One last time a deadline is busted. The weights are now seriously
   shifted towards taking tasks in a deadline come first order.
   After this correction things go smoothly, and the weights start
   to decay towards there initial values.
*******************************************************************

$$$$$$$$$Deadline Violation$$$$$$$
Task go-home has busted its deadline.
$$$$$$End Deadline Violation$$$$$$
                    !!!!Failed Environment!!!!
                    The following prefix deadends
     The schedule so far is:
          Move robot          (move robot from location home to laundr-o-mat at 30 mph
                                              (then drop off laundry))
          Reasons: (best-dd)
        drop-off-laundry                61 61  0
          Move robot          (move robot from location laundr-o-mat to po at 30 mph
                                              (the destination))
          Reasons: (best-dd closest-dl)
        mail-letters                    67 70  0
          Move robot          (move robot from location po to laundr-o-mat at 30 mph
                                              (the destination))
          Reasons: (best-dd min-state-change closest-dl)
        transfer-laundry                106 114  0
          Move robot          (move robot from location laundr-o-mat to garage at 30 mph
                                              (then drop off car))
          Reasons: (best-dd)
        drop-off-car                    111 119  0
          Move robot          (move robot from location garage to laundr-o-mat at 5 mph
                                              (then pickup laundry))
          Reasons: (best-dd closest-dl)
        pick-up-laundry                 167 175  0
          Move robot          (move robot from location laundr-o-mat to garage at 3 mph
                                              (then pickup car))
        pick-up-car                     232 240  0
          Move robot          (move robot from location garage to grocery at 30 mph
                                              (then pickup food))
          Reasons: (only-choice)
        food-shopping                   254 272  0

                    !!!!End Failed Environment!!!!H
```

```
I31I

------------Weights------------
The task weight factor is: 8
The approaching deadline factor is: 26
The RTT factor is: 11
The domain heuristics factor is: 13
The zero state delay factor is: 100
The waste time factor is: 8
----------End--Weights---------
The status of the Schedule Dependency Array is:
     task                          status          countdown
     drop-off-car                  finished        t
     pick-up-car                   schedulable     nil
     drop-off-laundry              finished        t
     transfer-laundry              finished        t
     pick-up-laundry               schedulable     nil
     food-shopping                 finished        t
     go-home                       waiting         nil
     mail-letters                  schedulable     nil
End status SDA
The tasks (mail-letters 0 40 2877 140 nil
           pick-up-car 63 40 93 140 (best-dd min-state-change closest-dl)
           pick-up-laundry 60 0 90 120 (lowest-rtt best-dd min-state-change closest-dl))
    are ready to be scheduled.

                ---Creating New Environment---:
     The environment at time 123 138 has position data
     -------------Start STATE OBJECT-------------
     The speed is in state: 3
     The car-loc is in state: 'garage
     The car-loc restriction:
         car-unavailable
     is active and will be terminated by the task: pick-up-car.

     The robot-loc is in state: laundr-o-mat
     The possessions is in state: (food)
     -------------End of STATE OBJECT-------------
The status of the Schedule Dependency Array is:
     task                          status          countdown
     drop-off-car                  finished        t
     pick-up-car                   schedulable     nil
     drop-off-laundry              finished        t
     transfer-laundry              finished        t
     pick-up-laundry               schedulable     nil
     food-shopping                 finished        t
     go-home                       waiting         nil
     mail-letters                  schedulable     nil
End status SDA
     The schedule so far is:
          Move robot          (move robot from location home to laundr-o-mat at 30 mph
                                  (then drop off laundry))
          Reasons: (best-dd)
        drop-off-laundry              61 61  0
          Move robot          (move robot from location laundr-o-mat to garage at 30 mph
                                  (then drop off car))
        drop-off-car                  66 66  0
```

```
        Move robot          (move robot from location garage to grocery at 5 mph
                                           (then pickup food))
        Reasons: (min-state-change)
      food-shopping                  98 108  0
        Move robot          (move robot from location grocery to laundr-o-mat at 3 mph
                                           (the destination))
        Reasons: (best-dd min-state-change closest-dl)
      transfer-laundry               123 138  0

I32I

------------Weights------------
The task weight factor is: 8
The approaching deadline factor is: 26
The RTT factor is: 11
The domain heuristics factor is: 13
The zero state delay factor is: 100
The waste time factor is: 8
----------End--Weights---------
The status of the Schedule Dependency Array is:
     task                         status        countdown
     drop-off-car                 finished      t
     pick-up-car                  finished      t
     drop-off-laundry             finished      t
     transfer-laundry             finished      t
     pick-up-laundry              schedulable   nil
     food-shopping                finished      t
     go-home                      waiting       nil
     mail-letters                 schedulable   nil
End status SDA
The tasks (mail-letters 0 4 2313 124 (lowest-rtt)
          pick-up-laundry 0 4 26 124 nil)
    are ready to be scheduled.

                    ---Creating New Environment---:
     The environment at time 187 202 has position data
     -------------Start STATE OBJECT-------------
     The speed is in state: 30
     The robot-loc is in state: garage
     The possessions is in state: (food car)
     The car-loc is in state: 'robot
     -------------End of STATE OBJECT-------------
The status of the Schedule Dependency Array is:
     task                         status        countdown
     drop-off-car                 finished      t
     pick-up-car                  finished      t
     drop-off-laundry             finished      t
     transfer-laundry             finished      t
     pick-up-laundry              schedulable   nil
     food-shopping                finished      t
     go-home                      waiting       nil
     mail-letters                 schedulable   nil
End status SDA
     The schedule so far is:
          Move robot          (move robot from location home to laundr-o-mat at 30 mph
                                           (then drop off laundry))
          Reasons: (best-dd)
```

```
    drop-off-laundry                61 61  0
      Move robot           (move robot from location laundr-o-mat to garage at 30 mph
                                     (then drop off car))
    drop-off-car                    66 66  0
      Move robot           (move robot from location garage to grocery at 5 mph
                                     (then pickup food))
      Reasons: (min-state-change)
    food-shopping                   98 108  0
      Move robot           (move robot from location grocery to laundr-o-mat at 3 mph
                                     (the destination))
      Reasons: (best-dd min-state-change closest-dl)
    transfer-laundry                123 138  0
      Move robot           (move robot from location laundr-o-mat to garage at 3 mph
                                     (then pickup car))
      Reasons: (best-dd min-state-change closest-dl)
    pick-up-car                     187 202  0

I33I

------------Weights------------
The task weight factor is: 8
The approaching deadline factor is: 25
The RTT factor is: 11
The domain heuristics factor is: 13
The zero state delay factor is: 100
The waste time factor is: 8
----------End--Weights---------
The status of the Schedule Dependency Array is:
     task                          status         countdown
     drop-off-car                  finished       t
     pick-up-car                   finished       t
     drop-off-laundry              finished       t
     transfer-laundry              finished       t
     pick-up-laundry               finished       t
     food-shopping                 finished       t
     go-home                       waiting        nil
     mail-letters                  schedulable    nil
End status SDA
The task (mail-letters 0 4 2808 8 (only-choice)) is ready to be scheduled.

                   ---Creating New Environment---:
     The environment at time 192 207 has position data
     -------------Start STATE OBJECT-------------
     The car-loc is in state: 'robot
     The speed is in state: 30
     The robot-loc is in state: laundr-o-mat
     The possessions is in state: (car food laundry)
     -------------End of STATE OBJECT-------------
The status of the Schedule Dependency Array is:
     task                          status         countdown
     drop-off-car                  finished       t
     pick-up-car                   finished       t
     drop-off-laundry              finished       t
     transfer-laundry              finished       t
     pick-up-laundry               finished       t
     food-shopping                 finished       t
     go-home                       waiting        nil
```

```
    mail-letters                  schedulable    nil
End status SDA
    The schedule so far is:
        Move robot           (move robot from location home to laundr-o-mat at 30 mph
                                         (then drop off laundry))
        Reasons: (best-dd)
      drop-off-laundry               61 61  0
        Move robot           (move robot from location laundr-o-mat to garage at 30 mph
                                         (then drop off car))
      drop-off-car                   66 66  0
        Move robot           (move robot from locatior garage to grocery at 5 mph
                                         (then pickup food))
        Reasons: (min-state-change)
      food-shopping                  98 108  0
        Move robot           (move robot from location grocery to laundr-o-mat at 3 mph
                                         (the destination))
        Reasons: (best-dd min-state-change closest-dl)
      transfer-laundry               123 138  0
        Move robot           (move robot from location laundr-o-mat to garage at 3 mph
                                         (then pickup car))
        Reasons: (best-dd min-state-change closest-dl)
      pick-up-car                    187 202  0
        Move robot           (move robot from location garage to laundr-o-mat at 30 mph
                                         (then pickup laundry))
      pick-up-laundry                192 207  0

I34I

------------Weights------------
The task weight factor is: 8
The approaching deadline factor is: 25
The RTT factor is: 11
The domain heuristics factor is: 13
The zero state delay factor is: 100
The waste time factor is: 8
----------End--Weights---------
The status of the Schedule Dependency Array is:
    task                          status         countdown
    drop-off-car                  finished       t
    pick-up-car                   finished       t
    drop-off-laundry              finished       t
    transfer-laundry              finished       t
    pick-up-laundry               finished       t
    food-shopping                 finished       t
    go-home                       schedulable    nil
    mail-letters                  finished       t
End status SDA
The task (go-home 0 56 92 7 (only-choice)) is ready to be scheduled.

                   ---Creating New Environment---:
    The environment at time 198 216 has position data
    -------------Start STATE OBJECT-------------
    The speed is in state: 30
    The car-loc is in state: 'robot
    The robot-loc is in state: po
    The possessions is in state: (laundry food car)
    -------------End of STATE OBJECT-------------
```

```
The status of the Schedule Dependency Array is:
     task                          status        countdown
     drop-off-car                  finished      t
     pick-up-car                   finished      t
     drop-off-laundry              finished      t
     transfer-laundry              finished      t
     pick-up-laundry               finished      t
     food-shopping                 finished      t
     go-home                       schedulable   nil
     mail-letters                  finished      t
End status SDA
     The schedule so far is:
          Move robot          (move robot from location home to laundr-o-mat at 30 mph
                                             (then drop off laundry))
          Reasons: (best-dd)
        drop-off-laundry               61 61  0
          Move robot          (move robot from location laundr-o-mat to garage at 30 mph
                                             (then drop off car))
        drop-off-car                   66 66  0
          Move robot          (move robot from location garage to grocery at 5 mph
                                             (then pickup food))
          Reasons: (min-state-change)
        food-shopping                  98 108  0
          Move robot          (move robot from location grocery to laundr-o-mat at 3 mph
                                             (the destination))
          Reasons: (best-dd min-state-change closest-dl)
        transfer-laundry               123 138  0
          Move robot          (move robot from location laundr-o-mat to garage at 3 mph
                                             (then pickup car))
          Reasons: (best-dd min-state-change closest-dl)
        pick-up-car                    187 202  0
          Move robot          (move robot from location garage to laundr-o-mat at 30 mph
                                             (then pickup laundry))
        pick-up-laundry                192 207  0
          Move robot          (move robot from location laundr-o-mat to po at 30 mph
                                             (the destination))
          Reasons: (only-choice)
        mail-letters                   198 216  0

I35I

------------Weights------------
The task weight factor is: 8
The approaching deadline factor is: 24
The RTT factor is: 11
The domain heuristics factor is: 13
The zero state delay factor is: 100
The waste time factor is: 8
----------End--Weights---------

*********************************************************************
*********************************************************************
******************************The Answer*****************************
*********************************************************************
*********************************************************************
```

Note that this time the schedule has the safe times for executing each task filled in (along with the high and low times).

```
**************************************************************************

  The schedule so far is:
     Move robot           (move robot from location home to laundr-o-mat at 30 mph
                                          (then drop off laundry))
     Reasons: (best-dd)
   drop-off-laundry                 61 61  69
     Move robot           (move robot from location laundr-o-mat to garage at 30 mph
                                          (then drop off car))
   drop-off-car                     66 66  74
     Move robot           (move robot from location garage to grocery at 5 mph
                                          (then pickup food))
     Reasons: (min-state-change)
   food-shopping                    93 108  106
     Move robot           (move robot from location grocery to laundr-o-mat at 3 mph
                                          (the destination))
     Reasons: (best-dd min-state-change closest-dl)
   transfer-laundry                 123 138  131
     Move robot           (move robot from location laundr-o-mat to garage at 3 mph
                                          (then pickup car))
     Reasons: (best-dd min-state-change closest-dl)
   pick-up-car                      187 202  208
     Move robot           (move robot from location garage to laundr-o-mat at 30 mph
                                          (then pickup laundry))
   pick-up-laundry                  192 207  213
     Move robot           (move robot from location laundr-o-mat to po at 30 mph
                                          (the destination))
     Reasons: (only-choice)
   mail-letters                     198 216  234
     Move robot           (move robot from location po to home at 30 mph
                                          (the destination))
     Reasons: (only-choice)
   go-home                          254 272  290
->
```