Meta-Level Control for Decision-Theoretic Planners

Richard Goodwin

September 16, 1998

CMU-CS-96-186

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

Thesis Committee: R. Simmons, Chair
T. Mitchell
H. Simon
M. Wellman, U. Michigan

This research was supported by the Jet Propulsion Laboratory under contract number 960332, and in part by NASA grant NAGW-1175. R. Goodwin was also recipient of a Natural Science and Engineering Research Council of Canada Scholarship (NSERC). The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of JPL, NASA, NSERC, or the U.S. government.

Keywords: Decision-Theoretic Planning, Bounded Rationality, Sensitivity Analysis, Meta-Level Control

Abstract

Agents plan in order to improve their performance, but planning takes time and other resources that can degrade performance. To plan effectively, an agent needs to be able to create high quality plans efficiently. Artificial intelligence planning techniques provide methods for generating plans, whereas decision theory offers expected utility as a measure for assessing plan quality, taking the value of each outcome and its likelihood into account. The benefits of combining artificial intelligence planning techniques and decision theory have long been recognized. However, these benefits will remain unrealized if the resulting decision-theoretic planners cannot generate plans with high expected utility in a timely fashion. In this dissertation, we address the meta-level control problem of allocating computation to make decision-theoretic planning efficient and effective.

For efficiency, decision-theoretic planners iteratively approximate the complete solution to a decision problem: planners generate partially elaborated, abstract plans; only promising plans are further refined, and execution may begin before a plan with the highest expected utility is found. Our work addresses three key meta-level control questions related to the planning process: whether to generate more plans or refine an existing partial plan, which part of a partial plan to refine, and when to commence execution. We show that an optimistic strategy that refines the plan with the highest bounds on expected utility first uses minimal computation when looking for a plan with the highest expected utility. When looking for a satisficing solution, we weigh the opportunity cost of forgoing more planning against the computational cost to decide whether to generating more plans. When selecting which part of a plan to refine, we use sensitivity analysis to identify refinements that can quickly distinguish plans with high expected utility. For deciding when to begin execution, previous methods have ignored the possibility of overlapping planning and execution. By taking this possibility into account, our method can improve performance by accomplishing a task more quickly. To validate our theoretical results, our methods have been applied to four decision-theoretic planners used in domains such as mobile robot route planning and medical treatment planning. Empirical tests against competing meta-level control methods show the effectiveness of our approach.

Acknowledgements

I have had the good fortune of enjoying the generous support and assistance of many friends and colleagues in the School of Computer Science at Carnegie Mellon University. Foremost among these is my mentor, Reid Simmons, who gave me the freedom to pursue my ideas and interests while teaching me the fundamentals of good scientific research. I thank him for his indulgence and guidance. I have also benefited greatly from my interactions with Lonnie Chrisman and Sven Koenig. They introduced me to decision-theoretic planning and meta-level reasoning. I thank them, together with Reid, for many enlightening discussions on a diverse set of topics in our weekly discussion group. In addition, I am in debt to everyone in Xavier Robot Group, especially Joseph O'Sullivan and Greg Armstrong. Robotics is a team sport and I enjoyed the trials, tribulations and triumphs of a combined effort to make a real machine work. Finally, I wish to thank my officemates, past and present, Jim Blythe, Zoran Popovic, Dario Salvucci, Prem Janardhan, Paul Allan, Dhiraj Pathak, Milind Tambe and Dave Plaut, for lively debates and the occasional volley of dinosaurs to keep me on my toes. But there should be much more to the life of a graduate student than reading and research and I thank my friends for distracting me from a seemingly endless task and keeping my spirits high (morp, morp, morp).

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Planning is the process of creating a policy for acting that an agent can use to increase the likelihood of achieving its objectives. Classical artificial intelligence planning methods formulated an agent's objectives as a set of first order predicate logic clauses. A plan, in this framework, is a sequence of operators, possibly with branches and loops, that takes an agent from an initial state to a goal state. Artificial intelligence planning research has concentrated on how to represent plans, actions and states, and how to efficiently create plans given these representations. An artificial intelligence planning agent is rational if it takes an action whenever it believes the action would increase the likelihood that the agent would achieve its goals [Newell, 1982].

Decision theory, developed in the context of economics and psychology, recognizes that the agent's objectives may not be captured by a single set of goal states. Instead, a utility function is defined over the set of possible outcomes that indicate the relative desirability of each outcome. An agent is rational if it acts to maximize its expected utility. Using decision theory allows an agent to take into account its relative preference for outcomes as well as their likelihood. It does not address the problem of how plans should be generated as artificial intelligence planning does [Simon, 1988].

A problem with both classical artificial intelligence planning and decision theory is that they ignore the cost of computation. An agent is not performing rationally if by the time it calculates the best action, that action is not longer applicable. Taking the cost of computation into account leads to what Simon calls *procedural rationality* [Simon, 1976] and what Good refers to as *type II rationality* [Good, 1971]. An agent exhibits such bounded rationality if it maximizes its expected utility given its computational and other resource limits. The problem of how to allocate resources efficiently, including computation, to produce a bounded rational agent is the *meta-level control problem*.

An agent is rational if it accomplishes its task efficiently, given its resources. Any real agent, whether it is a robot, a softbot or a human, will have limited resources, including computation. Our interest in creating useful agents leads us to explore meta-level control for resource-bounded rational agents.

1.1 The Problem

Decision-theoretic planners combine artificial intelligence planning techniques with decision theory to produce plans with high expected utility. The object level problem of selecting actions for execution is modeled as a decision problem. Solving a decision problem consists of four steps: formulating plans, estimating parameters, such as probabilities and costs, evaluating each plan and executing the plan with the highest expected utility for execution. The meta-level control problem is to judiciously allocate computation to each step of the process.

The first step in solving a decision problem is to enumerate the possible plans. Enumerating all the possible plans will take too long for all but the most trivial of problems. An efficient plan generator would ideally generate only the plan with the highest expected utility, but this is not generally possible. What can be done is to create a plan generator that iteratively produces plans and, hopefully, tends to produce plans with higher expected utility before plans with lower expected utility. A plan generator may also only partially elaborate the plans it creates by for example using abstract actions and not planning for all possible contingencies. Using abstraction and not planning of all possible contingencies can improve efficiency by reducing the amount of work wasted on low utility plans. The relevant meta-level control decisions are: when to generate more plans, when to refine partial plans and when to discard unpromising plans.

In order to evaluate the expected utility of a plan, estimates of parameters such as the durations of actions, resource use, and probabilities are required. In decision-theoretic planners, action models used to generate plans are augmented to include these estimates. Probabilities can be estimated from prior experience with the environment or through simulation techniques like temporal projection that use models of the environment [Hanks, 1990]. As with plan generation, greater efficiency is possible if the planner initially generates approximate estimates. Only those estimates deemed critical need to be refined. The problem for the meta-level controller is to decide how good parameter estimates need to be and when to refine them.

With complete plans and parameter estimates, evaluating a plan is a straightforward application of the utility function. A utility function maps outcomes to a real valued number indicating the relative desirability of each outcome. The valuation of each outcome takes into account tradeoffs between resource use and task achievement and preferences about risk. A plan is evaluated by applying the utility function to each possible outcome and summing the results, weighted by their likelihood, to give the expected utility. With partial plans and partial parameter estimates, it is not possible to calculate an expected utility. A planner may be able to calculate only a range of values or a most likely value of expected utility.

The final step in a solving a decision problem is to select the plan with the highest expected utility. Again, with a complete solution, this step is straightforward. The plan with the highest expected utility is selected for execution. When only some plans have been generated and when partial plans give ranges of expected utility, the decision is more difficult. The meta-level controller must decide whether the current best plan should be acted upon or whether more computation should be done first. More computation can lead to a better plan that can help performance, but delays the start of execution, that can hurt performance.

1.2 Motivating Example

To illustrate the issues involved in meta-level control for decision-theoretic planners, consider the example of a robot safety warden shown in figure 1.1. The safety robot's task is to patrol the halls of a building containing laboratories, using its sensors to monitor for hazardous chemical spills. When a spill is detected, the robot raises an alarm and then checks each lab to make sure the occupants have gotten out safely. The robot carries gas masks that it can pass out to anyone it finds in the building and it can use its radio to direct human rescuers to victims it cannot help. The safety robot's objective is to check all the labs as quickly as possible after it detects a spill.

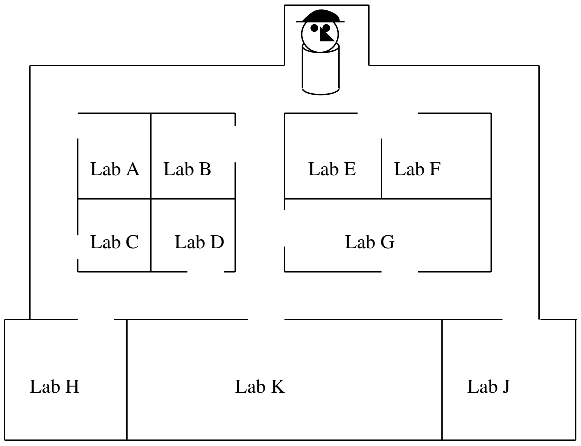

Figure 1.1: Robot Safety Warden.

The planning problem for the safety robot consists of ordering labs to visit and planning routes between labs. Any random ordering of the labs produces a valid tour of the labs, but the objective is to find a tour that can be followed quickly. The robot cannot move through walls, so to go from one lab to another, the robot has to plan a route along the hallways. Again, there are multiple routes between labs, but some routes are shorter than others. The robot could find the shortest tour by generating all possible tours and planning the routes between labs for each tour. The problem is that this approach can be prohibitively time consuming. Suppose that there are 10 labs and the time needed to plan a route is 100 microseconds. Since there are $10! = 3,628,800$ possible tours and each one takes 100 $usec$ to evaluate, the time needed to do complete planning is 6 minutes. Obviously, delaying the start of execution for 6 minutes to find the shortest tour is not the best strategy. On the other hand, a greedy, reactive strategy of always going to the closest remaining room may waste valuable time by unnecessarily traversing long hallways multiple times.

In between these two extremes, the robot could plan until it had a satisfactory solution and then begin to execute it while continuing to improve its plan.

In producing a satisfactory solution, the planner does not need to generate the full set of tours or plan all the routes between labs for the tours it does generate. Instead, the planner can interleave tour generation with route planning. When a tour is generated, the planner can get a lower bound on its length by summing the Euclidean distance between labs. The length of a route between labs can be longer, because of walls or other obstacles, but can never be shorter than the straight line distance. A possible upper bound on route length is the sum of the lengths of all the corridors, since any reasonable route would not travel the same section of a corridor more than once[1]. Using this observation gives us an upper bound on tour length. Planning a route between two labs in a tour reduces the range of distances for the route to a point-value and correspondingly reduces the range for the tour. Planning all the routes in a tour reduces the range of distances for the tour to a point-value. The meta-level control problem for the safety robot is to decide when to generate more tours, when to plan a route and when to begin execution.

The plan generation problem is a choice between generating another tour and planning a route in a partially elaborated tour. Generating a new tour explores more of the search space, possibly finding a much better tour. Refining an existing tour, by planning a route between labs, helps the planner distinguish short tours from long tours. The planner must also plan the route to the first lab in a tour before it can begin executing the tour.

Given that the meta-level controller has selected a tour to work on, the refinement guiding problem is to select the pair of labs to plan a route between. Planning some routes will have a larger affect on the range of lengths for a tour than others. Planning routes that have a larger affect on the range first helps to distinguish shorter tours from longer tours with less work. The effort of the planner should be focused on refining parts of the plan to which its expected utility is most sensitive.

Finally, the meta-level controller needs to decide when to begin execution. Continuing to plan can reduce the length of the tour, which shortens the time needed to execute it, but delays the start of execution. In making this tradeoff, the controller needs to take into account whether the robot can overlap execution of the first part of the tour while continuing to improve the rest of the tour.

The three meta-level control problems of plan generation, refinement guiding and commencing execution form the central topic of this dissertation. In the rest of this chapter, we describe the meta-level control problem for decision-theoretic planners and outline our approach to providing effective and efficient meta-level control.

1.3 Approach

Our ultimate goal is to create agents that can perform useful tasks. Since planning can help an agent improve its performance and decision theory is useful for evaluating the quality of

[1] Assuming the robot can open all doors and that all corridors are passable.

a plan, we are naturally led to explore decision-theoretic planning techniques. Our interest in creating real agents, which will necessarily have limited computation, also necessitates a concern for allocating computation efficiently.

In this section we outline our approach to providing effective meta-level control for resource-bounded agents. We begin by describing the conceptual organization of a resource-bounded agent that uses decision-theoretic planning and on-line meta-level control to allocate computation. Since decision-theoretic planners iteratively approximate a full solution to a decision problem, we present the decision-problem framework. We use this framework to identify the relevant computations and the associated meta-level decisions. For each decision, we describe how to make the decision given perfect information. Since perfect information is generally unavailable, or too expensive to compute, we describe how to estimate the required information using models of the planner and sensitivity analysis. We conclude by describing a range of decision-theoretic planners and domains that we will use in examples to illustrate meta-level control issues and to demonstrate how our approach can improve their performance.

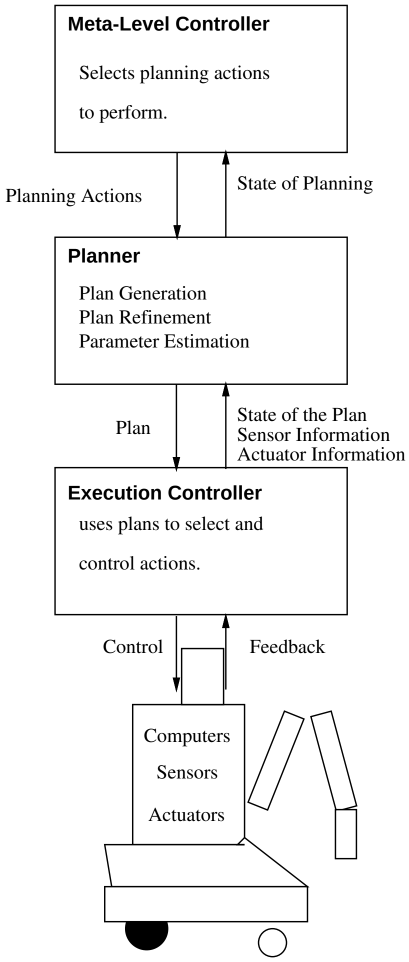

Figure 1.2: Conceptual Organization of a bounded rational agent.

A conceptual organization of a resource-bounded agent that uses planning and on-line meta-level control is illustrated in figure 1.2. At the lowest level is the interface between the agent and its environment, which provides the agent with its means of sensing and affecting the environment. The actions of the agent are controlled by an execution controller that is in turn guided by plans provided by the planner. The planner gets information about the environment and the current state of plan execution from the controller and uses this information to generate and refine its plans. The meta-level controller gets information from the planner about the current state of the planning process and uses this information to direct the actions of the planner. In this work, we focus on the organization and content of the meta-level controller and discuss the other components only to the extent required to understand meta-level control issues.

The basis of our approach to meta-level control is to model the decision-theoretic planning process as an iterative approximation to a full solution of a decision problem. Since decision theory underlies decision-theoretic planning, the advantage of this approach is that it allows us to identify the meta-level control decisions relevant to decision-theoretic planners in general. The steps in a decision problem are to generate plans, estimate parameters, such as probabilities, evaluate each plan using a utility function, and to execute the plan with the highest expected utility. The meta-level control problem is to judiciously allocate computation to each step in the process. Three key decisions that we focus on are whether to generate another plan or refine a partial plan, which part of a partial plan to refine and when to begin execution.

We consider how to make each meta-level decision in two contexts: when looking for a plan with the highest expected utility using the least amount of computation and when looking for a satisficing plan that produces the best performance taking resource limits into account. The two contexts are related. If a plan with the highest expected utility can be found quickly enough, then finding it and using it produces the best performance even with limited resources. The reason for considering how to find a plan with the highest expected utility efficiently is that it is often easier to analyze the meta-level decisions in this context. The resulting methods for making meta-level decisions can then be adapted when looking for satisficing solutions.

For each meta-level decision, we attempt to find an algorithm for making the correct decision assuming that we have perfect information and unlimited resources for meta-level reasoning. Perfect information in this case includes the duration and results of the calculations a planner could perform. The resulting idealized algorithms are not operational, but illustrate the factors that should be taken into account when making a decision and can serve as guides for implementing practical algorithms.

In order to make our meta-level control algorithms operational, we need to estimate the required information and approximate the full algorithm. To estimate the performance of continuous processes, we use an anytime algorithm approach and create performance profiles [Boddy and Dean, 1989]. These profiles show the expected result of a computation versus processing time. For discrete computations, we use models of the planner's performance to estimate duration and sensitivity analysis to estimate their effects. We approximate computationally expensive meta-level control algorithms using simplified greedy strategies.

While we examine three distinct meta-level control decisions in this dissertation, we take a common approach to making each decision. First, we formulate an idealized algorithm that could make the correct decision, given perfect information. We then create practical meta-level controllers by estimating the required information and approximating any computationally expensive algorithms. We use empirical tests to compare our approach to competing approaches. Our aim is to validate the following thesis.

Thesis Statement: Approximating the ideal solution to meta-level decisions for decision-theoretic planners, using sensitivity analysis and a model of the planner to estimate required information, provides effective allocation of computation.

We now give an overview of how our approach is applied for each of the three meta-level control decisions we address. The overview illustrates the key issues for each decision and outlines the contributions of this dissertation. We also include a summary of some of our empirical results that we use to justify the effectiveness of our approach.

Plan Generation: For efficiency, a decision-theoretic planner may only partially elaborate the plans it generates and then refine only promising plans. The idea is to waste as little computation as possible on plans with low expected utility. The meta-level controller must choose between generating more plans and refining one of the partially elaborated plans. To make this choice, we calculate bounds on the expected utility of partial plans and on the set of plans yet to be generated. Partial plans have ranges of expected utility that reflect the possible ways in which they could be elaborated. If plans are generated systematically,from shortest to longest for instance, then it may be possible to characterize the plans remaining to be generated and bound their expected utility. We will show that, when looking for a plan with the highest expected utility, the highest upper bound on expected utility is critical when using ranges to expected utility to evaluate partial plans. If a partial plan has the highest upper bound on expected utility, then the planner should refine it. If the highest upper bound on expected utility is for the set of plans yet to be generated, then another plan should be generated. This strategy leads to a plan with the highest expected utility with the best possible guarantee on the amount of computation required. In practice, this strategy can produce an order of magnitude in performance over previously used strategies.

Refinement Guiding: Once a partial plan has been selected for refinement, the meta-level controller must decide which part of the plan to refine. The objective is to select refinements that have a large effect on the upper bound on expected utility for a plan, since such refinements will quickly distinguish plans with low expected utility from plans with high expected utility. However, we do not know the effect that a computation will have on the expected utility bounds until we do the computation. To approximate this strategy, we use a sensitivity analysis to identify parts of the plan to which the upper bound on expected utility is most sensitive to. The meta-level controller also has to take into account the amount of computation that each refinement requires. We use a model of the planner to estimate the amount of computation that each refinement will take. The meta-level controller than greedily selects refinements that have the highest ratio of sensitivity to expected work. This is a general heuristic that adapts meta-level control to the particulars of a given problem. In practice, it produces results that are comparable to (and is some cases better than) hand-tuned strategies without the need for hand tuning.

Commencing Execution: Planning generally has a diminishing rate of return and an agent may improve its performance by beginning execution before a plan with the highest expected utility is found. The tradeoff is between finding an improved plan and delaying execution. Previous approaches to this problem have cast it as a choice between planning and execution. However, this ignores the possibility of continuing to improve a plan after execution has begun. We show how to take the possibility of overlapping planning and execution into account and demonstrate how this can improve performance by beginning execution earlier and completing the task sooner.

Planners: To demonstrate our approach to meta-level control and to illustrate the practical issues involved in creating meta-level controllers for real planners, we have applied our approach to four decision-theoretic planners: the Xavier route planner is a domain specific planner used to route a mobile robot around an office building; The DRIPS planner is a hierarchical refinement planner used to create medical treatment policies; The robot-courier tour planner is another domain specific planner that iteratively improves the ordering of a set of deliveries for a courier robot; The Pyrrhus planner is partial order planner that has been used in logistics domains with deadlines and resource constraints. We have implemented our techniques in the first three planners and have analyzed the fourth planner, Pyrrhus, to show how our meta-level control techniques apply. This set of four planners, which span the range of decision-theoretic planners created to date, serve as a guide for implementing our techniques in other planners. Our implementations also allows us to compare our approach empirically against other meta-level control techniques including domain specific heuristics and hand tuned meta-level control schemes. We use these empirical results to validate our approach.

1.4 Outline

We begin our examination of meta-level control for decision-theoretic planners by reviewing the work done in this area to date. We then characterize the decision-theoretic planning process and create a taxonomy of meta-level control decisions. This taxonomy of decisions provides the framework for identifying meta-level decisions in particular planners and gives the relevant factors for making each decision. Since we use sensitivity analysis to make some of our meta-level decisions, we provide a review of sensitivity analysis and its applicability to planning in Chapter 4. We then introduce the four decision-theoretic planners in Chapter 5 that we use to provide examples. These planners also serve as implementations of our techniques that we use for our empirical tests in the subsequent chapters. With the ground work established, we delve into the three core meta-level control decisions. Plan generation, refinement guiding and commencing execution are covered in chapters 6, 7, and 8 respectively. We conclude with a summary of our contributions to meta-level control for decision-theoretic planners and describe some problems and issues that remain for future work.

Chapter 2

Related Work

This dissertation develops meta-level control algorithms for effectively allocating computation for decision-theoretic planners in an effort to create resource-bounded rational agents. We build on ideas and concepts developed in the fields of bounded rationality, decision-theory and artificial intelligence in developing our meta-level control strategies. The idea of bounded rationality recognizes that any real agent, including a person or a machine, will have limited computational resources and needs to take these limitations into account in order to perform well. One method for taking computational limits into account is to explicitly allocate computation using on-line meta-level control. Work in the area of artificial intelligence has developed meta-level control algorithms for classical planners which represent objectives as a set of goals to be achieved. With the combination of decision-theory and artificial intelligence planning, there is a richer language for expressing objectives and a greater opportunity for on-line meta-level control algorithms to allocate computation in a way that helps to achieve those objectives. In creating meta-level control algorithms for decision-theoretic planners, we build on work done for meta-level control in classical planners. We also make use of techniques from sensitivity analysis and work on anytime algorithms to extend meta-level control to decision-theoretic planners.

This chapter reviews some of the related work upon which this dissertation is based. We begin by examining the development of ideas related to bounded rationality. We then look at work on meta-level control as it has been applied to classic artificial intelligence planners and, more recently, to decision-theoretic planners. Following this general review, we look at work related to each of the three meta-level control decisions that we address in this dissertation. Meta-level control for plan generation depends on the method used to generate plans and the method for determining when one plan dominates another. We examine work on efficiently creating plans with high expected utility and on methods for showing dominance. We also discuss competing meta-level control strategies that have been suggested in the literature and relate them to our approach. Refinement guiding is equivalent to the problem of flaw selection in classical planners. We describe the work that has been done on flaw selection for classical planners and on attempts to apply this work to decision-theoretic planners. We contrast this with our approach that uses sensitivity analysis to identify refinements that help to distinguish high expected utility plans. Finally, we review the work on when to begin execution.

2.1 Bounded Rationality

The basic idea of bounded rationality arises in the work of Simon with his definition of procedural rationality [Simon, 1976]. Simon's work has addressed the implications of bounded rationality in the areas of psychology, economics and artificial intelligence [Simon, 1982]. He argues that people find satisfactory solutions to problems rather than optimal solutions, because people do not have unlimited processing power. In the area of agent design, he has considered how the nature of the environment can determine how simple an agent's control algorithm can be and still produce rational behaviour [Simon, 1956].

In the area of problem solving, Simon and Kadane propose that search algorithms, for finding solutions to problems given in terms of goals, are making a tradeoff between computation and solution quality. A solution that satisfies the goals of a problem is a minimally acceptable solution. Finding such a solution quickly best uses computation [Simon and Kadane, 1974]. In using a decision-theoretic framework, we make the tradeoff between computation and solution quality explicit at run time On-line meta-level control allows us to take into account the characteristics of a particular problem rather than just the average case performance.

Good's type II rationality is closely related to Simon's ideas on bounded rationality [Good, 1971]. Type II rationality, which is rationality that takes into account resources limits, is a concept that has its roots in mathematics and philosophy rather than psychology. Good creates a set of normative principles for rational behaviour that take computational limits into account. He also considers explicit meta-level control and how to make decisions given perfect information about the duration and value of each possible computation. We use Good's approach for each of the three meta-level decisions we consider. First, we create an idealized algorithms that makes each decision correctly, given perfect information. Moving beyond Good, we then approximate the idealized algorithm to produce practical implementations. In deciding when to commence execution, we identify an assumption in Good's idealized algorithm that ignores the possibility of overlapping planning and execution. We remove this assumption and show how our revised algorithm can lead to improved performance.

Russell, Subramanian and Parr cast the problem of creating resource-bounded rational agents as a search for the best program that an agent can execute [Russell *et al.*, 1993]. This definition of rationality does not depend on the method used to create a program or the method it uses to do computation but only on the behaviours that result from running the program. The approach we take here is a constructive one that Russell calls meta-level rationality. By approximating the correct meta-level decisions, we create agents that produce high expected utility, given resource limits. However, we can make no guarantees about the optimality of the agents we create. In searching the space of programs, Russell can sometimes argue that his agents are optimal for a given class of programs or that his agents approach optimal performance with learning, again given a limited class of possible programs.

2.1.1 Meta-Level Control

On-line meta-level control uses computation to explicitly decide which object level computations to perform. The central questions are the types of decisions to be made and the algorithm used for making each decision. For planning, the decisions arise from the choice points in non-deterministic planning algorithms, and from deciding when to begin execution. Meta-level control algorithms can be simple heuristics or a recursive application of the full planning algorithm. In this section, we review general work on meta-level control and relate it to our approach.

Meta-level control has also been called meta-level planning [Stefik, 1981]. As this term implies, an agent can plan not only the physical actions that it will take but also the computational actions that it will take. The method for performing this planning can range from simple heuristics to recursive application of the full planner. Stefik's Molgen planner uses the base level planner to create meta-level plans [Stefik, 1981]. Molgen considers two levels of meta-level planning, in addition to base-level planning. The actions at each of these meta-levels create plans for the next lower level. In contrast, our approach uses only a single layer of meta-level control and uses algorithms and heuristics tailored to making particular meta-level control decisions. Additional layers of meta-level control have a diminishing rate of return since each layer adds additional overhead and there is a limit on how much meta-level control can improve performance. We use special purpose algorithms for meta-level control to minimize overhead. It is also difficult to encode the required information in a domain model that a general purpose planner could use to make meta-level control decisions. Perhaps the next thesis in this area can address the problem of how to encode the information needed for meta-level control in a common domain description language.

Decision theory provides a measure of an agent's performance that the meta-level controller can use when making meta-level control decisions. Russell and Wefald apply decision-theory and meta-level control to standard search problems. Their DTA* algorithm uses estimates of the expected cost and expected gain in utility for possible computations to decide which computation to perform or whether to act [Russell and Wefald, 1991]. The algorithm is myopic and considers only the implications for the next action to be executed. Their method for deciding which node in the search tree to expand can be cast in terms of a sensitivity analysis. The sensitivity analysis though, considers only the effect of changing one variable at a time. The major distinction between their work and our work is the focus on actions rather than plans. The DTA* algorithm only considers the affect that a computation will have on the value of the next action while we consider the effect on the value of an entire plan. The focus on plans rather than individual action is appropriate in domains where a sequence of actions are required to achieve a task and the value of an action depends on the actions that will follow it.

In order to make the tradeoffs necessary for effective meta-level control, the meta-level controller needs some method for predicting the effect of more computation on the quality of a plan. One method for doing this is to use a performance profile. The idea comes from the study of anytime algorithms that can be interrupted at any point to return a plan that improves with more computation [Dean and Boddy, 1988]. The performance curve gives the expected improvement in a plan as a function of computation time. Anytime algorithms can also be combined to solve complex problems. Zilberstein and Russell look at methods for combining anytime algorithms and performing meta-level control based on multiple performance curves [Zilberstein and Russell, 1992]. Combining anytime algorithms produces new planning algorithms that are also characterized by a performance curve. In our work, we use performance curves to predict the performance of the two-opt algorithm for tour improvement when deciding when to begin execution for the robot courier. In addition, we also parameterize the performance curve on the size of the planning problem and use it to make predictions of planner performance if the agent were to begin execution and the planner were left to work on a smaller problem.

An alternative to using performance curves is to use the performance of the planner on the current problem to predict the future. Nakakuki and Sadeh use the initial performance of a simulated annealing algorithm on a machine shop scheduling problem to predict the outcome for a particular run [Nakakuki and Sadeh, 1994]. They have found that poor initial performance on a particular run of the algorithm is correlated with poor final performance. This observation is used to terminate unpromising runs early and restart the algorithm at another random initial state. Using initial performance to predict future performance could be used for in our robot courier domain to get better predictions of future tour improvement on a particular problem. The key question is how much history information to maintain in order to do the prediction on-line. The advantage of using a performance profile is that it does not incur any overhead for maintaining history or require extensive computations to make predictions.

2.2 Plan Generation

Ideally, a planner would generate only a single plan with the highest expected utility. However, this is not possible for any but the most trivial planning problems. Instead, planners either try to generate a single plan with high expected utility or generate a sequence of plans and quickly eliminate inferior ones. Generating a single high-utility plan makes planning efficient, but foregoes the opportunity to improve performance though more planning. Producing multiple plans and comparing them offers the opportunity to find the plan with the highest expected utility, but may reduce performance by taking too long to do the planning.

Etzioni's work on tractable decision-theoretic control produces a single high utility plan efficiently by using an approximate greedy control algorithm [Etzioni, 1989]. Etzioni assumes that there are a set of actions to accomplish each goal and that the agent knows the duration and probability of success for each action. The meta-level controller constructs plans of action (where actions can be computations) using a greedy heuristic. Each time through the control cycle, the computational action with the highest marginal utility is selected for execution. The objective of the meta-level controller is to allocate the computational time before a given deadline to create a plan to achieve a set of goals. The planner learns the expected marginal return of each computational action by recording its performance on previous runs.

In classical artificial intelligence planning, the planner produces only a single plan to achieve its goals, but there can still be criteria for preferring one plan over another. Typically, shorter or less costly plans are preferred. Perez's thesis work explores methods for learning search control rules to improve plan quality [Pérez, 1995]. These rules depend only on local information and, while they tend to improve plan quality, there is no guarantee on how far the resulting plans are from the best plan. As with any method that attempts to produce a single high quality plan, there is no way to make the tradeoff between more planning and plan quality. In our approach, we generate a sequence of plans and explicitly make the tradeoff between plan quality and more planning.

2.2.1 Dominance

Rather than attempting to generate a single high quality plan, a planner can generate a sequence of plans and compare them to select the best one. The method for showing that one plan is better than another is the key to efficient planning. Methods that can compare partially elaborated plans allow the planner to quickly eliminate inferior plans, before a lot of computation is wasted on refining them. Wellman's Sudo planner uses qualitative reasoning to construct non-dominated plans [Wellman, 1988]. The planner eliminates non-sense plans, such as performing a medical test and ignoring the results. The planner uses qualitative probabilistic networks to eliminate dominated plans, without the need for detailed quantitative information on such things as the probability of a false negative result for a test. The planner produces a set of reasonable plans, those that are non-dominated, but cannot determine the best plan. In contrast, the planners we examine in this dissertation use quantitative reasoning and ranges of expected utility to determine dominance. The advantage is that we can use quantitative probabilities and utilities to determine the preferred plan. The disadvantage of using ranges of expected utility is that they are not as powerful for showing dominance and may cause the planner to expend more effort refining inferior plans in order to show that they are inferior. Clearly, combining the two approaches may lead to more efficient planning in the same way the Simmons combines associative and causal reasoning in the Gordius planner [Simmons, 1992].

Another technique that has been used to show dominance of one plan over another involves the use of stochastic dominance [Whitmore and Findlay, 1978]. Stochastic dominance makes use of probability distributions to show that one plan dominates another, given characteristics of the utility function, such as risk aversion. Wellman has made use of stochastic dominance to efficiently create plans in probabilistic transportation domains[Wellman *et al.*, 1995]. We believe that our methods for performing meta-level control could be adapted to planners that use stochastic dominance. However, rather than using sensitivity analysis to focus effort on refinements that affect the upper bound on expected utility, we would use sensitivity analysis to select refinements that affect the uncertainty in the probability distributions used to show stochastic dominance.

2.3 Refinement Guiding

Meta-level control for plan refinement selects which part of a partial plan to more fully elaborate. In a decision-theoretic planner, refining part of a plan gives tighter bound on the expected utility of the resulting plans. Tighter bounds allows the planner to distinguish between plans and focus its efforts on plans with high expected utility. The task of the meta-level controller is to select refinements that have a large affect on expected-utility bounds and require little computation. This problem is analogous to the flaw selection problem in classical partial-order planners where the planner chooses which open condition or threat to work on. Joslin and Pollack suggest a heuristic that selects which "flaw" in the plan to work on by determining which one produces the fewest children and hence has the least cost in terms of computation [Joslin and Pollack, 1994]. Their strategy, called *least cost flaw repair*, is similar to our approach where we try to minimize the amount of computation. The difference is that we also take into account the effect that a computation can have on the expected-utility bounds, which is a measure of the importance of each flaw to the quality of the plan. Minimal cost refinements are not useful if they don't affect the expected-utility bounds. For a classical planner, fixing any flaw has equal value.

The Pyrrhus planner combines classical partial order planning with utility theory to create a value-directed planner [Williamson and Hanks, 1994]. Williamson and Hanks look at applying flaw selection strategies for classical planners to their value-directed planner. They also create two new heuristics, *least-upper-bound* and *sum-of-upper-bounds*, that take into account the effect that a refinement has on the upper bound of expected utility for the resulting plan. The least-upper-bound heuristic chooses the refinement that produces a child with the lowest upper bound on expected utility. This heuristic in effect selects the refinement with the biggest effect, while ignoring the computational cost. The sum-of-upper-bounds heuristic selects the refinement where the sum of the upper bounds on the resulting plans is lowest. This heuristic takes into account, to some extent, the effect of the refinement and the cost of computation. Williamson and Hanks show empirically that the sum-of-upper-bounds heuristic performs well for a range of planning problems. In section 7.4, we show how this heuristic approximates our idealized algorithm for refinement selection and argue that this is why the heuristic performs well.

2.3.1 Sensitivity Analysis

In order to use the sum-of-upper-bounds heuristic for flaw selection, the Pyrrhus planner performs each possible refinement to determine the upper bound on expected utility for the resulting plans. The problem with this approach is that it wastes computation doing refinements that the planner will throw away. Instead of performing refinements in order to determine their effect, we use sensitivity analysis to estimate the affect of a refinement. The sensitivity analysis methods we use are based on methods for performing Bayesian sensitivity analysis [Insua, 1990, Insua and French, 1991]. These methods allow utilities and other values to be represented by families of functions that can be mapped onto the ranges of values that arise in a partially refined plan. For the planners we examine in this dissertation, the upper bound on expected utility for each plan is the critical for showing dominance. As a result, we select refinements to which the upper bound on expected utility is most sensitive.

The use of sensitivity analysis can also be considered in the more general context of model selection for reasoning. Any model of the environment is necessarily an abstraction. More abstract models typically take less computation to estimate values, but the resulting estimates are less exact. The expected utility of a plan is an estimate based on the model of the environment used to evaluate the plan. More detailed models can give tighter bounds on the estimate, but require more computation to evaluate. In this context, model refinement is related to plan refinement. Sensitivity analysis has been used in model refinement to decide when to move to a more detailed model in order to get parameter estimates with the appropriate accuracy [Weld, 1987, Weld, 1991]. A sensitivity analysis in this case is used to justify decisions based on approximate parameters. When a sensitivity analysis shows that a decision is not justified by the accuracy of the models used, the system moves to a more detailed model. The work uses qualitative reasoning techniques to perform the sensitivity analysis and requires that each model provide information on its accuracy. Similar methods may be applicable to model refinement decisions for decision-theoretic planners.

2.4 Commencing Execution

The decision of when to begin execution is critical for on-line systems that must perform a task in a timely fashion. The anytime-algorithm approach developed by Dean and Boddy uses a performance curve to characterize the expected rate of plan improvement [Dean and Boddy, 1988]. Execution is begun when the rate of plan improvement falls below the rate of execution. Their approach does not take into account the possible overlapping of planning and execution when making decisions. It can be viewed as an approximation of Good's algorithm where the performance curve provides an estimate of the perfect information about computations. Our approach also uses a performance cure to provide missing information, but we take into account the overlapping of planning and execution and use the performance curve to predict the effect of reducing the size of the problem that the planner is working on. We also consider the choice of when to begin execution for each individual action rather than for a plan as a whole. When we begin execution, we commit only to executing the first action in the current best plan rather than the entire plan.

Russell and Wefald's work on Decision-Theoretic A* (DTA*) addresses the question of when to begin execution as well as search control [Russell and Wefald, 1991]. Their meta-level control algorithm is also based on Good's idealized algorithm. Like our algorithm, DTA* considers the choice of when to begin execution for each step. To estimate the value of computation, they analyze the partially expanded mini-max or mini-min search tree. They determine which nodes would have to change value in order to change the preferred first action. To calculate the value of a computation (expanding a search tree node) they look at the amount by which the computation could change the expected utility if the preferred action changed, taking into account the delay that the computation would introduce. They then estimate the probability that the computation will change the preferred action and use this estimate to calculate the expected change in expected utility for a given node expansion. Computation continues while the expected change in expected utility is positive. This strategy ignores the possibility that while the expected increase in expected utility for computations related to the first action may be positive, the expected increase in expected utility for computations related to subsequent computations may be higher. Even if the expected increase for computations related to subsequence actions is not higher, these computations can proceed in parallel with action and the combination may produce a higher expected utility.

Planning for continuous and probabilistic domains can be viewed as creating a policy for action. Tom Dean, Leslie Pack Kaelbling and others have been modeling actions and domains using Markov models [Dean *et al.*, 1993]. Plans for these representations are policies that map states to actions. Planning consists of modifying the envelope of the current policy and optimizing the current policy. Deliberation scheduling is done while acting in an attempt to provide the highest utility for the agent. Work has also been done on deliberation scheduling when the agent has a fixed amount of time before it must begin acting. This work has not directly addressed the question of when the agent should begin acting. It is assumed that the agent is given a arbitrary deadline of starting to execute. Techniques developed in this work could be used to decide when to start execution eliminating the need for an arbitrary start deadline. These techniques may eventually be applicable to domains like Xavier route planning that uses a Markov model for position estimation. However, at this time even the approximation algorithms for generating policies are impractical for the size of problem we must solve [Lovejoy, 1991, Cassandra *et al.*, 1994].

Chapter 3

Meta-level Control

3.1 Objective of Meta-Level Control

Meta-level control is control of computational processes. It involves deciding which computations to perform when and whether to compute or act. The purpose of meta-level control is to improve the performance of an agent by allocating computation effectively, taking into account the time cost of computation. Meta-level control is appropriate whenever there are computational processes that allow approximation and where partial results are useful. Such is the case in planning where finding a near optimal plan can improve performance over using no plan and partial plans can be used to guide action.

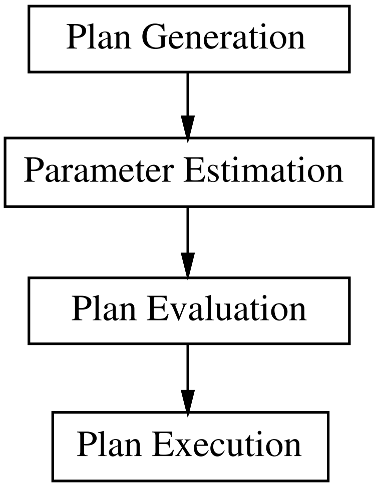

Figure 3.1: A decision problem involves 4 steps.

In this chapter, we examine the decision-theoretic planning process and identify the relevant meta-level control decisions at each step. In subsequent chapters, we examine each of these decisions in detail and show how to make each decision for a range of decision-theoretic planners.

A decision-theoretic planner finds plans with high expected utility by approximating a complete decision process. The steps in a decision process are to generate the set of candidate plans, estimate parameters, evaluate the plans and select the plan with the highest expected utility for execution (figure 3.1). In approximating the complete process (figure 3.2) the planner may generate only a subset of plans and may only partially elaborate some of the plans, giving plans with abstract actions and unplanned contingencies. Parameters, such as the cost of gas or the probability of rain next weekend, may not be estimated exactly, but given only rough bounds. Rough estimates are sometimes good enough to choose between plans and take less computation to generate. For example, it may be enough to know that the price of gas is between $1 and $5 a gallon to decide between flying somewhere and driving. With partial plans and partial estimates of parameters, plans cannot be evaluated to determine their exact expected utility. Instead, the planner can give only bounds on the expected utility. Finally, deciding which action to execute does not necessarily require waiting until the plan with the highest expected utility has been found. Instead, beginning execution sooner with a partial plan that may not have the highest expected utility in order to improve performance.

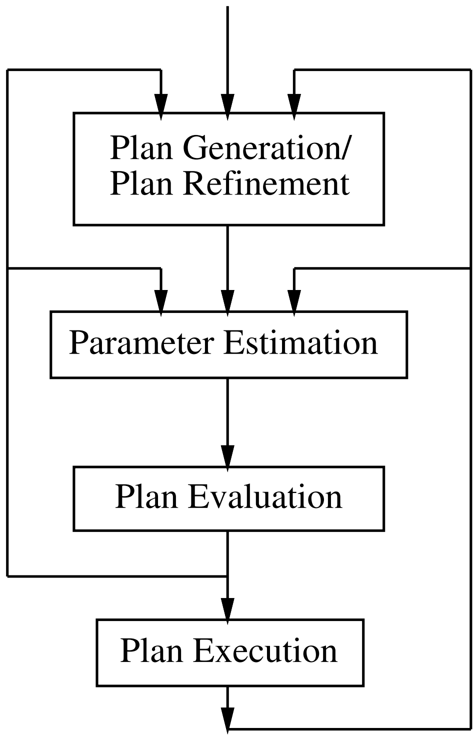

Figure 3.2: The decision problem can be approximated and solved iteratively. Partial plans are generated and then refined.

This chapter begins by detailing the meta-level control decisions needed when approximating each step in a decision process. We consider the related meta-level problems of solving the decision problem to find a plan with the highest expected utility using the least amount of computation and of producing a resource-bounded rational agent. These problems are related. If the plan with the highest expected utility can be found quickly enough, finding it and using it leads to the best performance. We then briefly discuss related meta-level control decisions appropriate for learning agents, to give a more complete picture of the meta-level control decisions a complete agent may have to make. Finally, we examine two issues related to the implementation of meta-level control. The first is whether the control should be done on-line or off-line. On-line control can adapt the decisions to the particular circumstance, but incurs an overhead in computation. The second issue concerns the quality of models and estimates used to make meta-level control decisions. Highly detailed models can provide better information for making meta-level decisions but may be more computationally expensive to use.

3.2 Meta-Level Control Questions for Decision-Theoretic Planners

In this section, we examine each step in a decision problem and discuss how artificial intelligence planning techniques can be applied to solving the decision problem. For each step, we also examine the meta-level control decisions to be made, including the information available to make the decision and the criteria for making a good decision. This will provide the framework for examining a set of decision-theoretic planners and methods for making the meta-level control decisions in each one.

3.2.1 Plan Generation

Generating the set of candidate plans is the first step in a decision process and has been the focus of much of the work in the field of artificial intelligence planning. In general, an artificial intelligence planner takes a domain description that describes the environment and the available actions and a task description, and produces a plan to achieve the task in the given environment. Planners differ in terms of the types of environments, actions and tasks they can generate plans for and the efficiency with which they can do it.

In a full solution to a decision problem, the complete set of feasible plans is created for evaluation. Traditionally, artificial intelligence planners find only a single feasible or satisficing plan [Simon and Kadane, 1974] since all such plans were considered equivalent. The planning process uses the task specification, which can be a set of goals or a task network, in order to guide the search for satisficing plans, focusing the search on plans that can potentially accomplish the task. Typically, a means-ends analysis is used to accomplish this focusing of effort. In a decision-theoretic framework, there is not necessarily a set of goals to be satisfied or a task tree to expand, and satisfying a set of goals need not be the only criterion for evaluating a plan. The desired outcome for the task may be discribed only in terms of a utility function, and there may not be any goals in the traditional planning sense. Instead of trying to achieve specific conditions in the world, the task may be graded rather than binary and involve tradeoffs. An example would be trying to get to Washington, DC as soon as possible, while spending as little money as possible.

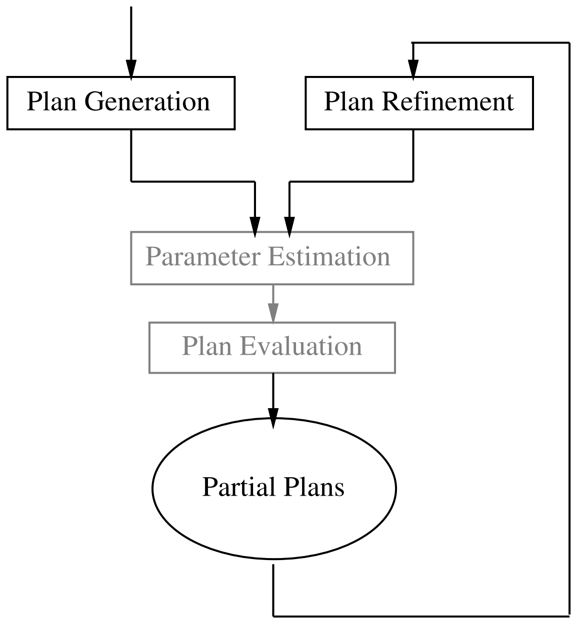

Figure 3.3: The decision problem can be approximated and solved iteratively.

Instead of generating a single plan, the plan generator for a decision-theoretic planner may generate a sequence of plans that can be evaluated and compared. In looking for plans that get to Washington as quickly and as cheaply as possible, the planner may need to generate the entire set of plans that get the agent to Washington and then evaluate each one to make tradeoffs between cost and time. Generating subsequent plans is typically done by forcing the planner to backtrack over previous decisions. Artificial intelligence planners generally allow backtracking in order to ensure completeness.

For decision-theoretic planners, a goal such as getting to Washington acts as a constraint on the set of feasible plans, restricting the planner to consider only plans that get the agent to Washington. Another form of goal gives objectives to maximize or minimize rather than absolute conditions to be met. Reducing cost and time are goals in the ceteris paribus sense, [Wellman and Doyle, 1991], since plans with reduced cost or time will have a higher expected utility, all other things being equal. Such goals can be recognized when the partial derivative of utility is always positive (or negative) for all allowable values of an attribute. This form of goal can also be used to limit the number of plans to be generated, if there are some constraints on the sequence of plans generated. For example, if each action takes a minimum amount of time T_{min} and has a minimum cost C_{min}, then any plan of length n will take time at least $n * T_{min}$ and cost at least $n * C_{min}$. If a plan is found that gets to Washington in k steps, with time T_k and cost C_k, then only plans with length $n \leq max(\frac{T_k}{T_{min}}, \frac{C_k}{C_{min}})$ could potentially have a higher expected utility. Once all the plans of this length or shorter have been generated, the planner can halt and guarantee that it has the plan with the highest expected utility. If the planner generates plans in a monotonically non-decreasing order of length, this is easy to achieve.

Partial Plan Generation

In approximating the solution to the decision problem, the plan generator does not have to create a complete set of fully elaborated plans in a single step. Instead, the plan generator is called iteratively to generate a sequence of partially elaborated plans. These partial plans are then passed on for evaluation and possible execution as they are generated. If, in this process, a partial plan is evaluated and found to be dominated by another plan, then it can be discarded. If a partial plan is found to be potentially optimal it can be chosen for further elaboration so that it becomes more operational and its expected utility can be estimated more exactly. Logically, the plan generator can be split into two parts (figure 3.3) one that generates the initial partial plans and one that takes a partial plan and refines it to produces a set of more fully elaborated plans.

The generation of partial plans makes the planning process more efficient, limiting the amount of work done on plans with low expected utility and by allowing execution to begin before a complete plan is found. A partial plan represents the set of plans that are the possible refinements of the partial plan. If a partial plan is evaluated and discarded because it has low expected utility, the planner has effectively pruned the set of plans that the partial plan represents without having to actually generate them.

A complete plan specifies a complete policy for acting that includes only operational actions, resolves all action conflicts and specifies actions for each possible contingency. *Operational actions* are actions that the agent can directly execute. If the planner is supplied with a set of goals, a complete plan would also satisfy all the goals. A partial plan may relax any of these requirements.

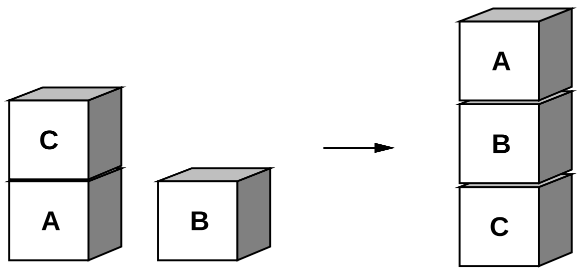

Figure 3.4: In the three block Sussman anomaly, the goal is and on(B,C) on(A,B). The problem is that putting block B on block C to achieve the first goal is undone when block B is cleared to move it on top of block C to achieve the second goal.

A partial plan may include abstract actions that are not operational but represent one or more sequences of operational actions. For example, an abstract action "GO_TO(Washington)" may represent operational actions that get to Washington via car, airplane and train. In this case, a partial plan with the action "GO_TO(Washington)" would represent the set of plans that include going to Washington using each of the three modes of transportation, each one using a different route.

Plans may also be partial in the sense that they do not resolve all action conflicts. For example, a partial plan may include actions "GO_TO(Washington)" and "GO_TO(Ottawa)" but not specify an ordering constraint between them, although it is obvious that the agent cannot travel to Washington and Ottawa at the same time. It is also the case that the cost and time needed for the "GO_TO(Washington)" action depends on the start location. Without ordering the actions, the partial plan represents the set of plans with all possible orderings. Other action conflicts include protection violations, such as the classic Sussman Anomaly, figure 3.4, where placing block B on block C clobbers the clear(c) condition needed to move block C to the table.

A partial plan may address only part of the task by achieving only some of the goals. For example, a plan to go to Ottawa partially satisfies the task of visiting two capital cities and represents the set of plans that includes Ottawa as one of the capital cities.

Meta-Level Decisions

The meta-level decisions needed for controlling plan generation are when to generate another plan and when to refine a partial plan. When refining a partial plan, the partial plan is selected from the pool of partial plans and the part of plan to refine must be chosen from the selected plan.

1. Generate another plan?
2. Refine a partial plan?
   (a) Which partial plan to refine?
   (b) Which part of the selected plan to refine?

In this section, we examine each of these choices, the information available to make the choice and the criteria for making a good choice.

Plan Generation: Ideally, the plan generator would generate only a single plan that would have the highest expected utility and would not waste any effort generating less optimal plans. If this were possible, then there would not be much of a planning problem and the meta-level control problem would be trivial. When the plan generator does not generate the optimal plan first, the question becomes how many plans to generate. The question may still be trivial if the planner begins by generating a single universal plan from which all possible plans are produced by refinements of the universal plan. Hierarchical task network planners [Sacerdoti, 1977] that begin with a single abstract task network and partial order planners [Pemberthy and Weld, 1992] that start with a plan consisting solely of place holding begin and end actions fall into this category. For these planners, the entire plan generation process can be viewed as plan refinement. The meta-level control decisions for this class of planners is covered in the paragraph on meta-level control for plan refinement.

For non-trivial plan generations problems, we consider first the case where the objective is to find the optimal plan with the least amount of computation. We then look at the problem of quickly finding satisficing solutions with high expected utility. For a decision-theoretic planner, a satisficing solution is one that represents a satisfactory tradeoff between plan quality and computation.

When looking for a plan with the highest expected utility, we need to continue the plan generation process only until an optimal plan has been generated. If it is not possible to recognize an optimal plan, then plan generation has to continue until it can be shown that the remaining ungenerated plans are all sub-optimal, or until all possible plans have been generated. The control question is one of recognizing an optimal plan or showing limitations on the ungenerated plans.

If a utility function has a maximum value, then any plan that achieves this maximum utility is necessarily optimal. Such is the case with utility functions that have a maximum value when a set of goals is achieved and a lower value otherwise. For example, if the goal is to get to Washington and spend less than \$500, a plan to fly that costs \$450 achieves the maximum possible utility. This is the same condition that allows plan generation to stop in a classical artificial intelligence planner when a set of goals has been satisfied.

In many cases, the optimal plan will not have the maximum conceivable utility because there are conflicting "goals". If the goal is to get to Washington by tomorrow at 6pm and spend as little money as possible, then the maximum utility occurs for a plan that gets there on time and spends no money. Such a plan may not be available. To recognize that a plan with the highest expected utility has been generated, something must be known about the plans yet to be generated. In the case of our example, if we could generate plans in a monotonically non-decreasing order of cost, then when we found a plan that got us to Washington by 6pm tomorrow, plan generation could be halted. Any ungenerated plans that achieved the goals would necessarily cost as much, or more, and need not be considered. It may not be possible to generate plans in order of monotonically non-decreasing cost, but it may be possible control the search for feasible plans so that shorter plans are generated before longer plans. If actions have a minimum cost, then it is still possible to recognize a plan with the highest expected utility before all plans are generated. If it is not possible to characterize the ungenerated plans or to recognize an optimal plan, then plan generation must continue until all plans are generated, if the planner is to be guaranteed of finding an optimal plan.

When attempting to find a satisficing solution quickly in order to improve overall performance, the plan generator does not need to guarantee that it has the plan with the highest expected utility and can halt when an good solution is found. The decision to halt is a tradeoff between the time needed to find a better plan and the amount by which the current plan could be improved. The key input for this decision is an estimate of the expected plan improvement versus the expected computation time. This type of information can come from prior experience and be compiled into performance curves that summarize the expected performance of the planner [Boddy, 1991a]. The performance of the planner on a particular problem can also be monitored and when the rate of plan improvement falls below a threshold, plan generation can be halted. The question of when to halt plan generation is closely related to the question of when to begin execution and will be discussed further in section 3.2.4.

Plan Refinement In addition to meta-level decisions about plan generation, there are also decisions to be made about plan refinement. When a partial plan has been generated, it can be sent back to the planner for further elaboration. Elaborating a partial plan by refining an abstract action, resolving a conflict or planning for a contingency results in a set of more complete plans. The meta-level decisions include which plan to select for elaboration and which part of the selected plan to elaborate. In making this decision. the planner has access to the evaluation of the partial plan, the structure of the partial plan and the domain model. This information can be used to determine the number of plans that could result from the elaboration and allows for calculations, including sensitivity analysis, to determine the possible effect of the elaboration on the expected utility of the resulting plans. As with plan generation, the expected computation needed to elaborate a partial plan can be estimated from previous experience on the same, or other, planning problems.

When searching for a complete plan with the highest expected utility, the plan refinement process must completely elaborate one such plan, but avoid doing work on other plans. Again, if such a plan cannot be recognized immediately, the refinement process must continue until a plan with the highest expected utility can be distinguished from the sub-optimal plans. The objective of the meta-level control is to completely elaborate an optimal plan and do as little work as necessary to distinguish the optimal plan from sub-optimal plans.

When searching for a satisficing plan, the objective of the meta-level controller is to find a plan with high expected utility using a minimum of computation. The plan does not have to be fully elaborated since execution can begin if the first action in the plan is operational. The planner can then continue to refine the rest of the plan while the first part is being executed. An ideal meta-level controller would focus all the refinement effort on a single high utility plan and specifically on refining the beginning of the plan. But, without knowing a priori which plans have high expected utility, the planner may have to expend some effort refining low expected utility plans to show that they have low expected utility. The planner may also have to refine more than just the beginning of a plan in order to determine its expected utility. The meta-level controller must trade plan quality, measured as expected utility, for planning time, which delays execution. The criteria for a good decision selecting refinements that tend to improve the expected utility of the best plan faster than delaying execution reduces expected utility.

3.2.2 Parameter Estimation

In order to evaluate a plan, the parameters needed to calculate the expected utility need to be estimated. In many cases, the parameters are explicitly supplied as part of the domain model. In a transportation domain, examples include the distance between cities and the fuel efficiency of a vehicle. Other parameters may be known only implicitly. For example, the domain model might include the current weather conditions and a model of how the weather evolves over time. If the utility of a plan depends on whether it is raining tomorrow in Washington at 6pm, then the model could be used to predict the probability of rain. The model used to make the prediction may allow for different levels of accuracy that require different amounts of computation. The weather model may make better predictions if forward projection is done in steps of 1 minute rather than 1 hour. The tradeoff is between getting better parameter estimates that lead to better estimates of the expected utility of a plan, and the computation required.

Better estimates of expected utility can help to distinguish high expected utility plans from low expected utility plans, but highly accurate estimates may not be required. It may be enough to know that the probability of a thunder storm is less than 10% in order to choose flying over driving to Washington. In Hanks's dissertation on using projection to estimate the probabilities of various outcomes, plan projection is stopped when a probability is known to be above or below a threshold given in a query [Hanks, 1990]. The value of the probability that would allow one plan to be selected over another could be used to form the query. Hanks's plan projector allows another form of query that returns a probability estimate within a given error bounds.

The meta-level questions here are how accurately does the parameter have to be estimated and how to control the parameter estimation process. The information available includes the plans being evaluated and estimated performance of the parameter estimation code. As with plan generation, the performance of the parameter estimation code can be characterized by past performance or by performance on the current problem.

It is important to note that parameter estimation, in this sense, is different from information gathering. Actions, such as turning on the radio to get a weather report form part of the plan being created rather than being a method for evaluating a plan. The plan produced by the planner might be a contingent plan to turn on the radio to get the weather report and then to fly if the prediction is for clear weather, and to drive otherwise. We may even delay part of the planning process until we have gathered some information. For example, we start execution of the partial plan to turn on the radio and then complete the plan only after we have the weather report. There are interesting tradeoffs between information gathering and parameter estimation that involve trading off the cost of computation and the quality of the estimate for the cost of action. In these situations, the amount of computation used to estimate a parameter should always be limited to the cost of acquiring the information from another source.

With partial plans, the parameter estimation problem becomes more complicated. For example, estimating the travel time and cost for the "GO_TO(Washington)" action depends not only on the start location, which may be unspecified, but also on the mode of transportation. No amount of parameter estimation, short of elaborating the plan, can give an accurate estimate of these parameters. Instead, the estimate needs to take into account the range of possible values. This could be done by calculating a mean value, with an appropriate variance. The problem with this approach would be in selecting the appropriate probability distribution for calculating the mean. A more common method is to give upper and lower bounds on the value of the parameter. In the "GO_TO(Washington)" example, the time could be limited by the shortest and longest trips to Washington from any location. Similarly, cost could be limited by the most and least expensive trips. As a partial plan was elaborated to include the start location or the mode of transportation, these estimates could be updated to produce smaller ranges.

Meta-Level Decisions

Some planners require that parameter estimates be explicitly given in the domain description. For these planners, the meta-level control for parameter estimation is moot. Other planners use techniques, like Bayesian Networks, simulation and Monte-Carlo techniques to estimate parameters. Although the specific meta-level control decisions for these planners depend on the specific techniques begin used, there are some general questions that are applicable to all forms of parameter estimation.

1. When to refine an estimate?

2. How to refine an estimate?

3. How accurate does an estimate need to be?

4. What are the "landmark" values?

The crux of the problem is to allocate computation in a way that provides parameter estimates that are the most useful for finding either the plan with the highest expected utility or a satisficing plan with high expected utility. The most basic question is when to refine an estimate and what method to use. Often, there will be only one method for estimating each type of parameter. Even if there is only a single method, the meta-level controller may need to provide information to control the parameter estimation process. For example, Hanks's plan projector makes use of landmark values and requirements of parameter accuracy to control how much computation it does. Other parameter estimation methods may take other parameters that control how much computation it does.

The specifics of how to provide effective meta-level control for parameter estimation will depend on the specifics of the parameter estimation methods used. Also, many current planners avoid the problem of controlling parameter estimation by requiring that parameters be explicitly given in the domain model. In the remainder of this dissertation, we will assume that relevant parameter estimates are given explicitly in the domain model or can be quickly calculated in full detail. We leave the problem of controlling parameter estimation processes and trading-off parameter estimation for information gathering for future work.

3.2.3 Plan Evaluation

In order to select a plan from the set of candidate plans, each plan must be evaluated to determine its expected utility. This is done by applying the utility function to each possible outcome, or chronicle, for a plan and weighing the value by the probability of the chronicle.

$$\sum_{\text{c:chronicle}} p(c) * U(c)$$

A *chronicle* is a description of the state of the world over time [McDermott, 1982]. When a plan has been fully elaborated, calculating the expected utility is a straightforward application of the utility function and the result is a point-valued expected utility. With partial plans and partial estimates, plan evaluation becomes more difficult.

The problem with evaluating a partial plan is that it may be impossible to determine the set of possible chronicles and their probabilities. If actions in the plan are unordered, then the order of states in the chronicle cannot be determined. Abstract actions represent sets of plans and hence sets of sets of chronicles. A partial plan may not specify what to do in all possible circumstances. Without knowing the set of chronicles and their probabilities, a planner cannot calculate an exact expected utility for a partial plan.

Instead, we can treat the calculation of the expected utility for a partial plan as a parameter estimation process. In this case, the parameter is the expected utility, which is a function of other parameters. Rather than attempting to calculate an exact value of expected utility for partial plans, the value can be bounded. An upper bound on the expected utility allows the planner to determine how promising a particular plan is. A lower bound limits the disappointment that selecting a particular plan could have. Comparing bounds also allows the planner to prune clearly sub-optimal plans from the search space without further elaboration. The problem for the planner is how to determine these bounds. The meta-level control question is to decide how much computation to allocate to finding tighter bounds, both upper and lower, on expected utility for a plan.

For a particular partial plan, the upper bound on expected utility is critical for determining whether it belongs in the set of potentially optimal plans. If the upper bound can be shown to be less than the expected utility for some fully elaborated plan or the lower bound of any other partial plan, it can be pruned from the search space. A lower bound on expected utility is useful as a guarantee of minimum quality of a partial plan. Lower bounds on partial plans are useful when looking for a satisficing plan. A decision-theoretic planner can find plans with the highest expected utility if it can calculate upper bounds on expected utility for partial plans and exact values of expected utility for fully elaborated plans.

One approach to calculating an upper bound on expected utility for partial plans is to take an optimistic view of all possible outcomes. With a lack of direct evidence to the contrary, the assumption is made that all goals are achieved and that all conflicts and unplanned contingencies are handled without incurring additional costs. This includes assuming that all unconstrained actions can happen in parallel. For example, a plan included the unconstrained actions "GO_TO(Washington)" and "GO_TO(Ottawa)", may take only as long as the longest of the two trips. Although, in this case, simultaneously traveling to Washington and Ottawa is impossible, determining this may require additional computations. Assuming that actions can take place in parallel does not require additional computation and provides a valid bound. Any uncertainty in the probability of each outcome is resolved to assign the maximum possible probability to the most favourable outcomes. Lower bounds can be calculated in a similar way, using a pessimistic assumption about possible outcomes.

Meta-Level Decisions

As with the estimation of other parameters, the estimation of expected utility depends on the specifics of the planner and the domain. The cost of evaluating the utility function may depend on the representation of a plan. For example, it may take more computation to evaluate the expected utility of a partial-order plan than a totally ordered plan. The exact form of the utility function may also determine how easy it is to evaluate or bound. For example, decomposable utility functions may be efficiently calculated using a divide and conquer strategy. Specific domains may also admit simplified formulas that quickly bound a utility function. For example, the straight line distance to a destination and the maximum speed of a robot can be used to limit the earliest arrival time. However, there are some general meta-level control decisions that are common to evaluating plans.

1. How much computation to allocate to evaluating expected utility?
    (a) When to refine a bound?
    (b) How tight does a bound have to be?
2. When to estimate lower bounds as well as upper bounds?
3. When to use other methods for showing dominance?

The basic questions is to decide when to evaluate a plan and how much computation to allocate to this process. The objective of meta-level control is to identify the plans with high expected utility with a minimum of computation. One way of evaluating partial plans is to bound the range of expected utility. Using this approach, a meta-level controller may have control over how tightly the bounds are computed and whether the planner calculates a lower bounds as well as an upper bound. But using ranges of expected utility is not the only method of evaluating a plan. Sometimes a planner can show that one plan dominates another (and therefore has higher expected utility) without calculating the expected utility of either plan. In section 4.2 we will examine some methods of showing dominance.

A full examination of the plan evaluation for decision-theoretic planners would have to cover the problems of estimating expected utility and of showing dominance between plans. Using bounds on expected utility is only one technique for evaluating expected utility and showing dominance. But even if we restrict our investigation to methods for finding bounds on expected utility, we are still faced with a daunting task. Since a utility function can be an arbitrary function mapping outcomes to values, methods for finding bounds on expected utility includes the all the possible methods for obtaining bounds on arbitrary functions.

To make the examination of plan evaluation for decision-theoretic planners tractable, we restrict ourselves to planners that calculate bounds on expected utility for partially refined plans. We assume that each planner has a single method for calculating bounds and that this method has been designed to calculate bounds efficiently without additional meta-level control. In addition, we will examine a range of methods for showing dominance and demonstrate how they are used in the four planners we examine. As we did with parameter estimation, we will leave the problem of controlling the process of bounding expected utility to future work.

3.2.4 Plan Execution

When solving the complete decision problem, deciding when to begin execution is trivial. Execution is begun when a plan with the highest possible expected utility is found. It is controlling the plan generation and plan refinement processes leads to an efficient solution of the decision problem.

When looking for a satisficing solution that produces the best overall performance, the decision to begin execution is a tradeoff between the current best plan, one that might be only partially elaborated, and continuing to plan in a hope of finding a better plan. The opportunity cost of forgoing possible improvements must be weighed against the cost of further delays for additional computation and is the crux of the meta-level control problem for rational agents with limited computation.

Meta-Level Decisions

1. When to begin execution of the current best plan?

In making the decision to begin execution, the planner does not need to elaborate an entire plan, but simply needs to create enough of a plan to give high expected utility. This may entail deciding only on the first action or may require a complete plan. In benign environments where performing actions and undoing their consequences is relatively inexpensive, simply deciding on the first action may be enough. In environments where an agent can become trapped, the plan must be detailed enough to avoid getting painted into a corner [Nourbakhsh, 1996]. The plan may also have to cover unlikely, but catastrophic events. For example, a plan may need to include purchasing fire and accident insurance before beginning to build a house. On the other hand, delaying planning decisions until after the start of execution has advantages. Overlapping planning and execution can improve performance. As a plan executes, more information is available that can eliminate the need to deal with unrealized contingencies and improve estimates needed to decide between future courses of action.

3.3 Other Meta-Level Control Decisions

Although planning and the related meta-level control are major factors in determining the performance of a resource-bounded agent, there are other factors that make significant contributions. In particular, learning can play a significant role in improving performance over time. Techniques have been developed for improving computational performance through speed-up learning and for creating better models and parameter estimates through inductive learning. As with planning, meta-level control can effectively focus the learning efforts. In this section, we list some other major meta-level decisions, including a subset related specifically to learning. These questions are also indirectly related to planning since improved models and estimates can lead to better plans, whereas computation used for learning can take away from the computation available for planning. We include these questions to show that there are a range of meta-level control questions related to meta-level control for agents that use decision-theoretic planning, but that are not directly related to controlling the planning process.

- **When to learn?** When should an agent do computation to improve a model, possibly incorporating new information gathered through interaction with the environment?
- **What to learn?** Which parts of a model and which parameters should the learning effort be focused on?
- **What to remember and what to forget?** How much information should be stored about past experience in order to make predications about the future? This is especially relevant for agents with limited memory that operate over long periods of time.
- **Learning method to use?** Which learning algorithm should be used and which evaluation method should be used to determine which of several possible models is better?
- **Exploration versus exploitation.** Should the agent take actions that are currently believed to be best or try other actions to gain information that can improve future performance.

3.4 Form of Meta-level Control: On-line versus Off-line

A question, left implicit to this point, is how to implement meta-level control. One dimension for characterizing a meta-level control strategy is the amount of computation done off-line versus on-line. Meta-level control can be very efficient if the meta-level decisions are made off-line and the results compiled into the planner. For example, the decision about which planning algorithm to use is often made off-line based on what is known about the domain and the characteristics of each planning algorithm. Other decisions, such as those involving search control, can include a combination of off-line and on-line control. Off-line control may be based on extensive experience or properties of the search that indicate which choices tend to produce good results. On-line control, on the other hand, can take advantage of information available about the particular situation. Using available information can lead to better decisions, but analyzing the information can take more time to make the decision. In fact, the basic question is not just one of off-line versus on-line, but one of how much meta-level control computation should be done at run time.

In a sense, off-line versus on-line is a meta meta-level question where the decision is about how to control the meta-level controller. In order to avoid an infinite regress of meta-level controllers for meta-level controllers, these meta meta-level decisions are made off-line. The basic idea is that one level of on-line meta-level control can improve performance, whereas additional levels of meta-level control offer a diminishing (and possibly even negative) rate of improvement.

3.5 Meta-Meta Decisions

One meta meta-level topic that we wish to address briefly is that of the estimates used to make meta-level decisions. These estimates include the expected amount of computation needed to perform a calculation and the effect a computation will have on a plan. If these estimates are generated off-line, then considerable effort can be put into obtaining accurate estimates. These estimates can then be compiled into performance curves for efficient run time use. For estimates generated on-line, there is a tradeoff between better estimates and the time needed to generate them. In many cases, it is likely that rough estimates can lead to good meta-level decisions. Although the decision to perform one computation over another is binary, their relative value is not. In cases where there is a vast difference in value between possible computations, rough estimates of their value should easily distinguish them. In cases where the values are nearly equal, the decision of which computation to do does not matter as much, since the opportunity cost of selecting either option is low. This suggests that using very approximate estimates for meta-level control should still produce good meta-level control decisions.

3.6 Summary

Table 3.1 summarizes the meta-level control questions outlined in this chapter. The central questions that we address in this dissertation: plan generation, plan refinement and commencing execution will be covered in detail in chapters 6, 7 and 8 respectively. To facilitate our discussion of these questions, we provide some required background in the next two chapters. Chapter 4 reviews sensitivity analysis and its application to planning when plans and parameter estimates are only partially complete. Chapter 5 introduces four decision-theoretic planners that will be used to provide examples and empirical results when examining each of the meta-level control questions.

1. Plan Generation:
   (a) Generate another plan?
   (b) Refine a partial plan?
      i. Which partial plan to refine?
      ii. Which part of the selected plan to refine?
2. Parameter Estimation:
   (a) When to refine an estimate?
   (b) How to refine an estimate?
   (c) How accurate does an estimate need to be?
   (d) What are the “landmark” values?
3. Plan Evaluation:
   (a) How much computation to allocate to evaluating expected utility?
      i. When to refine a bound?
      ii. How tight does a bound have to be?
   (b) When to estimate lower bounds as well as upper bounds?
   (c) When to use other methods for showing dominance?
4. Commencing Execution:
   (a) When to begin execution of the current best plan?

Table 3.1: Meta-level questions for decision-theoretic planners.

Chapter 4

Sensitivity Analysis

A decision is sensitive if relatively small changes in parameter values change the preferred alternative or plan. A sensitivity analysis provides a method for determining which parameters affect the preferred alternative and how much each parameter can vary without affecting the decision. The results of a sensitivity analysis indicate which parameter estimates are most critical for making the decision, suggesting that effort be spent refining these estimates. The analysis can also suggest how the preferred plan should be modified to reduce its sensitivity.

In this chapter, we give an overview of sensitivity analysis, with emphasis on its relation to decision problems and planning. We begin with a simple medical example that can be illustrated graphically. The example is used to introduce the concepts of dominance, opportunity cost, indifference range, the 100% rule and shadow prices that are central to a basic sensitivity analysis of a complete decision problem. Since establishing dominance of one plan over another is critical for both pruning the search space and sensitivity analysis, we also enumerate the techniques available for showing dominance and explain their uses. We discuss the characteristics of each method and how it can be used in planning. We then move on to discuss sensitivity analysis for partial decision problems where the alternatives or plans are not completely elaborated, and parameters may have only ranges of values. The algorithm for performing sensitivity analysis for partial plans and parameters is based on work by Insua that uses belief functions and families of utility functions [Insua, 1990]. We examine the steps in this algorithm and their relation to the underlying decision problem. Although the examples we give in this chapter involve linear problems, the techniques generalize to non-linear problems. Finally, we discuss a set of metrics used to compare the sensitivity of systems and some methods of approximating sensitivity analysis.

4.1 Example

Suppose a rural medical clinic is considering setting up a local laboratory to perform medical tests in order to save its patients the time and expense of traveling to a larger town 50 miles away. The clinic is considering building facilities to perform two tests, one a blood test and the other a urinalysis. The clinic has some resource limits and the local community has limited demand for each type of test. The problem is to decide whether the lab should offer only one of the tests or some combination of the two tests or neither test.

The particulars of the problem can be represented as a set of constraints and an objective function that is to be maximized. For the purposes of this example, each blood test requires 10 minutes of a nurse's time to draw the blood and 30 minutes of a technician's time to process the blood. A urinalysis also requires 10 minutes of the nurse's time to prepare the sample, but only 15 minutes of the technician's time to process the test. The clinic has a technician that is available for 40 hours per week and a nurse that can spend 20 hours per week preparing tests. Furthermore, suppose that some market research indicates that the current demand is for 60 blood tests per week and 140 urinalysis tests per week. The physicians at the clinic have determined that having locally available blood tests is 50% more valuable than having locally available urinalysis.

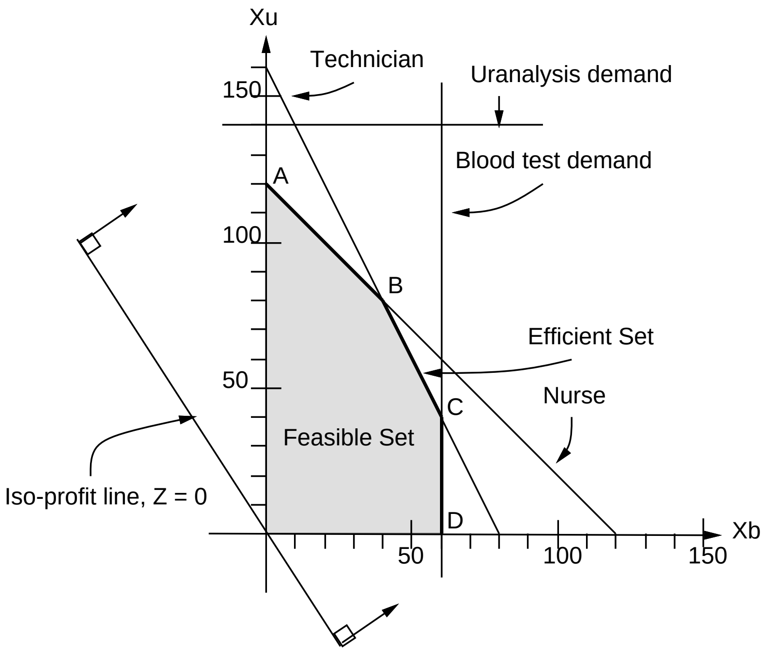

Figure 4.1: Graphical representation of the test selection problem for the clinic.

The problem can be represented using the following inequalities.

Let X_b = the number of blood tests.
X_u = the number of urinalysis tests.

$$
\begin{aligned}
30X_b + 15X_u &\leq 40 * 60 \\
10X_b + 10X_u &\leq 20 * 60 \\
0 \leq X_b &\leq 60 \\
0 \leq X_u &\leq 140
\end{aligned}
$$

$$\text{maximize}z = 30X_b + 20X_u$$

Represented in this way, the decision problem for the clinic is a linear programming maximization problem of two variables that can be represented graphically as shown in figure 4.1. The shaded area in the diagram is the *feasible set* where all constraints are satisfied. The upper-right boundary of this set, show in a heavy black line, is the *efficient set* that contains solutions that maximize objective functions that are positive linear combinations of the two tests. Any point in the feasible set dominates points below and to the left since such points represent performing fewer of one or both tests. The upper-right boundary is the set of points that are not dominated and therefore must contain the optimal solution.

To find the optimal operating point, start with a line at Z=0 and slide it in the direction of increasing Z, as shown in the diagram. The last point or set of points in the feasible set that the line touches is the optimal solution. For this problem, the point B where $X_b = 40$ and $X_u = 80$ is the optimal solution with $Z = \$30 * 40 + \$20 * 80 = \$2,800$. In general, the solution to this type of decision problem can be found efficiently using the simplex algorithm rather than the graphical method used in this case. The simplex method makes use of the fact that, for linear objective functions, the optimal set will include at least one of the points on the edge of the feasible set where two or more constraints intersect. These points correspond to A, B, C and D in our example (figure 4.1). Since we need to find only one optimal operating point, we can cast the problem as a choice of one of these four points.

Opportunity Cost

Suppose the physicians who run the clinic had decided to perform only urinalysis tests in the lab. They might have reasoned falsely that since doing only urinalysis tests would take up all the available time of the nurse and since there was a high demand for urinalysis tests, that this would produce the most benefit. The value of this alternative is \$2400, which is \$400 less than the optimal solution. The \$400 difference represents a lost of the opportunity to improve the objective function by \$400. The difference in value between a selected alternative and the optimal alternative is the known as the opportunity cost. By selecting one alternative, the decision maker gives up the opportunity of selecting another alternative that could improve the expected utility.

Sample Sensitivity Analysis

A sensitivity analysis of a decision problem determines how changing the parameters in the constraints and the objective function affects the preferred alternative and its value. Changing these parameters corresponds to translating and rotating the lines in the graphical representation of the problem. We will make extensive use of the graph to give an intuitive interpretation of sensitivity.

Consider first the resource constraints that define the feasible set. The optimal operating point, B, is bounded by the constraints on the availability of the nurse and the technician.

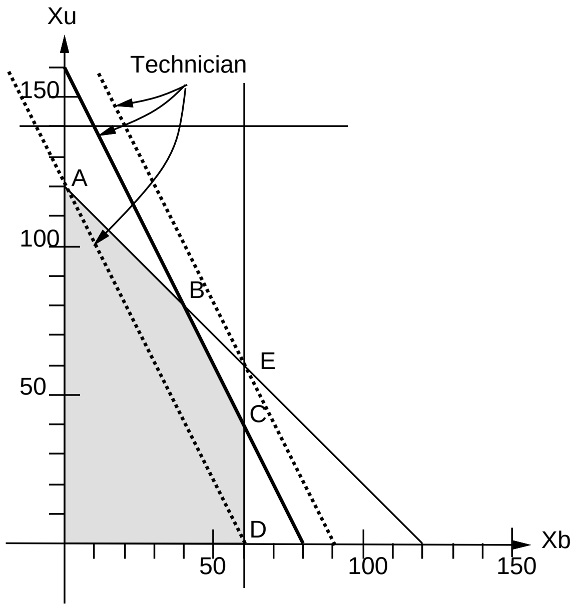

Figure 4.2: Changing the number of hours the technician is available translates the line that represents the constraint on the technician's time. In the range where the line intersects the constraint on the nurse's time between points A and E, the intersection B remains the optimal operating point and the shadow cost of the technician's time remains constant.

The constraints on demand for each test do not currently limit the value of the solution. In fact, the demand for the blood tests could decrease by 20 and the bound on demand for urinalysis could decrease by 60 without affecting the current best solution. The amount by which a parameter can change without changing the preferred solution or its value is a measure of the insensitivity of the solution to the parameter. This amount is the slop in the constraint with respect to the optimal solution. We will call this slop the *indifference range* since the choice of an alternative is indifferent to changes of the parameter in this range.

For the constraints that bound the optimal solution, any change in the parameter values will affect the position of the intersection and the value of the optimal operating point. For example, if the availability of the technician increases by an hour, then B is the point $X_b = 44$ and $X_u = 76$ and the value of the solution is $Z = \$30 * 44 + \$20 * 76 = \$2,840$. The increase in the objective function ($40) is the *shadow price*, the price that could be paid for an extra hour of the technician's time, while still maintaining the same level of profit. The shadow price for an hour of the nurse's time is $60. If the technician's time or the nurse's time can be acquired for less than the shadow price, then profits can be increased. On the other hand, the shadow prices for increasing demand for tests, by for example advertising, is zero since increasing either of the demand constraints does not affect the optimal operating point. The shadow prices remain constant for changes in constraint bounds as long as intersection B is the optimal operating point. In the example, the shadow price for the technician's time, remains constant as long as the amount of the technician's time available is between 30 and 45 hours (figure 4.2). Beyond 45 hours, the shadow price for the technician's time drops to zero since the number of tests that can be performed is limited by the availability of the nurse. Below 30 hours, the shadow price for the technician's time rises to \$80 per hour since the technician's time becomes the limiting constraint and the technician can process 4 urinalysis tests per hour, worth \$80.

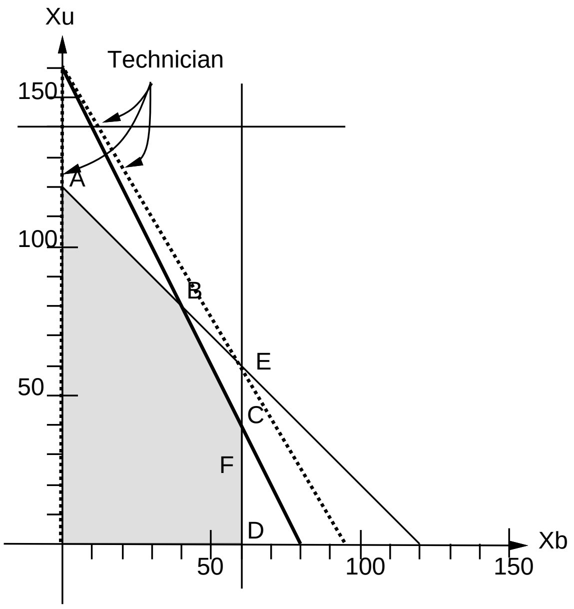

Figure 4.3: Changing the technician time required per blood test changes the intersection of the technician constraint with the Xb axis and the slope of the constraint.

A second set of parameters that affect the resource constraints are the technology parameters. In our example, these parameters are the amount of the nurse's and technician's time needed for each test. These are called technology parameters since the technology used in production determines how much of each type of resource is needed per unit produced. Purchasing an automated blood tester, for example, could reduce the amount of the technician's time needed for each blood test. Such a change would alter the slope of the constraint on the technician's time. As with the parameters that define resource constraints, we can determine how much a technology parameter can change before the current optimal solution becomes sub-optimal and how small changes in the parameter affects the value of the optimal solution.

Continuing with our example of an automated blood tester, saving one minute of technician's time per blood test increases the objective function by \$28.57. Saving another minute per test increases the objective function by a further \$32.97. Unlike a shadow price, the value of changing a technology parameter does not remain constant because the objective function is not a linear function of the technology parameters. In this example, the change in the objective function $\triangle Z = 400\left(\frac{\triangle c}{15-\triangle c}\right)$ where $\triangle c$ is the change in the amount of the technician's time. In addition to the magnitude of the change, we may also be interested in the rate of the change, $\frac{\partial Z}{\partial c} = \frac{-6000}{(c-15)^2}$ and the instantaneous rate of change at the current operating point $\frac{\partial Z}{\partial c}\mid_{c=30} = \frac{80}{3}$. The derivative at the current operating point can also be used to estimate the change in the objective function for a small change in c using a Taylor series approximation, $\triangle Z \approx \frac{\partial Z}{\partial c}|_{c=30} * \triangle c = \frac{80}{3} * 1 = 26.6$

The value of reducing the technician's time on blood tests remains positive until the time per test falls below 25 minutes. At that point, the number of tests becomes limited by constraints on the nurse's time (figure 4.3). On the other hand, increasing the technician's time per blood test does not affect the choice of the intersection of the two technology constraint lines as the best alternative until the time per test becomes infinite. The intersection of the constraint on the technician's time and the nurse's time define the optimal operating point throughout the range from 25 minutes to infinity.

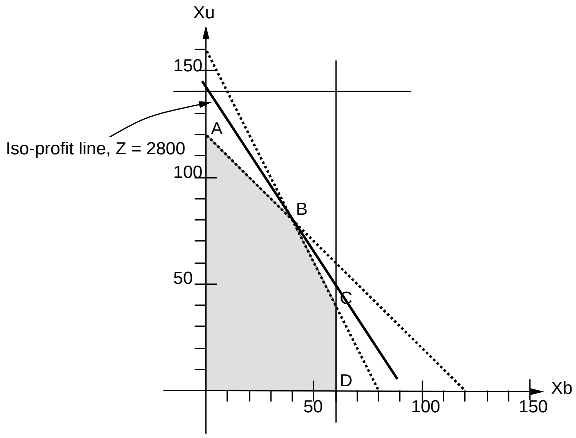

Figure 4.4: Changing the relative value of each test changes the slope of the iso-profit lines.

In addition to the parameters in the constraints, a decision is sensitive to the parameters in the objective function. These parameters give the relative value for each product. The products in our example are the two tests. Changing the relative value of each test changes the slope of the *iso-profit lines*, lines where the profit remains constant. Figure 4.4 shows the $2800 iso-profit line that intersects the efficient set at point B. If the value of blood tests increases relative to the value of urinalysis, then the slope of the iso-profit lines becomes steeper. If the value increase to the point where each blood tests are worth more than two urinalysis tests, then point C becomes the optimal solutions. The reason is that for iso-profit lines steeper than -2, the constraint on the technician's time becomes critical. Conversely, if the relative value of a blood test becomes less than the value of a urinalysis, then point A becomes the optimal solution. At this point, the constraint on the nurse's time becomes critical. For slopes in between where a blood test is worth between one and two urinalysis tests, the preferred option remain unchanged.

100% Rule In an objective function, it is the relative value of each parameter that determines the preferred option and not the absolute value. Doubling all the parameters in the objective function would double the value of each of the alternatives, but would not change the preferred alternative. Instead of looking at the absolute change needed in a parameter in order to change the preferred alternative, the relative changes in the set of parameters needs to be examined. One way of doing this is to look at the sum of the relative changes in each of the parameters. For each parameter, we calculate how much the parameter would have to increase or decrease in order to change the preferred option, while holding all other parameters constant. For example, the parameter giving the value of a urinalysis test would have to increase by 10 or decrease by 5 in order to change the preferred alternative. Increasing or decreasing the value of a blood test by 10 would change the preferred alternative. When assessing the effect of a set of changes in the parameters, divide each increase or decrease in a parameter by the amount needed to change the preferred option to give a relative change in each parameter. If the relative changes sum to less than 1, or if expressed in percentages, sum to less than 100%, then the preferred alternative remains unchanged. To give a concrete example, suppose the value of urinalysis decreased from 20 to 16.5 and the value of a blood test increased from 30 to 32. The sum of the relative changes is $\frac{3.5}{5} + \frac{2}{10} = 90\%$, which is less than 100% so the preferred alternative remains the same. If the changes sum to more than 100%, the preferred option may or may not change. This simple rule, called the 100% rule, provides a quick method for showing that the preferred alternative remains unchanged for small changes in the parameters of the objective function.

Sensitivity Analysis Summary

A full sensitivity analysis consists of determining the indifference range and the partial derivative of the objective function for each parameter. These values are then used to determine which parameters should be refined and how the alternative plans can be modified to improve performance.

The indifference range for each parameter shows how much a variable must change before the preferred alternative changes and indicates how precisely each parameter needs to be estimated. If the indifference range is relatively large, then refining an estimate will likely not change the preferred alternative and the effort put into refining the estimate is likely wasted. The end points of the indifference range are the significant values for the parameter, also called *landmark values*. When refining an estimate, the problem can be re-cast to one of asking whether a parameter is larger or smaller than these landmark values. This question may be much easier to answer than trying to generate an arbitrarily precise estimate.

The rate of change in the objective function for a change in a parameter indicates how the plan should be modified to improve performance, and the relative value of each type of improvement. Shadow prices for resources indicate how much the decision maker should be willing to pay for additional resource of each type. If a resource is available for less than its shadow price, then purchasing more of the resources will increase the objective function. Non-zero partial derivative indicate the relative importance of parameters and suggest how to improve the plan by improving technology or the methods of accomplishing the task. On the other hand, modifying a plan to make small changes in parameters with zero shadow prices (partial derivatives) will not affect the value of an alternative.

Finally, the opportunity cost of selecting a sub-optimal plan is the cost of forgoing the better alternative. When refining parameter estimates or trying to improve on the current preferred plan, the amount of effort should not exceed the value of the potential opportunity cost. If refining an estimate could at best improve performance by \$10, then at most \$10 worth of effort should be put into improving the estimate.

4.2 Dominance

The purpose of solving a decision problem is to find the best plan of action in a given circumstance, taking into account the preferences of the decision maker. A critical element in this process is the method used to show that one plan is better than, or dominates, another. The straightforward application of utility theory to determining dominance simply calculates the expected utility of each plan and selects the one with the highest expected utility. Although higher expected utility is the basis for the definition of "better" in a decision problem, using expected utility relies on having complete plans and exact parameter estimates. Other methods of showing dominance relax these requirements and can show dominance when there is uncertainty in the decision makers preferences, degree of risk aversion, or when plans are only partially elaborated.

The method used to show dominance is also important to sensitivity analysis for partial plans and for meta-level control. If ranges of expected utility are used to characterize partial plans and to determine when one plan dominates another, then the decision of which plan to choose is sensitivity only to computations that change the bounds on the expected utility. Computations that do not disambiguate overlapping ranges of expected utility will not affect the preferred plan. On the other hand, if stochastic dominance is used, determining a probability distribution of a random variable may be enough to show dominance, even though ranges of expected utility still overlap. The information needed to show dominance depends on the method used for showing dominance. Whether a decision is sensitive to a particular computation depends the information it returns and whether that information is relevant to the method of showing dominance that a planner uses.

In this section, we give an overview of methods of showing dominance. We introduce the concepts of Pareto optimal solutions and stochastic dominance and show how they can be used to prune the space of plans given only partial information about preferences. We show how problem decomposition and local dominance can be used to further prune the search space. We then show how using a limiting case, such as the value of perfect information, can also be used to limit the search space. Finally, we look at methods for comparing partial plans based on ranges of expected utility and examine the tradeoff between using more computationally expensive dominance proving methods and the resulting pruning of the search space.

4.2.1 Pareto Optimal Solutions

Most decision problems involve tradeoffs in allocating resources to accomplish desired ends. In the clinic example, the decision involves a tradeoff between the number of blood

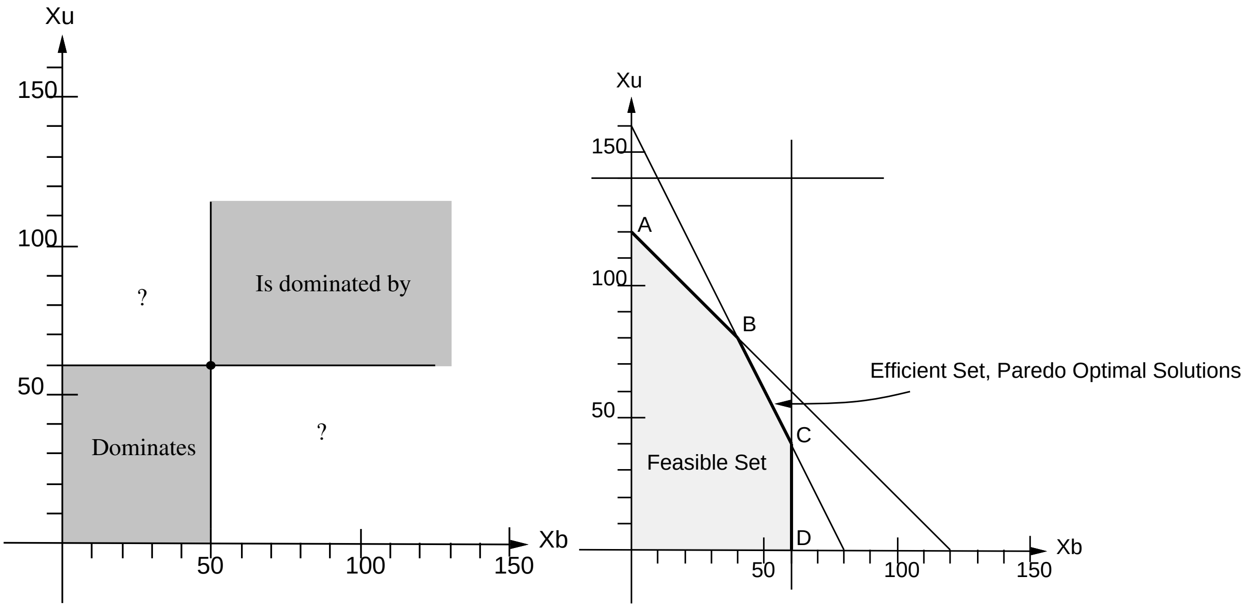

Figure 4.5: A point dominates points to the left and right and is dominated by points above and to the right.

tests and the number of urinalysis tests. Resolving this tradeoff depends on the relative value of each test. However, simply knowing that more of each test is desirable allows the search space to be pruned considerably. Any plan that produces as many, or more, of each test, using the same resources, will be at least as preferred. A point in the space of tests dominates points that produce as many or fewer of both tests (figure 4.5). In the clinic example, a feasible solution can be used to eliminate all solutions to the left and below. The only non-dominated points are those along the upper right boundary of the feasible region (figure 4.5). These points are the *Pareto optimal*[1] points since they are undominated. The set of Pareto optimal points is also called the efficient set since there points represent solutions that efficiently allocate resources to accomplish the desired ends. Other points in the space waste resources.

As we stated earlier, linear and non-linear programming algorithms achieve their efficiency by restricting their search to points in the efficient set. Pareto optimality is the principle used to eliminate other points in the feasible set.

4.2.2 Stochastic Dominance

Pareto dominance is useful in deterministic domains where the result of each plan can be directly compared. For non-deterministic domains, we need to take probability distributions into account. Consider a problem of selecting a method for getting to the airport for a 5pm flight. The available options include taking a subway, a taxi and walking. In making our

[1]The term Pareto optimal is named for the 19th century economist Vilfredo Pareto.

choice, we are concerned only with getting to the airport on time[2]. The subway takes longer on average than a taxi, but has a smaller variance. Walking is very reliable, with a very small variance, but is guaranteed to arrive after the flight has left. The important parameter in this example is the arrival time at the airport, but characterizing this variable using only its mean and variance is insufficient for determining the preferred method of transportation. The critical value is the probability of arriving before 5pm, $P(\mathsf{arrival} \leq \mathsf{5pm})$. This may or may not be the method with the earliest expected arrival time or the smallest variance. Figure 4.6 shows four examples where the preferred method can have any combination of relative expected arrival times and variances. The preferred method is always the one where the area under the curve, to the left of the 5pm cutoff, is the largest. A method *stochastically dominates* other methods when this area is larger for all deadlines.

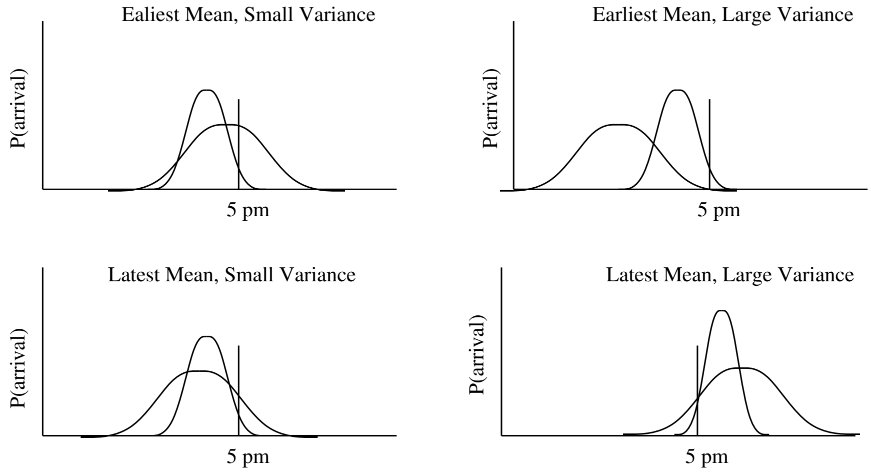

Figure 4.6: The relative mean and variance do not determine the preferred option. We show examples where each of the four possible combinations of relative mean and variance is the preferred option.

Now, suppose we modify our example to allow the plane's departure to be delayed. It may be possible to select the preferred method of transportation without knowing the length of the delay or its probability. Figure 4.7 shows the probability distribution and the cumulative probability of arrival for the taxi and the subway. The subway has a higher cumulative probability of arrival for times after 5pm and would be the preferred alternative, independent of the length of the delay or its probability. In this example, the subway stochastically dominates the taxi because its probability of success is higher for the range of possible departure times after 5pm.

Stochastic dominance is useful for showing dominance in probabilistic domains, especially where there is only partial information. It depends on the probability distribution of a variable and certain common characteristics of utility functions. In the airport example, we rely only on the preference for making the flight and not on the relative difference in value between making the flight and missing it. In the example, we use a simple form

[2]This example is based on Wellman's example of getting to the train on time in [Wellman *et al.*, 1995].

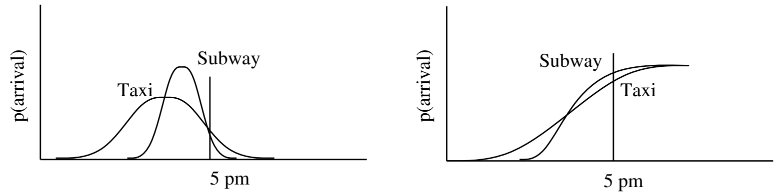

Figure 4.7: Probability of arrival and cumulative probability of arrival.

of stochastic dominance called first degree stochastic dominance. There are also second and third degree stochastic dominance methods that can be used in selecting the preferred plan. In the rest of this section, we give examples of each of the three forms of stochastic dominance and the formulas used to show dominance.

Outcome Probabilities

	1/6	1/6	1/6	1/6	1/6	1/6
F	1	4	1	4	4	4
G	3	4	3	1	1	4

Figure 4.8: The values in row 1 do not dominate the corresponding values in row 2.

Outcome Probabilities

	1/6	1/6	1/6	1/6	1/6	1/6
F	1	1	4	4	4	4
G	1	1	3	3	4	4

Figure 4.9: Reordering the columns shows first degree stochastic dominance

To illustrate the three types of stochastic dominance, we present three examples using decision matrixes[3]. In a decision matrix, the decision maker selects the row and the environment probabilistically selects the column (figure 4.8). The value at the intersection of the selected row and column is the reward that the decision maker receives. The first example, shown in figure 4.8, gives the choice between two options, each with rewards between 1 and 4, depending on the outcome. If the values in one row were equal to or larger than the corresponding values in the other row, then that row would dominate. In our first example, this is not the case, but it is still possible to show dominance. In deciding between the options, we care only about the probability of receiving each reward and not the outcome itself. This allows us to rearrange the decision matrix by increasing reward as shown in figure 4.9. The first option is now clearly the better choice, since the values in the first row are greater than or equal to the corresponding values in the second row. The columns in this example can be rearranged because each outcome has equal probability. In general, this will not be the case, but we can still make use of the underlying principle that depends only on the cumulative probability of the reward. Figure 4.10 shows the cumulative probability for

[3]These examples are taken from [Fishburn and Vickson, 1978].

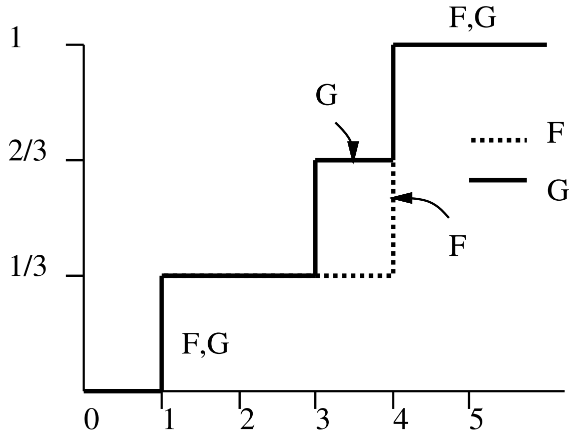

Figure 4.10: The cumulative probability for F is less than that for G, showing first degree stochastic dominance of F over G.

each row in our example. Row F dominates because its cumulative probability is always less than or equal to that for the second row. Whenever $G(x) \geq F(x)$, and the utility function increases with increasing reward, we can conclude that plan F dominates plan G.

Outcome Probabilities

	1/6	1/6	1/6	1/6	1/6	1/6
F	1	1	4	4	4	4
G	0	2	3	3	4	4

Figure 4.11: In this example, reordering the rows does not show dominance.

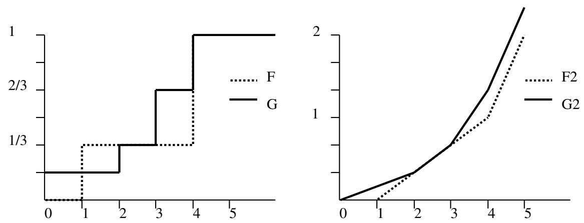

Figure 4.12: Second degree stochastic dominance.

An example of second degree stochastic dominance is shown in figure 4.9. In this example, neither row dominates the other even if the columns are rearranged. To show dominance in this example, we need to make the assumption that the decision maker is risk averse. Given a choice between a sure thing and a gamble with the same expected value, the decision maker would always take the sure thing. Looking at only the first two columns of the decision matrix, we see that the decision maker has a choice between two options, both with an expected value of 1. The difference is that option F has a guaranteed reward of 1 whereas option G involves a gamble. A risk averse decision maker would thus prefer option F if only the first two outcomes were possible. Looking at the remaining columns, we see that option F is preferred for these outcomes as well. The general principle used to show dominance in this example is illustrated in figure 4.12. The cumulative distributions do not show dominance, $G(x) \not\geq F(x)$, but the integrals of the cumulative distributions satisfy the relationship $\int G(x)dx \geq \int F(x)dx$, which is the definition of second degree stochastic dominance.

Outcome Probabilities

	1/4	1/4	1/4	1/4
F	13	11	11	11
G	10	12	12	12

Figure 4.13:

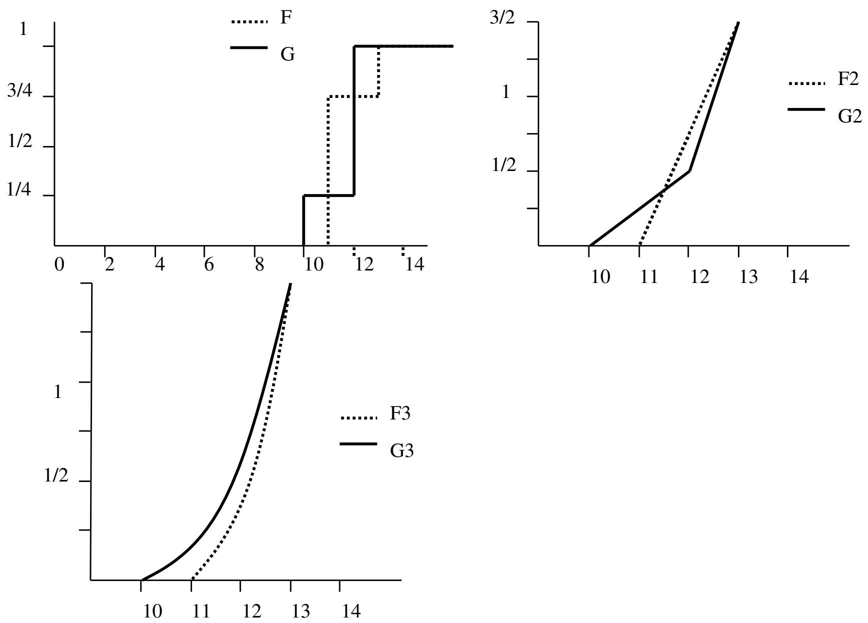

Figure 4.14:

The third form of stochastic dominance applies when the decision maker is risk averse but becomes less risk averse as his level of wealth increases. This corresponds to a utility function where a poor person is not willing to risk \$1000 but a millionaire would be. Technically, such a utility function has a negative second derivative and a positive third derivative. Figure 4.13 shows an example where option F dominates option G using third degree stochastic dominance. As shown in the accompanying graph, figure 4.14, first and second degree stochastic dominance do not suffice to show dominance in this example. Third degree stochastic dominance depends on the relationship between the second integrals, $\int\int G(x)dxdx \geq \int\int F(x)dxdx$.

Higher degrees of stochastic dominance can be defined, but are generally not used because they place additional, hard to justify, constraints on the form of the utility function.

For a complete discussion of stochastic dominance and related theorems, see [Whitmore and Findlay, 1978].

4.2.3 Problem Decomposition and Local Dominance

Divide and conquer is one of the basic strategies for creating efficient algorithms. Efficiency is achieved by dividing a problem into smaller sub-problems, solving them separately and then gluing the results together to solve the entire problem. This strategy works if the solutions to the sub-problems are easy to glue together and the result is a valid solution to the entire problem. In planning, it is natural to break a task down into sub-tasks and plan for each one independently. The result can be glued together as long as the sub-plans do not interact badly, for example by clobbering the initial conditions that other sub-plans depend on.

In decision-theoretic planning, the same types of sub-plan interactions can prevent gluing together sub-plans to form a feasible solution. In addition, decision-theoretic planners need to consider the expected utility of the resulting plans and whether using sub-plans with the highest expected utility leads to a plan with the highest expected utility. In some domains, gluing together locally dominant solutions creates a globally dominant solutions. Consider the task of visiting Ottawa and then Washington, with the objective of traveling as cheaply as possible. Finding the cheapest mode of transportation for each leg of the trip results in the cheapest overall plan. However, if we modify the problem to include a preference for taking less time, the result does not necessarily hold. The reason is that each sub-problem involves a tradeoff between time and money and the best global solution may not be composed of the locally optimal solution. This occurs, for example, when there is a soft deadline with a decreasing utility for plans that miss the deadline.

Even when the best local tradeoff does not lead to the best global tradeoff, we can still use some forms of dominance to limit the sub-plans that need to be considered. In our transportation example of visiting Ottawa and Washington, we may have a choice of driving, taking a bus, or flying. The best method for getting to Ottawa may not be part of the best plan for visiting both. Package deals, like three-day bus passes and weekly rates for rental cars may make it better to use the same method of transportation for both legs of the trip even when another method would be better if the problem was only to get to Ottawa. However, within each mode of transportation we may be able to eliminate some plans. When driving, for instance, we may have a choice of routes. If there are two routes that differ only in length, then we can reject the longer route. If, however, the routes differ in length and cost, because the shorter one has a toll and the longer one does not, we may not be able to decide between them without considering the rest of the task. On the other hand, if one route cost more and is longer, we can reject it. Wellman uses local dominace to prunce the space of plans in a path planning problem with time-dependent uncertainty [Wellman *et al.*, 1995].

4.2.4 Limiting Solutions

To this point, we have discussed methods for showing that one plan dominates another. In this section, we look at methods for comparing a plan with all possible plans. If a planner can show that a plan dominates all other plans or nearly dominates all other plans, then it can adopt the plan without having to generate and evaluate any more plans. There are two approaches to evaluating a plan relative to all other plans. The first approach involves using a limiting solution or a theoretically best possible plan. The second approach requires the planner to search the space of plans systematically.

A best case limiting plan is generated using the properties of the domain, ignoring some constraints that may make a plan infeasible. For example, we could ignore all sub-goal interactions and create a plan that used the locally optimal solution to each sub-task. In machine shop scheduling problems, this corresponds to assuming that all unordered activities can take place in parallel, ignoring constraints on machine availability. In multi-objective problems, such as visiting Ottawa and Washington, while minimizing time and money, we may simply ignore the tradeoff. A lower bound on the time is the sum of the shortest travel times for each leg of the journey and a lower bound on the cost is the sum of the cheapest methods for each leg of the journey. The resulting bound on utility could be obtained only if the cheapest method was also the fastest, which is unlikely. However, the bound can be used to calculate a limit on the opportunity cost by comparing the currently preferred plan with this limiting solution. The result could then be used to decide if further computation to improve the plan is warranted.

Another method for generating a limiting solution makes use of the value of perfect information. In some domains, the preferred plan may depend on some unknown state of the world, such as whether a patient has a particular disease. Tests may be available to indicate whether the disease is present or not, but these tests may be imperfect, returning false negatives and false positives on occasion. To improve reliability, a plan may include multiple tests or repeat the same test multiple times. The planner can limit the combinations of tests by considering only combinations of tests that cost less than the value of perfect information. Suppose there were a test that fave perfection information about the presence of the disease. The price of perfect information is the maximum price that coupd be paid for the test and still have a pla with the test have the highest expected utility. To calculate the value of perfect information, we calculate the increase in expected utility from knowing the value of a variable, such as the presence of a disease, for each possible value of the variable. We then weight each increase by the probability that the variable has that value to get the value of information. For example, suppose the treatment for a disease cost \$100 per patient and patients have the disease with probability 1/2. Knowing that a patient did not have the disease can be used to save \$100, the cost of the treatment. Since the probability that a patient does not have the disease is 50%, the expected value of knowing whether a patient has the disease is \$50. Since any combination of tests can at best provide perfect information, a combination of tests that exceeds the cost of perfect information (\$50 in this case), can be pruned because it cannot possibly be better than just treating everyone. Using the value of perfect information reduces the problem from searching a infinite space of combinations of tests to searching the limited space where combinations of tests cost less than a threshold.

In addition to limiting the potentially opportunity cost by using a limiting solution, a planner can limit the quality of ungenerated plans by systematically searching the plan space. For example, suppose each action takes a minimum amount of time t and the planner generates plans from shortest to longest. If the planner finds a valid plan that takes T seconds, then it needs to consider only plans less than T/t actions long. Once the planner has generated a plan longer than T/t, it can terminate and guarantee that it has a plan that takes the least amount of time.

4.2.5 Comparing Partial Plans

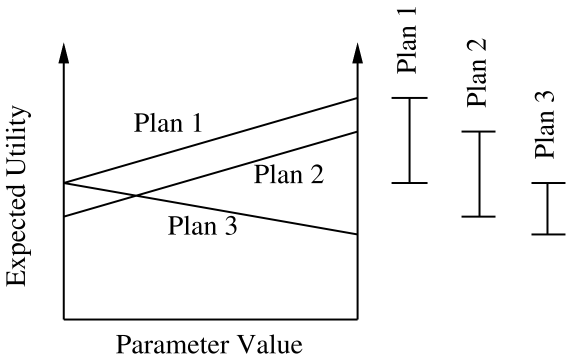

Figure 4.15: Ranges of expected utility can overlap even when one plan clearly dominates another.

In concluding our discussion of methods for showing dominance, we need to consider the computational costs and related tradeoffs. As an example, we will consider the comparison of three partial plans, shown in figure 4.15, where the expected utility depends on a partially refined parameter. From the graph, it is obvious that plan 1 dominates the other plans. For any particular value of the parameter, plan 1 has an expected value at least as high as any of the other plans. To show dominance, a planner would take the difference between the expected utility of pairs of plans and evaluate the difference over the allowed range for the parameter. If the difference in expected utility, $EU(\mathsf{plan1}) - EU(\mathsf{plan2})$ was always non-negative, then plan 1 would dominate plan2. An alternative approach would be to evaluate and compare the range of expected utility for each of the plans, as shown on the right of figure 4.15. In this example, the ranges show that plan 1 dominates plan 3 but do not show that plan 1 dominates plan 2. The tradeoff in this example involves the relative costs for calculating each method of dominance and the resulting pruning of the search space. In some planners, finding the bounds on expected utility might be significantly less time consuming than subtracting the value of one plan from another and evaluating the result over a range. This is especially true if the function for evaluating a plan is not available symbolically. Using ranges of expected utility makes dominance calculations

Blood Tests	Urinalysis
0	120
40	70
44	29
60	25
60	0

Table 4.1: Feasible combinations of tests.

more efficient, but may require more plan refinement before the dominant plan can be identified.

4.3 Sensitivity Analysis for Partial Plans

At the beginning of this chapter, we examined sensitivity analysis for a complete decision problem with exact estimates for each parameter. In this section, we relax these conditions to include partial plans and parameter estimates that include ranges of values. We also include problems with discrete choices, such as deciding whether to drive or fly to Ottawa, in addition to the continuous choice allowed in selecting the number of tests in the clinic example. To put our approach on a firm foundation, we base our sensitivity analysis on methods developed for Bayesian decision theory by Insua [Insua, 1990]. We adapt the techniques from these methods for use with partial plans and estimates. We then examine measures of sensitivity, including measures that account for more than one parameter varying at a time.

Much of the original work in sensitivity analysis originated in the fields of economics and operations research. Continuing work in these fields has produced a well founded formulation of sensitivity analysis for Bayesian based decision theory. In particular, Insua's dissertation specifies a method for doing sensitivity analysis where preferences are modeled by families of utility functions and probabilities are modeled by belief functions [Insua, 1990] [Insua and French, 1991]. These sensitivity analysis methods developed for Bayesian decision theory are applicable for performing sensitivity analysis for partial plans since they allow ranges of values for parameters, including probabilities. In partial plans, these ranges arise because of abstraction, incompleteness and limited parameter estimation. As with sensitivity analysis for a complete plan, sensitivity analysis for partial plans can identify the set of non-dominated, potentially optimal plans and the ranges of indifference for each parameter. The potential opportunity cost can also be used to limit the effort spent trying to determine the plan with the highest expected utility. The basic outline of the method is given in table 4.2. This method includes specifications of the required algorithms for linear problems and suggests methods for solving non-linear problems. Although the complete method is generally too computationally expensive for use in meta-level control, it is useful as a framework on which to base our implementations of sensitivity analysis.

The algorithm assumes that strict ranges are given for each parameter as well as a most

1. Represent the problem as a decision problem.
2. Let $a_i, i = 1 \dots n$ be the set of possible alternatives.
3. Let w be a vector of parameters of the decision problem with uncertain values.
4. Let w_* be the current best estimate of the parameter values and a_* be the current preferred alternative.
5. Let $S = \{w\}$ the set of feasible parameter values.
6. Let $EU_i(w)$ be the expected utility of alternative i with parameters w.
7. Identify the set of non-dominated alternatives, which are the alternatives for which there is no other alternative that is always as good or better. The set of alternatives, a_i, where $\forall j \mid \min_{w \in S}(EU_i(w) - EU_j(w)) \leq 0$
8. Identify the set of potentially optimal (and non-dominated) alternatives, which are alternatives that for some parameter value are at least as good as all other alternatives. The set of alternatives, a_i, where $\forall j \neq i \min_{w \in S}(z_i \mid EU_i(w) - EU_j(w) + z_i \geq 0) z_i \leq 0$
9. Identify the set of adjacent potentially optimal (and non-dominated) alternatives. a_j is adjacent potentially optimal to $a*$ iff $S_j \cap S_* \neq \emptyset$

   Where S_j is the set of parameter values for which a_j is optimal.
10. Select a distance metric $d(w_i, wj)$ that gives some indication of relative difference in the two parameter vectors.
11. Find $\rho = \min_{w \in S} d(w, w_*) \ s.t. \ EU_j(w) - EU_*(w) = 0$ and let $\hat{w} \in S \ s.t. \ \rho = d(\hat{w}, w_*)$.

    ρ is an absolute measure of sensitivity. ρ is the radius of the largest sphere centered on w_* for which a_* is guaranteed to be optimal.
12. Find $\delta = \max_{w \in S} d(w, w_*)$ and let $\check{w} \in S \ s.t. \ \delta = d(\check{w}, w_*)$ δ is the radius of the smallest sphere centered on w_* which includes the feasible set of parameter values S.
13. Calculate $r = \rho/\delta$, a relative measure of sensitivity. This is a surrogate for the sensitivity measure $t = Volume(S_*)/Volume(S)$ where S_* is the region of S where a_* is optimal.

    Note that :

    - $\rho = 1 \Rightarrow completely - insensitive$
    - $\rho = 0 \Rightarrow completely - sensitive$

Table 4.2: Summary of sensitivity analysis for partial plans

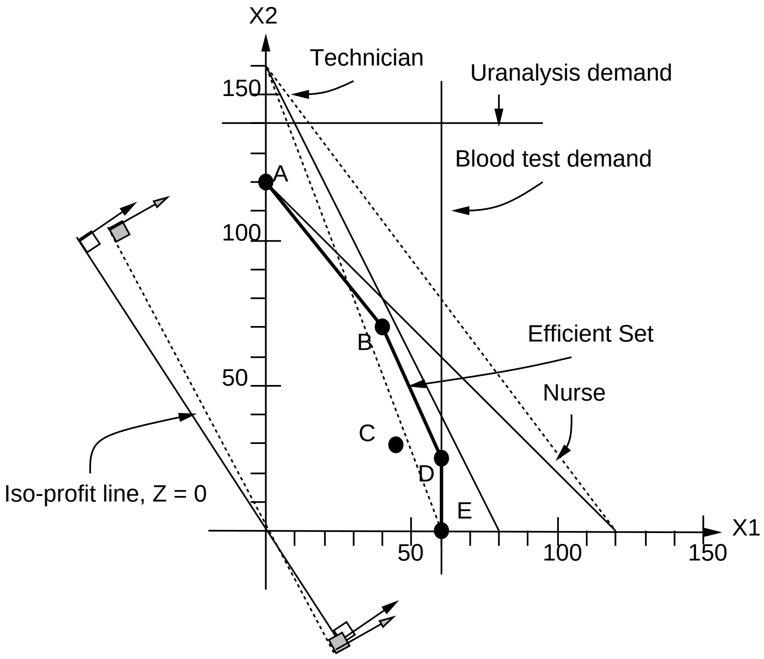

Figure 4.16: Graphical representation of the discrete decision problem.

likely value for each parameter. In some cases, the most likely value of a parameter within a range may not be known. In such cases, the analysis can simply use the middle of the range or do a parameterized analysis when the most likely value is allowed to vary over the entire range. A better approach is to use the value consistent with the choice the decision maker would make if forced to choose without further refinement of the parameters or the plans. Alternatively, the expected value of a parameter and its variance may be available, but not the variable's range. In such cases, the variance could be used to give approximate bounds. We will discuss the methods for dealing with missing information about parameters in the detailed discussion of the sensitivity analysis algorithm.

In the succeeding sections, we describe in detail each step in the sensitivity analysis and its application to sensitivity analysis for partial plans. We use a modified version of the medical clinic example to illustrate each step in the process. In the modified problem, we consider additional constraints that limit the choice of how many tests of each type to perform to a set of discrete choices. These constraints arise from laboratory processing equipment that comes in discrete sizes. For our example, suppose that combinations of equipment are available to process tests in the specific combinations shown in table 4.1. Plans corresponding to each combination of equipment are shown in figure 4.16 as points labeled A, B, C, D, and E, all within the feasible region for the original problem. In our discussion of the methods for sensitivity analysis, we limit our choice to these five competing plans. In addition, we introduce some uncertainty into the parameters associated the technician who is to be hired to perform the tests. We suppose that the exact time needed by the technician to perform a blood test is known only to be between 20 and 40 minutes, with 30 minutes being the most likely. The reason for the uncertainty is that not all of the steps in the process have been planned in detail. We also suppose that the cost of the technician ranges from \$20 - \$25 per hour. The technician's exact rate of pay will depend on market conditions and the qualifications of the person hired. The most likely value, \$22.50 / hour is taken from a recent market survey. The uncertainty in these two parameters are show as ranges in the slopes of the technician's constraint line and the iso-profit line in figure 4.16.

4.3.1 Sensitivity Analysis Procedure

Step 1: Represent the problem as a decision problem.

Decision-theoretic planners cast the planning problem as a decision problem where the effort is focused on creating the set of alternative plans, estimating parameter values, calculating expected utility and selecting the plan with the highest expected utility. To solve the decision problem efficiently, the planner may not produce a full set of completely elaborated plans and may have only partial estimates of parameter values. The result is a partial set of partially elaborated plans with ranges of parameter values.

Step 2: Let $a_i, i = 1 \ldots n$ be the set of possible alternatives.

We represent the set of alternatives as a discrete set of plans. In our example, these correspond to the five distinct combinations of tests shown in figure 4.16. The original continuous problem can be recast as a discrete problem by considering only the points where constraints intersect. For linear problems, this approach works because at least one of these points must be in the set of optimal solutions. For non-linear problems, we can parameterize a set of discrete alternatives. For example, if the objective function in the clinic example were a non-linear, monotonically non-decreasing function of the number of each test, the set of non-dominated alternatives would correspond to the three line segments in the efficient set of figure 4.16. The alternative plans would correspond to points on these lines. For example, $\mathsf{plan1} = (120 - 40t \text{ } \mathsf{urinalysis}, 40t \text{ } \mathsf{bloodtests})$ where $0 \leq t \leq 1$. The range in the value of the parameter t can be treated in the same as the range in other parameters values. If needed, the value of t could be restricted by further computation to determine which values of t produce the plan with higher expected utility.

Step 3: Let w be a vector of parameters.

We represent each parameter w_i in the decision problem as a triple $[w_{i_l}, w_{i_b}, w_{i_h}]$ that gives the lower bound, the best estimate and the upper bound respectively. For example, the technician's pay is represented by the triple $[20, 22.5, 25]$. In some cases, we may not have strict upper and lower bounds on the parameter values. In these cases, the sensitivity analysis could use approximate bounds, such as a fixed number of standard deviations above and below the best estimate. Conversely, if the bounds are known, but not a best estimate, the sensitivity analysis can proceed using a value for the best estimate that is consistent with the choice the decision maker would make if forced to choose, without further refinement of parameter estimates or plans. This may be simply the decision that would result from selecting the middle of the range for each parameter or be based on more global strategies like maximum-minimum, maximum-maximum or minimum regret. We examine these criteria for making decisions in the next section. Finally, when the bounds or the best estimate of a parameter are approximations, the results of the sensitivity analysis must be interpreted in light of these approximations.

In our example, if the recent salary survey gave only the range of pay for a technician, the average pay would have to be estimated. Similarly, if the mean and a standard deviation were given, the bounds could be approximated using two standard deviations above and below the mean. If we assume a Gaussian distribution, there is a 95% probability that the value is within this range.

Step 4: Let w_* be the current best parameter estimate.

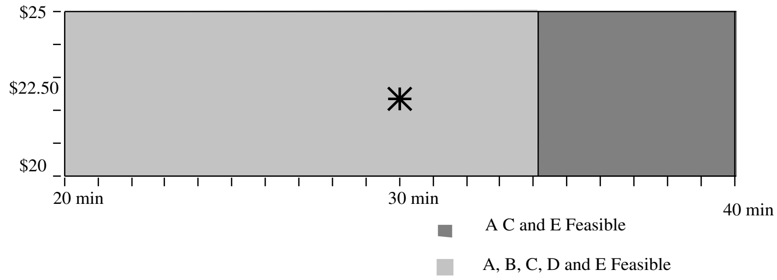

Figure 4.17: Feasible Region

Using the current best estimate for each parameter w_i, we get a point in the parameter space w_* (figure 4.17). The currently preferred alternative is the one that has the highest expected utility at this point in the parameter space.

Ranges of parameter values may be available without corresponding best estimates. Some planning methods allow parameters, such as the time needed for the technician to perform a blood test to be bounded but do not provide any information on the expected value. In such cases, we wish to select w_* in a way that is consistent with the plan that the planner would select if forced to make a decision at this point without further planning. In a decision-theoretic planner, the planner selects a plan with the highest expected utility. However, with only ranges of parameters, each alternative may have a range of expected utility and there may be no clearly best plan. Figure 4.18 shows an example where plan 1, 2 and 3 are all potentially the best plan, whereas plan 4 is clearly not.

In cases of overlapping ranges of expected utility, the planner could use a simple heuristic to decide between plans if it were decided that determining which one was better was not worth the expected computational cost. Heuristics that could be used include the maximum-minimum rule, the maximum-maximum rule and the minimum regret rule. The maximum-minimum rule selects the plan with the best guarantee on expected utility,

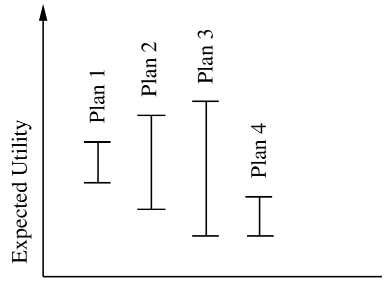

Figure 4.18: Ranges of parameter values give ranges of expected utility. In this example, plan 4 is clearly dominated by plan 1, but the choice between plan 1 2 and 3 is unclear.

which is the plan with the greatest lower bound (plan 1 in figure 4.19). The maximum-maximum rules selects the plan with the highest upper bound on expected utility (plan 3 in figure 4.19). The minimum regret rule selects the plan with the lowest possible opportunity cost. Figure 4.19A shows how the expected utility of three alternatives might vary with the value of a parameter. The opportunity cost for each alternative is the difference between its expected utility and the expected utility of the best plan. Figure 4.19B shows the opportunity cost for each plan as a function of the parameter value. The minimum regret rule selects the plan with the smallest upper bound on the opportunity cost, in this case plan 2. The idea is that no matter what the true value of the parameter, the decision maker minimizes how much they regret not having picked the better alternative. Note that this plan has neither the greatest upper bound or lower bound on expected utility.

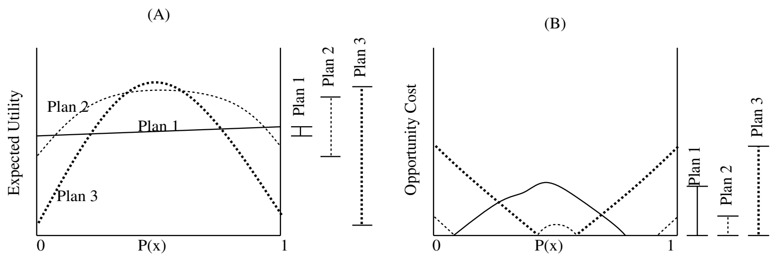

Figure 4.19: The value of each alternative varies with P(x), with each one being optimal for some values (A). The opportunity cost for each one is the difference between its expected utility and the expected utility of the best alternative for each values of P(x).

When there is a best estimate of the parameter values, w_*, the sensitivity analysis determines how much this point can vary without changing the preferred alternative. When there is no best estimate, the currently preferred alternative, selected using one of the heuristics described in the previous paragraph, is based on the ranges of parameter values. A sensitivity analysis needs to determine how much the range can change before changing the preferred alternative.

Consider again the example in figure 4.19. Restricting the range on probability reduces the range of the expected utility and the range of the opportunity cost. If the ranges are restricted as shown in figure 4.20, plan 2 and plan 3 become equally preferred in terms of

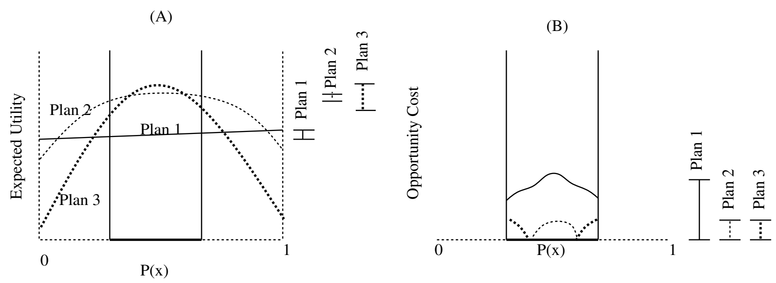

Figure 4.20: Reducing the range of probability changes the preferred plan.

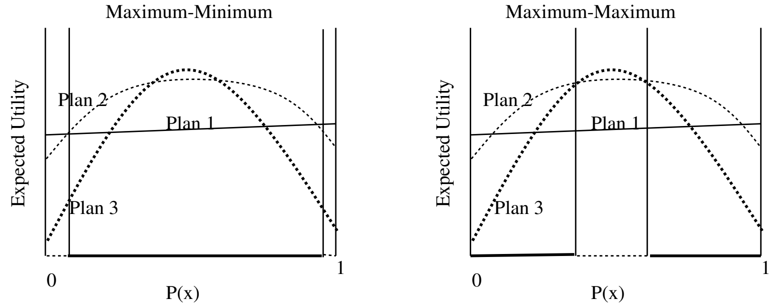

Figure 4.21: Different range restrictions are needed to change the preferred plan for each plan selection method.

the maximum regret. Restricting both bounds beyond this point results in a preference for plan 3 since it will have the lowest bound on opportunity cost. Although the preferred plan changes when the range is restricted in this way, the plans still have overlapping ranges of expected utility. If one of the other two methods were used to select the preferred plan, the selected plan would remain unchanged. Figure 4.21 shows the changes needed in the bounds to change the preferred plan when the maximum-minimum and maximum-maximum rules are used. In doing a sensitivity analysis, the method for selecting between potentially optimal partial plans needs to be used to determine the range of parameter values that would leave the preferred alternative unchanged.

Step 5: Let $S = \{w\}$ the set of feasible parameter values.

Each of the parameters can be considered an axis that defines a real valued n-space. The bounds on the parameter values form an n-cube within this space. However, all the points within this n-cube are not necessarily feasible. Certain combinations of parameter values may be infeasible because of other constraints. In our example, the market would allow up to 60 blood tests and up to 140 urinalysis tests, but producing combinations of tests outside the shaded region in figure 4.16 is infeasible because of other constraints. In performing the sensitivity analysis, we need consider only points in the parameter space that represent feasible solutions. We also need to consider the feasible space for individual alternatives. In the clinic example, if the time needed by the technician per blood test rises above

34.3 minutes, alternatives B and D become infeasible. Figure 4.17 shows graphically how this affects the parameter space for the clinic example.

Step 6: Let $EU_i(w)$ be the evaluation of alternative i with parameters w.

We evaluate each alternative to give its value as a function of the parameters. In general, each alternative will have a range of values that depend on the values of the parameters. The function mapping parameter values to expected utility is used to determine the value of an alternative at a specific point in the parameter space.

Step 7: Identify the set of non-dominated alternatives.

A dominated alternative, by definition, will never be the preferred alternative no matter how the range of parameters changes. Dominated alternatives can thus be eliminated from the sensitivity analysis.

To identify the non-dominated alternatives, we look at the difference in expected utility for pairs of plans over the allowed range of parameter values.

Let

$$\mathsf{minDiff} = \min_{w \in S} EU_j(w) - EU_k(w)$$

Then a_j dominates a_k if $\mathsf{minDiff} \geq 0$. We do this comparison for each pair of alternatives to find the set of non-dominated alternatives. This algorithm is $O(n^2)$ where n is the number of alternatives. Taking the difference relies on having the expected utility functions for each alternative in symbolic form. Alternatively, if the functions are known to be linear, being able to evaluate them at the boundaries of the parameter space is sufficient.

In the modified clinic example, the non-dominated alternatives are the points A, B, C and D in figure 4.16. Point E is dominated by point D, since point D results in as many blood tests and more urinalysis tests. Taking the difference between these two alternatives shows that $EU_D(w) \geq EU_E(w)$ for all allowed values of the parameters.

Any of the methods for showing dominance that we examined in section 4.2 are also applicable for showing dominance when doing a sensitivity analysis. In the clinic example, we could also eliminate option E because it is not Pareto optimal. In probabilistic domains, we could use stochastic dominance.

Step 8: Identify the set of potentially optimal alternatives.

Even though an alternative is not dominated, it still may never be the preferred alternative. Consider point C in figure 4.16. None of the other alternatives dominate alternative C. However, if the set of feasible solutions remains unchanged, then for any relative positive weighting of blood test and urinalysis tests, either alternative B or alternative D will be preferred to alternative C. When C is preferred to D, B is preferred to C and when C is preferred to B, D is preferred to C. Alternative C can be removed from the set of alternatives considered in the sensitivity analysis because it will never be the preferred

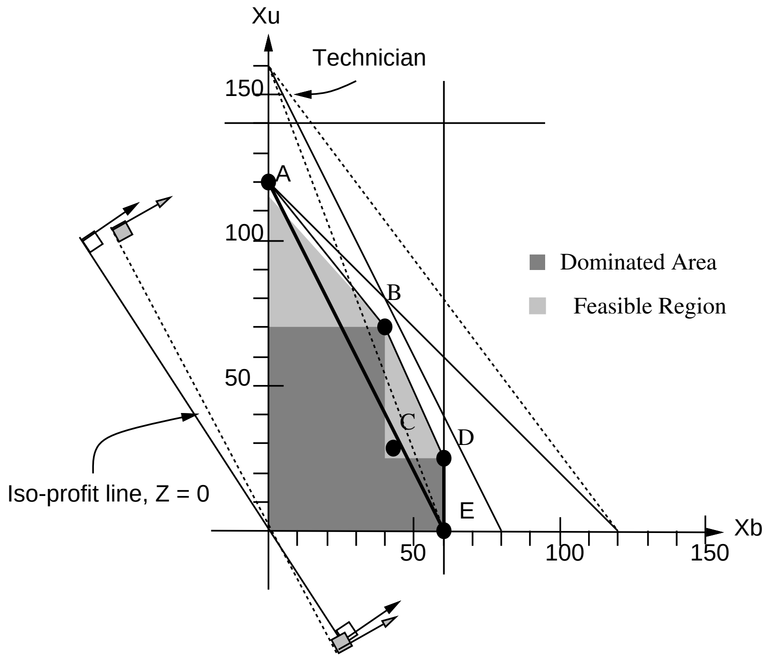

Figure 4.22: Potentially optimal test combinations.

alternative. However, in this particular instance, we cannot use alternatives B and D to eliminate alternative C because the uncertainty in the technician constraint may eliminate alternatives B and D from the set of feasible solutions. Even if these alternatives are eliminated, alternative C is still never potentially optimal, because either alternative A or alternative B will always be preferred to alternative C. Figure 4.22 shows why graphically. Point C is below the line that connects point A and point B. If the slope of the iso-profit lines is steeper than this line, option E is preferred. Similarly, if the line is less steep, option A is preferred. If the two lines are parallel, then both A and E are both preferred to C. Alternative C is never the preferred alternative and can be eliminated from consideration.

More formally, we need to identify the set of potentially optimal alternatives in the set of non-dominated alternatives. The algorithm for doing this $O(n^2)$ where n is the number of non-dominated alternatives.

Let

$$Z_j = \min_{w \in S} z \; s.t. \forall k \neq j \; EU_j(w) - EU_k(w) + z \geq 0$$

Then a_j is potentially optimal if $Z_j \leq 0$. Informally restating this formula, an alternative, j, is potentially optimal if there is a point in the parameter space where the value of the alternative is at least as high as all other alternatives, k.

It is important to note that, although step 8 may seem to subsume step 7, it does not. If figure 4.22, alternative E is potentially optimal, because it is optimal when the profit depends only on the number of blood tests, Xb. We can discard alternative E because it is dominated by alternative D, not because it is not potentially optimal.

Step 9: Identify the set of adjacent potentially optimal alternatives.

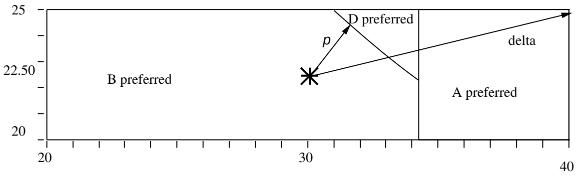

Figure 4.23: The Potentially optimal plan changes over the parameter region.

The objective of a sensitivity analysis is to identify the smallest changes in the current parameter estimates that change the preferred solution. The currently preferred alternative will be preferred over some region of the parameter space adjacent to the current best estimate of the parameter values. To determine the extent of this region, we want to identify the preferred alternatives in adjacent regions. This is done by looking for the intersection of these regions where the competing alternatives will have the same value.

Formally, a_j is adjacent potentially optimal to $a*$ iff $S_j \cap S_* \neq \emptyset$, where S_j is the set of parameter values for which a_j is optimal and S_* is the set of parameter values where the currently preferred alternative is optimal.

Figure 4.23 shows the regions of the search space where alternatives B, A and D are preferred. Since B is the currently preferred alternative, alternatives that are preferred in regions that border the region where B is preferred are adjacent potentially optimal alternatives.

Step 10: Select a distance metric.

A sensitivity analysis measures the required change in parameter values needed to change the preferred alternative. In a single attribute sensitivity analysis, this change can be expressed as a distance along a single axis that the current best estimate w_* would have to move to change the preferred alternative. In multi-attribute sensitivity analysis, finding the shortest distance to move w_* to change the preferred alternative involves comparing multi-attribute vectors in the n-dimensional parameter space. In our clinic example, this might involve comparing a 3 minute per blood test increase in technician time with a $2 increase in the technician's hourly wage. To make this comparison, we need to select a distance metric $d(w_i, w_j)$ that gives some indication of relative difference in the two parameter vectors, w_i and w_j. Possible functions include a weighted Manhattan distance, a weighted Tchebycheff distance and weighted Euclidean distance.

For meta-level planning control, the distance measure should indicate the cost of computation for refining parameter values. A weighted Manhattan distance can be used if each parameter is refined by a different computation. If a single computation refines all parameters at once, then a weighted Tchebycheff distance would be appropriate. Mixed distance measures may also be appropriate depending on the expected effect of performing computation. The appropriate distance measure to use will depend on the domain and the characteristics of the planner. In our clinic example, if determining inforamtion about the technicial reduces both parameter ranges, then a weighted euclidian distance is best. The weighting reflects the relative change in each parameter for a given amount of refinement.

Step 11: Absolute measure of sensitivity.

An absolute sensitivity measure gives the distance in the parameter space from the current best estimate w_* to the closest point when another alternative becomes preferred.

Let $\rho = \min_{w \in S} d(w, w_*)\ s.t.\ EU_j(w) - EU_*(w) = 0$

ρ is an absolute measure of sensitivity. It is the radius of the largest sphere centered on w_* for which a_* is guaranteed to be optimal.

For the clinic example, if we take a simple weighted Euclidean distance measure where each of the weights is one, the closest point on the boundary to the current best estimate $w_* = (30, 22.5)$ is (31.68, 24.39) and $\rho = 2.53$. ρ is shown graphically in figure 4.23.

In addition to indicating the sensitivity, the vector ρ indicates the relative importance of refining each parameter estimate. From the graph in figure 4.23, we see that just refining the rate of pay for the technician would not change the preferred alternative. Changing the time per blood test would, but the amount of change needed can be reduced by increasing the rate of pay. The vector $\rho = (1.68, 1.89)$ gives the relative change needed in each parameter to reach the boundary with the least amount of change. If the distance function reflects the relative cost of refining the best estimate for each parameter, then the vector suggests the relative effort that should be put into refining each estimate.

Step 12: Relative measure of sensitivity.

Relative sensitivity measures give information on the proportion of the parameter space for which the current alternative is preferred.

Let $\delta = \max_{w \in S} d(w, w_*)$. δ is the radius of the smallest sphere centered on w_* which includes the feasible set of parameter values S.

The ratio, $r = \rho/\delta$, is a relative measure of sensitivity. r gives the ratio of the distance to the next preferred alternative to the total distance the parameter estimates could move. This is a surrogate for the sensitivity measure $t = Volume(S_*)/Volume(S)$ where S_* is the region in S where a_* is optimal and S is the entire feasible parameter space.

Note that :

- $\rho = 1 \Rightarrow completely - insensitive$
- $\rho = 0 \Rightarrow completely - sensitive$

In our example, $\delta = 10.3$ and $r = 2.53/10.3 = .245$, so the choice is relatively sensitive. This makes intuitive sense. Looking at figure 4.23, we see that the region where B is preferred covers most of the parameter space, but the current best estimate is relatively close to the boundary between two regions.

4.4 Approximate Sensitivity Analysis

Many of the planning situations where meta-level control could be applied tend to be complicated and non-linear. A complete sensitivity analysis for large, non-linear problems is computationally expensive. Fortunately, the information needed for meta-level control does not have to be accurate to ten decimal places. For meta-level control, a computationally less expensive approximation to a full sensitivity analysis is more appropriate.

One method of approximating a solution to the non-linear problem is to replace each non-linear constraint with its linear approximation and to solve the resulting linearized problem. This method can provide reasonable approximations, especially when the non-linear constraints are almost linear. If more accuracy is required, iteratively linearizing and solving the linearized problem can be used to refine a solution.

Qualitative reasoning provides a second approach to approximating a sensitivity analysis. Comparative analysis techniques give the qualitative response of a system to qualitative changes in system parameters values [Weld, 1987]. These techniques could be combined with order of magnitude reasoning to determine which effects are significant [Raiman, 1991].

Chapter 5

Planner Descriptions

In chapter three, we outlined the basic questions related to meta-level control and considered approaches for answering each of them. In this chapter, we present a set of four decision-theoretic planners that shall be used in the rest of this dissertation to provide concrete examples for examining each of the meta-level control questions in more detail. These four planners span much of the range of decision-theoretic planners and, in applying our meta-level control methods to them, we show how to make our methods operational. The resulting implementations also allow us to evaluate our approach empirically.

We begin this chapter by discussing some of the dimensions along which decision-theoretic planners can be characterized. These dimensions include the characteristics used to describe artificial intelligence planners such as whether a planner searches in a space of plans or in a world space, whether it uses abstraction and whether it is domain independent. Decision-theoretic planners can also be characterized by the degree to which the planner allows utility functions and probabilities, the components of decision theory. We discuss the tradeoffs that each dimension entails and the relevance of each planner characteristic to particular domains. We then introduce the four planners and summarize their characteristics along each dimension. This summary serves as an introduction to the four planners and shows how they span the range of decision-theoretic planners created to date. We then examine each planner in detail. The first of the four planner is the Xavier route planner; a special purpose planner used to create routes for the Xavier robot in an office environment. The Pyrrhus planner is a domain-independent partial-order, generative planner, used for resource optimization planning problems. The DRIPS planner is a domain-independent hierarchical refinement planner that has been used to create medical treatment policies. Finally, the two-op planner for a simplified robot-courier is an example of an anytime planning algorithm that finds increasingly better solutions, given more computation. Each planner is discussed in detail to highlight the relevant factors affecting each of the meta-level control questions addressed in the subsequent chapters. We conclude with a summary of the meta-level control questions relevant for each planner. This summary serves as an outline for the subsequent chapters.

5.1 Classification of Decision-Theoretic Planners

The potential benefits of combining decision theory with artificial intelligence planning techniques have long been recognized in the planning community. In their seminal 1977 paper on the topic, Feldman and Sproull outlined many of the uses for decision theory in planning [Feldman and Sproull, 1977]. The one decision-theoretic idea that have been applied to planners based on traditional artificial intelligence techniques is the use of probability. Planners such as Buridan [Kushmerick *et al.*, 1994], ε-safe planners [Goldman and Boddy, 1994] and Weaver [Blythe, 1994] use probabilistic operators and creates plans with a high probability of achieving their goals. These planners do not rely on having complete information about the state of the world and can create conditional and iterative plans. However, these planners ignore the utility of each outcome and the cost of achieving the goal state[1]. More recently, planners such as Williamson's Pyrrhus planner have combined domain-independent, partial-order planning with utility theory to produce plans with high utility in deterministic domains [Williamson and Hanks, 1994]. Bringing both utility theory and probability theory together in a single planner, the DRIPS planner provides a full decision-theoretic framework for evaluating plans generated using domain-independent, hierarchical refinement techniques.

In characterizing decision-theoretic planners, we need to consider the attributes used to characterize classical artificial intelligence planners and the degree to which the planner allows probabilities and utilities, the components of decision theory. In the following section, we give a brief outline of eight dimensions along which decision-theoretic planners can be classified. For more information on the classification of classical artificial intelligence planners and their history, we recommend [Drummond and Tate, 1989].

5.1.1 Classical AI Planner Characteristics

In this section, we examine five dimensions along which classical artificial intelligence planners can be classified. These dimensions are also useful for characterizing decision-theoretic planners.

Domain Independence

A domain specific planner solves planning problems in a particular domain. Retargeting a domain specific planner to solve problems in another domain usually involves rewriting some or all of the planner. Domain independent planners separate the domain dependent and domains independent parts of the planner. A domain independent planner encodes a general planning algorithm that takes both a domain description and a problem description and produces a plan. A domain independent planner can be applied to a problem in another domain by supplying a new domain description.

[1] Recent work by Koenig has shown that some decision-theoretic planning problems for risk sensitive agents can be transformed into problems where finding a solution with a high probability of success translates into a solution with a high expected utility in the original problem [Koenig and Simmons, 1994].

The advantage of domain independence is that the planning code can be written, tested and debugged once and then reused, provided that a domain can be adequately represented in the language the planner uses to describe domains. The advantage of a domain specific planner is that it can be tuned to a particular domain. The characteristics of a domain may admit a simplified version of a planning algorithm or a special purpose algorithm that could not be used in a general purpose planner. An application like real-time motion planning for a robot arm may warrant creating a special purpose planner to meet the real time performance constraints. The basic tradeoff is between the time needed to create the planner and the efficiency of planning. However, it is not always the case that using a domain independent planner reduces the time needed to create the planner. It may be very difficult to create an adequate domain description that would allow a domain independent symbolic planner to solve a robot motion planning problem. Creating a domain dependent planner may actually take less effort. There are also degrees of domain independence. A robot motion planner may accept a description of the arm, including the joints and link lengths that would allow it to be easily retargeted to planning for different robot arms. Such a planner would not be domain independent, but would not be specific to a particular robot arm.

Search Space

The earliest artificial intelligence planning systems, such as GPS [Newell and Simon, 1961], planned in the space of world states. A problem was represented as an initial world state and a set of predicates on the desired, or goal, world state. A forward chaining planner, such as TLPlan [Bacchus and Kabanza, 1996], selects an operator and propagates the initial state through the effects of the operator to create the next state. The planner continues to select operators and apply them until the goal state is reached or the planner is forced to backtrack. Backward chaining planners, such as GPS, work similarly, but instead of propagating the initial state through a sequence of operators, they regress the goal state back through a sequence of operators until the initial state is reached.

As an alternative to searching in the world space, partial order planners, such as Noah [Sacerdoti, 1975], NONLIN [Tate, 1977], SNLP [McAllester and Rosenblitt, 1991] and UCPOP [Pemberthy and Weld, 1992], search a space of partial plans. The planner starts with a null plan and adds operators, ordering constraints and variable bindings until the plan is guaranteed to transform the initial state into the goal state. The advantage of plan space planning is that it allows the planner to delay commitments to operator ordering and variable binding until more information is available. A partial-order planner can add actions to the middle of a plan as well as to the ends of a plan. Using a partial order plan representation can reduce the amount of backtracking required to find a solution. However, evaluating a partial-order plan to determine how much of the task it accomplishes is harder than evaluating a sequence of actions. The most efficient search space depends on the domain and the problem [Stone *et al.*, 1994].

Infinite Search Spaces

Whether a planner searches a space of plans or a space of world states, we can characterize planners and domains into those with finite search spaces and those with infinite search spaces. A planner that searches a finite space could potentially enumerate the entire space, eventually finding the best plan or a satisficing plan, if one exists. With an infinite search space, it is not possible to search the entire space.

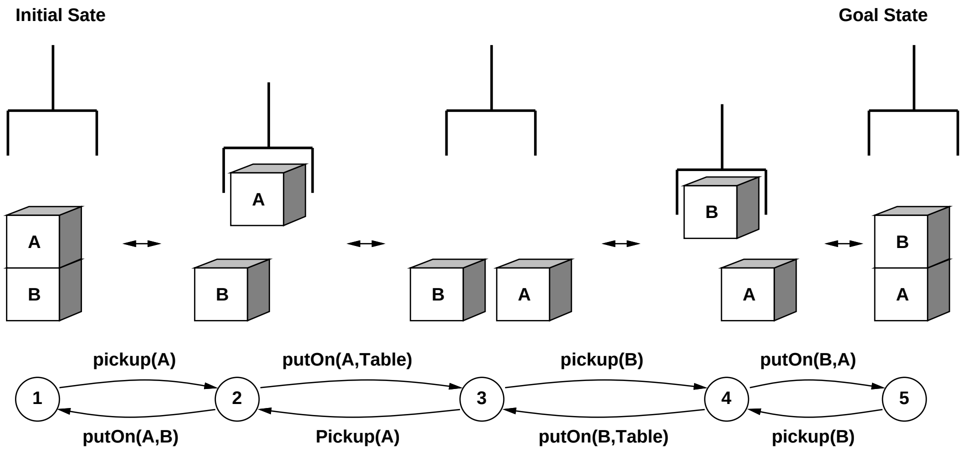

Figure 5.1: Simple blocks world problems can have infinite plan spaces. In this example, the state space is finite (and rather small) while the plan space is infinite.

Consider the simple two block problem illustrated in figure 5.1. The shortest plan that achieves the goal has four actions, but there are an infinite number of longer plans that achieve the goal. These plans correspond to the paths from state 1 to state 5 in the state graph, including loops in the graph. There are also an infinite number of plans that do not achieve the goal. Classical artificial intelligence planners recognize world state loops and goal loops to keep the planner from getting stuck in a cycle that would prevent it from finding a valid plan. Decision-theoretic planners also need to avoid getting caught in loops, but cannot rely on just detecting the loop. Some loops, such as repeating a test to get better information, are required to get plans with high expected utility. Decision-theoretic planners that allow infinite search spaces and loops are more expressive, but need mechanisms to ensure that they do not get stuck searching the wrong part of an infinite search space [Goodwin, 1996].

Abstraction

One of the basic methods for improving planner efficiency is to plan at an abstract level and to consider only the details when a valid abstract plan has been found. Abstraction can be achieved by using abstract operators, such as macro operators, or by simply ignoring generally irrelevant details in the domain. A plan at an abstract level can focus the planner on relevant attributes and reduce the amount of the space that must be searched. In decision-theoretic planners, determining that an abstract plan has low expected utility allows the planner to prune the abstract plan and the set of more detailed plans that it represents, without having to generate the full set of more detailed plans. But not all abstractions are useful. For some domains and problems, bad abstractions hide relevant details from the planner, causing it to search more of a larger search space [Bäckström and Jonsson, 1995].

A planner can be characterized by the types of abstraction it allows. The most common forms of abstraction are action abstractions. These include macro-abstraction where sequences of operators are abstracted into a macro-operator and instance abstraction where similar operators are grouped together and summarized [Haddawy and Doan, 1994]. Simpler forms of abstraction just ignore some of the preconditions and effects of the operators [Knoblock, 1991].

On-line versus Off-line Planning

In off-line planning, a plan is prepared well in advance of its use and may be used repeatedly. Under these conditions, finding very high quality plans is generally more important than saving a small amount of planning time. Conversely, in on-line planning, a plan is generated and used immediately. The overall efficiency depends not only on the quality of the plan, but the speed with which it can be generated. Whether planning should be done on-line or off-line is determined by the characteristics of the domain. If the information needed to do the planning is available before the plan is needed, then creating the plan ahead of time can improve efficiency.

On-line planning also raises issues related to coordinating planning with execution that off-line planning does not face. An on-line planner need not create an entire plan before execution can begin, but if planning and execution are overlapped, the planner must take into account the actions that will be carried out while it continues to plan.

5.1.2 Decision-Theoretic Planner Characteristics

Utility

Utility theory provides methods for assigning relative values to outcomes [Raiffa, 1968]. Traditional artificial intelligence planners evaluate plans on a binary scale and assign plans that achieve the desired goals a high value and assign all other plans a low value. Decision-theoretic planners typically assign utility values on a real number scale, where the utility value reflects the relative tradeoff between goal achievement and resource consumption.

Probability

Some domains are not deterministic and the outcome of an action may be known only probabilistically. This is particularly true if the planner only ever has probabilistic information about the state of the world. Planners that can handle probabilistic domains are more general, but the extra planning machinery they need adds complexity and can reduce efficiency when used for deterministic domains.

Combining utility theory and probability theory gives a full decision-theoretic planner that can account not only for the desirability of each outcome but also its probability.

Satisficing

Planning is computationally intractable even for classical planners where the objective is only to find a valid plan [Chapman, 1987]. In looking for a valid plan, classical planners make an implicit choice not to look for the best plan, but to look only for a satisficing plan, using as little computation as possible [Simon and Kadane, 1974]. The algorithms these planners use to perform search tends to find shorter, and generally more efficient, plans first. By adopting the first valid plan found, these planners heuristically attempt to improve overall efficiency, taking into account the planning time as well as execution time.

The introduction of decision theory to planning allows the planner to compare the quality of plans and to look for the plan with the highest expected utility. However, searching until a plan with the highest expected utility is found does not necessarily produce the best performance. Decision-theoretic planners can still choose to satisfice rather than optimize. The advantage of satisficing in a decision-theoretic framework is that the planner can assess the quality of its current plans and possibly asses the opportunity cost of forgoing more planning. The choice of whether to satisfice or optimize depends on the domain and the particular problem. In domains where computation is much faster than execution, optimizing is typically better. When execution is faster than computation, reactive systems with a minimum of planning are appropriate. In between, it is advantageous to have the planner decide how much planning to do. Decision-theoretic planners can be characterized by whether they always optimize or can satisfice.

5.2 Planner Classifications

The space of possible decision-theoretic planner is large, even if we restrict our analysis to the eight attributes listed previously. Although the four planners we have selected for analysis cannot completely fill this space, they do form a diverse set that spans much of the space and are representative of the decision-theoretic planners created to date. The Xavier route planner is a simple domain dependent planner that provides simple examples for examining meta-level questions that can be illustrated graphically. The Pyrrhus planner is a decision-theoretic planner derived form the line of research into classical partial-order planners. It combines domain independent classical planning with utility theory. Another line of planning research into hierarchical task network planning is represented by the DRIPS planner that uses a task network, combines utility theory with probabilities. Finally, the robot-courier planner provides a well understood domain specific anytime planning algorithm that can be easily analyzed to provide a mathematical model against

	Domain Independent	Search Space Type	Search Space Size	On-line/ Off-line	Abs.	EU Utility	EU Prob.	Optimizing/ Satisficing
Xavier	Domain Dependent	World Space	Finite	On-line	Yes	Yes	Yes	Max EU
Pyrrhus	Domain Independent	Partial-Order Plan Space	Infinite	Off-line	No	Yes	No	Max Utility
DRIPS	Domain Independent	Ordered Plan Space	Infinite	Off-line	Yes	Yes	Yes	Max EU or High EU
Robot Courier	Domain Dependent	World Space	Finite	On-line	No	Yes	No	Satisficing

Table 5.1: Summary of planner characteristics

which to compare empirical results. It is prototypical of the class of anytime planning algorithms [Boddy and Dean, 1989].

Table 5.1, summarizes the characteristics of each of the four planners along eight dimensions. We briefly explain the characterization of each planner before going on to give detailed descriptions in the subsequent sections.

The Xavier route planner is a domain specific planner used for routing a mobile robot in an office environment. Used on-line, it generates routes as they are needed by searching a finite world space to find an optimal route. An optimal route in this case is one with the shortest expected travel time. The planner uses abstraction for efficiency and accounts for probabilistic events like encountering a closed door.

The Pyrrhus planner [Williamson and Hanks, 1994] is an enhancement of SNLP [McAllester and Rosenblitt, 1991] that allows it to use utilities and backtracks until it finds a plan with the highest utility. The planner inherits its domain independence and infinite partial-order plan search space from its SNLP roots. The lack of probabilistic reasoning limits the planner to deterministic domains. It is usually used off-line to generate optimal solutions.

The DRIPS planner [Haddawy and Doan, 1994], is a full decision-theoretic planner that searches a space of ordered plans. It uses an action abstraction hierarchy to define the space of plans and, with recent extensions [Goodwin, 1996], searches an infinite space of plans. DRIPS is domain independent and is typically used off-line to evaluate medical treatment policies. Abstraction refinement is the principle method used to efficiently search the space of plans in order to find an optimal or satisficing plan.

The robot-courier tour planner that uses the two-op traveling salesman tour improvement algorithm represents the class of anytime planning algorithms. This algorithm can be interrupted at anytime to return a plan. With more computation, the quality of the plan returned improves. The two-op algorithm is domain specific and does not allow abstraction or probabilistic actions. These restrictions make the planner easier to analyze and allow our meta-level control methods to be compared both analytically and empirically to other methods.

5.3 Xavier Route Planner

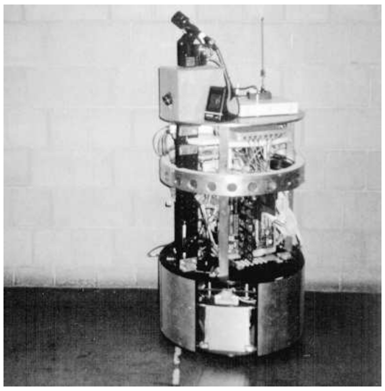

Figure 5.2: Xavier the robot

The Xavier robot, pictured in figure 5.2, performs deliveries and related tasks in an office environment. It provides a test-bed for research into a range of topics from low-level real-time obstacle avoidance to task level scheduling and coordination. The robot accepts task requests via its world wide web interface (http://www.cs.cmu.edu/ ˜ Xavier). The interface provides continuous feedback on the position and status of the robot as it traverses Wean Hall.

Figure crashes some postscript interpreters.

Figure 5.3: Xavier Software Architecture

The robot has been in almost daily use in our building since December 1995, serving over 1500 navigation requests and traveling a total of more than 60 kilometers in the process [Simmons *et al.*, 1997]. Xavier can travel up to 60 centimeters per second in peopled environments. Its task completion rate is 93 percent. Its travel speed is currently limited only by the cycle time of its sonar sensors, and tasks fail mostly due to radio link problems – both problems unrelated to the robot software.

Xavier is built on top of a 24 inch diameter RWI B24 base, a four-wheeled synchro-drive mechanism that allows for independent control of the translational and rotational velocities.

Its sensors include bump panels, wheel encoders, a sonar ring with 24 ultrasonic sensors, a Nomadics front-pointing laser light striper with a 30 degree field of view, and a color camera on a pan-tilt head. Control, perception, and planning are carried out on two on-board 486 computers, and an additional 486 lap-top computer is used to monitor Xavier's status and for debugging. The computers are connected to each other via thin-wire Ethernet and to the outside world via a Wavelan wireless Ethernet system [Hills and Johnson, 1996].

The control software for the Xavier robot, illustrated in figure 5.3, is organized into a set of modules with well defined interfaces. The lowest level is responsible for real-time obstacle avoidance, while providing sensor and status information to the upper levels. The robot currently uses the curvature-velocity method for obstacle avoidance [Simmons, 1996], but also has implementations of the potential field and vector field histogram methods that can easily be substituted. The navigation level is responsible for tracking the position of the robot using information from the wheel encoders and by analyzing the sensor data looking for features such as walls and corridor openings. It uses a partially observable Markov decision process model (POMDP) to maintain a probability distribution over possible robot locations and orientations [Koenig *et al.*, 1996]. This probability distribution, along with the route plan, are used to select the appropriate directive (go forward, turn or stop) to send to the obstacle avoidance level. The Xavier route planner takes the performance characteristics of the robot into account to find a route with the shortest expected travel time. The route planner gets the destination from the task scheduler and the current location from the navigation module and returns an appropriate route to the destination. The task scheduler is responsible for queuing and ordering tasks so that they can be done efficiently. The user interface allows users to specify tasks and provides constant feedback on the location and status of the robot. The modules are tied together using the Task Control Architecture that provides facilities for communication and coordination between multiple processes running on multiple, heterogeneous computers [Simmons, 1994].

Navigation

In order to explain the Xavier route planning problem, we need to first describe some of the operational details of the navigation module since it provides the mechanisms for executing the route plans. As we stated previously, the navigation module is responsible for tracking the position of the robot as it navigates towards a goal location. The task is complicated by noisy sensors and effectors and by the fact that the robot may have only approximate distance information in its map.

The navigation module receives sensor reports about the surroundings from its sonars and laser light striper and information about how far it has traveled from the wheel encoders. This information is integrated into a local occupancy grid that is used as a basis for recognizing features in the environment, such as walls and corridor openings. Feature recognition in the local occupancy grid is used to create three virtual sensors that report the feature to the right of, in front of, and to the left of the robot. Integrating information from multiple sensors and multiple readings into the occupancy grid makes these virtual sensors more accurate, but still not perfect [Simmons and Koenig, 1995]. To deal with the remaining uncertainty, the navigation module uses a partially observable Markov decision process (POMDP) model to track the position of the robot as a probability distribution. The model is only partially observable because, although the robot can observe features in the environment, it cannot directly sense its location. For tractability, our POMDP model discretizes the location and orientation of the robot. The orientation is discretized into the four compass directions, in keeping with the rectilinear nature of most office environments. The location is discretized with one meter resolution; a reasonable compromise between increased precision that a finer resolution would provide and the extra processing and memory that it would require. The navigation module maintains a probability distribution over the states in the POMDP model to represent its belief about the likely pose of the robot. The distribution is updated using Bayes' rule whenever a new sensor report is received.

Decision-Theoretic Route Planning

The use of Markov models for position estimation and action execution suggests using POMDP algorithms on the same models for goal-directed route planning. At present, however, it is infeasible to determine optimal POMDP solutions given our real-time constraints and the size of our state spaces (over 3000 states for one floor of our building) [Lovejoy, 1991, Cassandra *et al.*, 1994]. Instead, our planner currently associates a directive (turn right, turn left, go forward or stop) with each state in the POMDP model. The navigation module chooses a directive to send to the obstacles avoidance module greedily by weighting each directive by the probability that the robot is in a state that suggests that directive and selecting the directive with the highest total weight:

$$\arg \max_{d \in Directive} \sum_{s \in States | d(s)=d} p(s)$$

The preferred directive can be calculated very efficiently, making the action selection component very reactive.

Recent advances in approximate algorithms for solving POMDP problems [Littman *et al.*, 1995] [Parr and Russell, 1995] suggest that it might eventually become feasible to use them for route planning. The promise of such methods is that they can plan for information-gathering actions and they allow the amount of computation to be traded off against the quality of the resulting plan. How to perform this tradeoff is critical to the performance of the robot.

We model the route-planning problem as a decision problem: the planner generates a set of alternative plans, evaluates the likelihood of each outcome for each plan, evaluates the utility of each outcome and, finally, picks the plan with the highest expected value; in this case the plan with the smallest expected travel time. Generating all possible plans is prohibitively time consuming for all but the smallest problems. To improve efficiency, we use several techniques: we use abstractions of the state space when generating plans, we generate and evaluate only a subset of the possible plans, and we do not require plans to be complete before we evaluate them.

Route Generation

The route planner for the Xavier robot creates a route for the robot to follow from one location to another. The problem is different from path planning because the plan is at the level of corridors to follow and rooms to traverse rather than an exact and continuous set of points to move through. Given a route plan, the robot uses its local obstacle avoidance and position estimation behaviours to traverse the corridors while avoiding people and obstacles.

When creating the route, the Xavier route planner must take into account the performance of the robot, since it is easier for the robot to follow some routes than others. For example, the robot can easily miss a turn in the middle of a corridor whereas a turn at the end of a corridor is hard to miss because the robot would run into the wall. All other things being equal, the planner should prefer routes with turns at the ends of corridors rather than in the middle of corridors. The planner also has to allow for blockages, such as closed doors. Since the robot does not have an arm, it cannot open doors and has to find another route if it encounters a closed door. As with turns in the middle of a corridor, the planner should prefer plans that avoid doors, all other things being equal. If avoiding doors and turns in the middle of corridors requires a longer route, the planner must weigh the extra distance against the cost of recovering from a closed door or a missed turn.

The Xavier route planner generates a sequence of routes to the goal using a modified A* search algorithm. The A* search algorithm is a best first search algorithm that uses an admissible heuristic [Hart *et al.*, 1968, Hart *et al.*, 1972]. In the case of the Xavier planner, the Euclidean distance to the goal location is used as the admissible heuristic, causing the planner generates the shortest distance plan first. The modifications to the A* algorithm allow it to be called repeatedly to generate a sequence of routes with monotonically non-decreasing distance. As each plan is generated, it is evaluated using a simple forward projection to estimate the expected travel time.

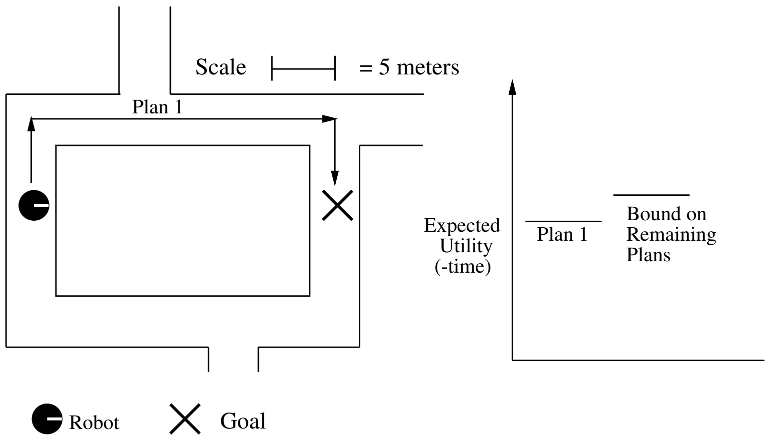

Figure 5.4: The first plan generated is the shortest route, but the time to follow the route is longer than the distance/speed.

To illustrate the operation of the route planner, we present a sequence of examples that cover the details relevant to meta-level control. The first example, in figure 5.4, shows a simple route planning problem and the first plan found by the planner. When the plan is evaluated, the forward projection takes into account the possibility of missing the last turn and adds the expected recovery time.

$$\begin{aligned}
Utility(plan1) &= -Cost(plan1) \\
&= -distance(plan1)/aveSpeed - P(missingTurn) * E(recoveryCost) \\
&= -38m/0.30m/s - 20.0s = -146.67s \\
Utility(remainingPlans) &\leq -distance(plan1)/aveSpeed = -38m/0.30m/s = -126.67
\end{aligned}$$

Since plans are generated in non-decreasing order of length, we can conclude that no paths are less than 38m in length. In general, this allows us to use the length of the first plan as a lower limit on the length of all remaining plans, which in turn provides an upper bound on the utility of the remaining plans. The chart on the right side of figure 5.4 shows the expected utility of plan 1 and the upper bound on the expected utility of the remaining plans. The expected utility of plan 1 is less than the bound on the utility of the remaining plans because of the cost added to account for the possibility that the robot will miss the last turn.

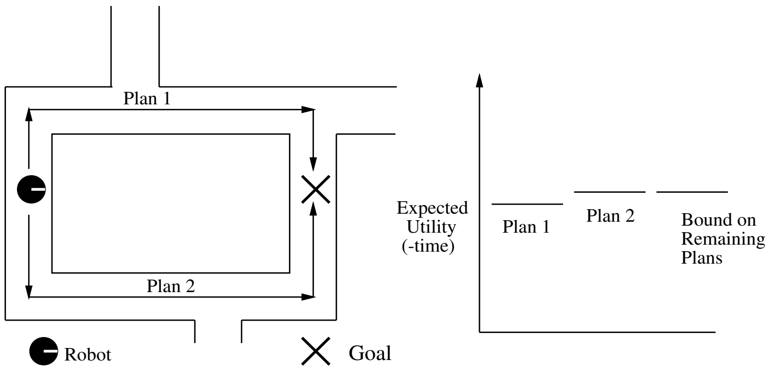

Figure 5.5: The second plan generated has the same utility as the upper bound on the remaining plans, so plan generation can stop.

Figure 5.5 shows the second plan generated, the relative utility of plan 1 and plan 2 and the bound on utility for the remaining plans. Since plan 2 does not have any corners that can be missed or doors that can be closed, the expected utility is simply:

$$\begin{aligned}
Utility(plan2) &= -distance(plan2)/aveSpeed \\
&= -42m/0.30m/s = -140.0 \\
Utility(remainingPlans) &\leq -distance(plan2)/aveSpeed = -42/0.30m/s = -140.0
\end{aligned}$$

The expected utility of plan 2 is the same as the limit on the upper bound of expected utility of the remaining plans, so plan 2 is a plan with the highest possible expected utility. Plan generation can stop since there is no point in continuing when a plan with the highest

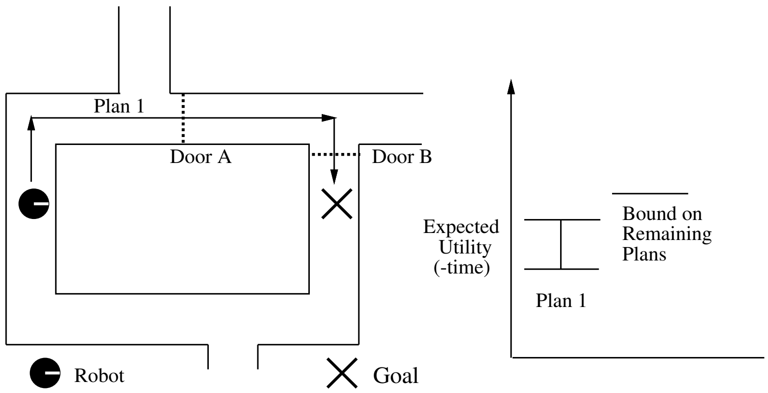

Figure 5.6: Doors A and B may be closed, introducing uncertainty about the time needed to follow the route leading to a range of expected utility.

expected utility has been found. Note that there may be many other routes to the goal, some of which may also be optimal, but the planner does not need to consider these.

In the case of simple plans that can be fully elaborated and evaluated, the stopping criteria is clear, as is the choice of what to do next. The planner generates and evaluates more plans until a plan with the highest expected utility is found. In the case of abstract plans and partial plans, the situation is less clear. Consider the example in figure 5.6 where the shortest route goes through two doors that might be closed. The expected utility is given by:

$$\begin{aligned} Utility(plan1) &= P(A)P(B)Utility(plan1|A,B) + \\ &\quad P(\overline{A})Recovery(plan1|\overline{A}) + \\ &\quad P(A)P(\overline{B})Recovery(plans1|A,\overline{B}) \end{aligned}$$

where the P(A) is the probability that door A is open and Utility(plan1|A,B) is read "the utility of plan 1 given that doors A and B are open." $Recovery(plan1|\overline{A})$ is the expected utility given that plan 1 is followed to door A and the door is found to be closed. For simplicity, we assume that the doors do not open or close while the robot is traversing the route. This ensures that the robot can always backtrack through doors it has gone through. It also removes the possibility that simply waiting for a door to open is the best plan.

In order to fully evaluate the expected utility of plan 1, the planner needs the probabilities that the doors are closed and the time needed to recover from a closed door. The catch is that to determine how long it will take to recover, the planner would have to completely plan the recovery route, including recovery plans for doors the recovery route would go through. This work is unnecessary if the plan is not potentially optimal. Instead, the planner uses bounds on the recovery cost and calculates a range of expected utility, as shown on the right of figure 5.6.

The use of bounds allows the planner to prune plans that are not potentially optimal without further work. Consider the set of plans in figure 5.7 where the planner has created three plans. All the plans have ranges of expected utility, but the possible values for plan 1 are greater than the possible values for plan 2, $Utility(plan1) > Utility(plan2)$. Plan 1 dominates plan 2, so plan 2 is removed from consideration.

Showing dominance of one plan over another is a key method used by decision-theoretic planners to prune the search space. The simple form of dominance used to prune plan 2 in figure 5.7 requires non-overlapping ranges of expected utility. Other forms of dominance, such as local dominance, do not require disjoint ranges of expected utility and can prune sub-optimal plans with less refinement. For instance, if two plans differed only in the route between two doors in a single room, we could conclude that the plan with the shortest route in the room also had the shortest route to the goal, without knowing anything else about either route. Techniques for showing dominance were discussed in section 4.2.

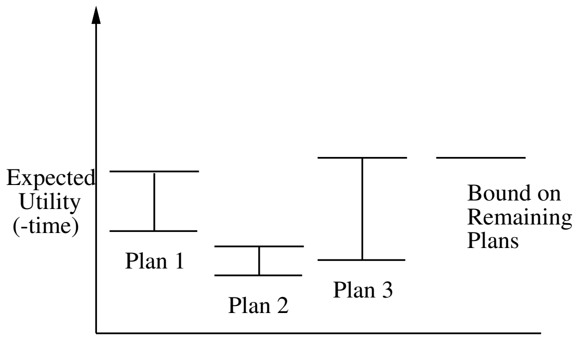

Figure 5.7: Plan 2 is clearly dominated by plan 1 and can be pruned. The question is whether to refine plan 1 or plan 3, or to generate a new plan.

Bounds

To avoid planning for all possible contingencies, the range of recovery costs must be bounded. Tighter bounds are better, because they lead to pruning of sub-optimal plans with less work, but doing extensive calculations to find good bounds defeats the purpose of using bounds rather than calculating the exact value. The critical requirement is that the bounds not under-estimate the expected utility of an optimal plan or over-estimate the expected utility of sub-optimal plans. If either of these constraints are violated, then the plans with the highest expected utility may be pruned from the search space.

Two sources of easily calculated bounds are the original partial plan and limiting cases defined by the environment. The original plan, for which the recovery plan is being bounded, is already partially elaborated and gives information about the unplanned recovery plan. In case of a closed door, the cost to move from the door to the goal in the original plan can be used as a lower bound on the cost for the recovery plan. If going through the door was the best route, then going around the door could only be worse. This reasoning leads to a correct route planner, but the bounds it produces are not completely accurate. It is important to note that this method can under-estimate the expected utility of a sub-optimal plan, but not an optimal plan. In figure 5.8 the route through the door is sub-optimal and the recovery route is shorter than the original route from the door to the goal. A recovery

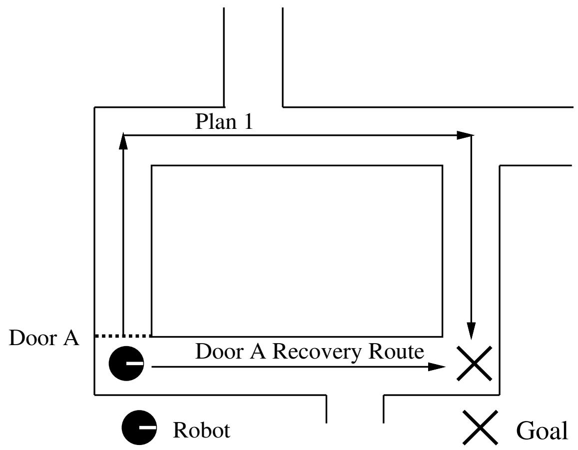

Figure 5.8: The recovery route from door A is shorter than the original distance along path 1.

plan that used the better route would have an expected utility outside of the bounds. This can happen only with a sub-optimal plan since a better plan could always be constructed by replacing the original route by the recovery route. Using this approximate bound does not violate the requirement that the expected utility of optimal plans not be understated.

The upper bound is a bit more problematic since it must account for the longest of all the unexamined routes and the possibility that the robot may encounter more closed doors. One way to solve this problem is to consider a limiting case where the robot uses an exploration strategy to visit every accessible location at least once and terminates when it reaches the goal or determines that the goal is unreachable. A depth-first search with backtracking will eventually visit all reachable locations. It is equivalent to traversing a spanning tree rooted at the robot and covering the accessible area. The robot traverses the spanning tree and either reaches the goal or completes the traverse and determines that the goal is not reachable. In traversing the spanning tree, the robot travels each corridor twice. The limit on the total distance is twice the length of all the corridors, $2\sum_{i=1}^{n} length(corridor_i)$. This bound is not very tight, but does serve to limit the upper bound on recovery cost and the lower bound on expected utility in the worst case. It also has the advantage that it can be calculated once for any building layout and does not require probability estimates.

$$Recovery(plan1|A) = \left[-2\sum_{i=1}^{n}\frac{length(corridor_i)}{average(Speed)} \ldots Utiltiy(plan1|A,B)\right]$$

Abstraction

In route generation, subsequent plans are generated by repeated calls to the A* search algorithm and are generated in a monotonically non-decreasing order of length. The problem with this approach is that it tends to produce sequences of very similar plans.

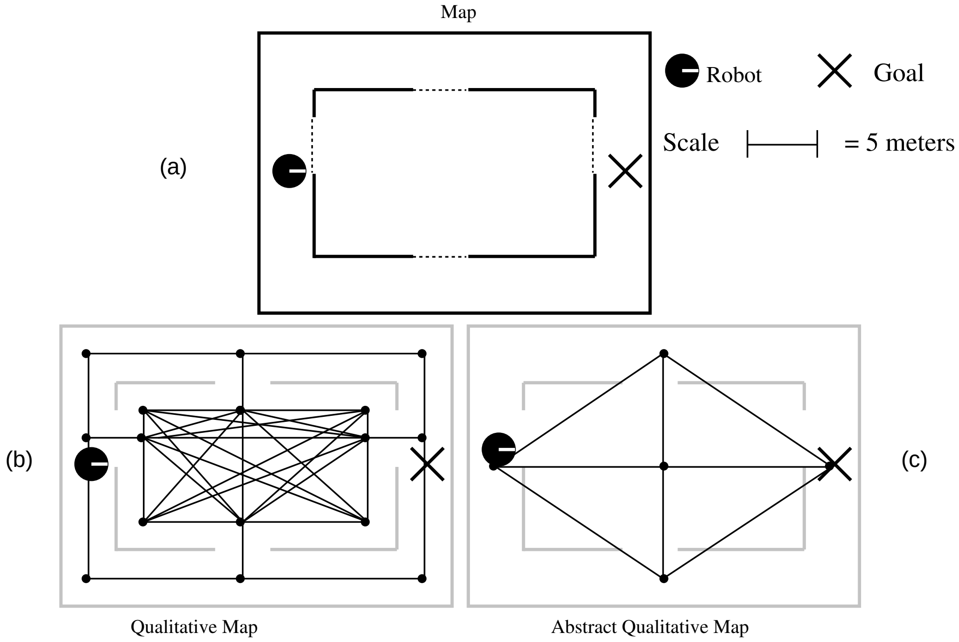

Figure 5.9: Abstraction leads to a reduced number of qualitatively different routes.

Consider the simple route planning problem in figure 5.9. Each significant location is a node in the qualitative map. These locations include corners, intersections, dead-ends and the inside and outside of each door. The shortest route from the robot to the goal travels straight through the room, in the left door and out the right. One of the next shortest routes goes through the same doors, but travels through the room via the upper right corner. A similar route through the upper left corner has the same length. The routes through the other corners of the room are only slightly longer. Routes that go through both of the upper or lower corners are only slightly longer again. The problem arises because the nodes in a single room are fully connected. A route between two nodes in a room can use any subset of the remaining nodes and arrange them in any order. The number of such routes is given by $\sum_{i=0}^{n-2} \binom{n-2}{i} i!$ where i is the number of nodes in the room. The problem arises because the nodes in a single room are fully connected. A route between two nodes in a room can use any subset of the remaining nodes and arrange them in any order. The number of such routes is given by $\sum_{i=0}^{n-2} \binom{n-2}{i} i!$ where i is the number of nodes in the room[2]. There are, in fact, 1957 routes through the room using the same two doors. The problem for the planner is that all these routes are very similar. Deciding between them is difficult since the expected utility is very nearly the same. The uncertainty introduce by the range of recovery costs for either of the doors will swamp this difference and require the planner to fully expand each plan before the optimal one can be determined.

Although the number of routes between the door on the left and the door on the right is

[2] Subject to the restriction that the route cannot pass through the same node twice.

problematic, the route from the robot to the goal can use any combination of 2 or 4 doors. For each combination, there are 1957 possible routes through the room. The total number of routes in figure 5.9 is

$$\binom{4}{2}\sum_{i=0}^{n-2}\binom{n-2}{i}i! + \binom{4}{4}\sum_{j=0}^{n-4}\left[\binom{n-4}{j}j! * \sum_{i=0}^{n-2}\binom{n-2}{i}i!\right] = 12491$$

, many of which are very similar.

To address the problem of many similar routes, we introduce an abstraction of the qualitative map, which is itself an abstraction of the full POMDP model. Each room and corridor is abstracted into a single abstract node, as shown in figure 5.9b. A single link connects abstract nodes linked by doors or intersections. Abstraction significantly reduces the number of routes, ensuring that each one is qualitatively different. In figure 5.9, the number of routes is reduced to 9, a reduction of over 3 orders of magnitude.

Using abstraction, the distance of a route and the expected utility can no longer be calculated exactly, even in the absence of potential blockages. A route through an abstract node will have a range of distances corresponding to the shortest and the longest routes through the room or corridor. For example, the range for the top corridor in figure 5.9 is bounded from below by the distance from the door to the nearest end and from above by the length of the corridor.

When refining an abstract plan, the planner can now select either a contingency to plan for or an abstract node to expand. Recovery plans are created by finding a route from the door to the goal, assuming that the door is closed. These routes are created using the same abstract map as is used for the original plan. To fully elaborate a plan, the planner has to expand all the abstract nodes, both in the recovery plans as well as in the nominal route.

Having to expand the abstract nodes raises the question of whether the complexity avoided by using abstraction has just been delayed to a later planning stage. There are still 1957 routes through the room in figure 5.9 that the abstract node could be expanded into. Does the planner have to consider all possible combinations of routes through each abstract node in order to find the best plan? Fortunately, this is not the case since there are some differences between the planning a route through a single room or corridor and planning a route through a sequence of rooms and corridors. Because of the way we have organized the abstraction, each abstract node can be refined independently and still produce the route with the shortest expected travel time.

Any optimal route that traverses an abstract node between a given entry and exit must use the optimal route through the abstract node between the given entry and exit. For example, any optimal route in figure 5.9 that entered the left door and exited the right door must use the route between the two doors with the shortest expected travel time. We can prove this using a form of local dominance. Suppose there was an optimal route that traversed an abstract node but did not use the best route between the entry to and the exit from the abstract node. The route could be improved by replacing the original route through the abstract node with the better route. But this contradicts the assumption that the original route was optimal. For a route to be optimal, it must include one of the optimal routes through each abstract node between the entry and the exit of the node. This observation allows us to solve each sub-problem independently without having to consider different combinations of routes through each abstract node.

Choice Points

The key meta-level control questions for the Xavier route planner revolve around the two choice points in the algorithm. The first choice is between generating another route and selecting one of the existing routes for refinement. If an existing route is selected for refinement, the second choice is to select the abstract node to refine or the contingency to plan for. The methods for making these choices are covered in chapters 6 and 7 respectively.

5.4 Pyrrhus planner

Pyrrhus is Williamson's decision-theoretic planner, based on SNLP [McAllester and Rosenblitt, 1991], is a generative, partial-order planner [Williamson and Hanks, 1994]. The planner searches an infinite space of partially ordered plans and uses restrictions on the operators and the utility function to ensure that the planner terminates with a plan with the highest utility. The planner handles only utility functions with deadlines and, since all actions take some minimum non-zero time, the deadline must eventually be reached as the length of a plan increases. Williamson also imposes a constraint on resource consumption. Every domain must adhere to "strict *asset-position monotonicity*", which means that an action that increases some resource must decrease another so that the overall value of the resources does not increase. In other words, the agent cannot get something for nothing.

The planner finds an plan with the highest utility by starting with a partial order plan that specifies only the initial state and the goal state. It uses the utility function to calculate the upper bound on expected utility for this abstract plan that represents all possible plans in its search space. The planner proceeds by planning for goals and sub-goals and resolving conflicts. In situations where there is more than one way to achieve a goal or resolve a conflict, the planner creates a set of plans, one for each possible refinement. These plans are evaluated to determine their upper bound on expected utility. The planner continues to generate more detailed plans until it has found a complete plan that it can evaluate to get a lower bound one utility. This lower bound can then be used to prune dominated plans. The planner does not prune any plans from the search space until it has found one complete plan.

The planner has two choice points, the first is to choose which partial order plan to refine and the second is to select a flaw in the chosen plan to correct. The first choice point is analogous to selecting which plan to refine in the Xavier planner. The second choice is analogous to the choice of which contingency to plan for or which abstract node to expand. In the Pyrrhus planner, flaw repair can include addition an operator to achieve a goal or sub-goal, imposing an ordering constraint to prevent a conflict and deciding on a variable binding. As with the Xavier Route Planner, planning continues until an plan with the maximum expected utility is found.

5.5 DRIPS planner

A decision-theoretic refinement planner, such as DRIPS takes as input a probability distribution over initial states, an action abstraction hierarchy and a utility function. It returns a plan, or a set of plans, with the highest expected utility. The initial probability distribution gives the prior probabilities, such as the likelihood of a particular disease. The action abstraction hierarchy represents an abstraction of all possible plans in the domain. These actions can have conditional and probabilistic effects. The utility function maps chronicles of world states to utility values. A chronicle is one possible evolution of world states over time [McDermott, 1982]. The planner evaluates a plan by projecting the (conditional) effects of its actions to create a set of chronicles. The expected utility of the plan is calculated by weighting the utility of each chronicle by its likelihood. Abstract actions in the plan can have ranges of attribute values and ranges of probabilities, that lead to a range of expected utility values.

The planner begins with the top action in the abstraction hierarchy and refines it into a set of more detailed plans. These plans are evaluated and dominated plans are discarded. The planner continues to refine actions in one of the non-dominated plans until all the non-dominated plans have been fully expanded. The remaining non-dominated plans are the set of potentially optimal plans.

Action Abstraction

We begin our description of DRIPS by focusing on the abstraction hierarchy used to encode the set of possible plans. Encoded in the hierarchy are instance abstractions and macro abstractions [Haddawy and Doan, 1994]. Instance abstractions are used to group together a set of actions with similar effects, whereas macro abstractions are used to summarize sequences of actions.

Consider a simple medical domain with two tests and a treatment for a disease. Figures 5.10a and 5.10b give the definitions of the two tests and figure 5.10c gives their abstract instance action. The actions are conditional and probabilistic. The conditions, $disease = T$ and $disease = F$, used to label the branches, are mutually exclusive and exhaustive. Each conditional branch can have probabilistic effects, given as a set of changes to attribute values labeled with their probability. In our example, one test is 95% accurate for detecting the presence of the disease and costs \$160, whereas the other test is 98% accurate but costs \$300. Both tests are 90% accurate for detecting the absence of the disease. Abstract actions can have effects with ranges of probabilities and ranges of attribute values (figure 5.10c).

One possible treatment plan is to test for the disease and treat if the test is positive. For our example, let the cost of treatment be \$5000, which cures the disease with 100% certainty, and let the cost of an untreated case of the disease be \$100,000. Figure 5.11a

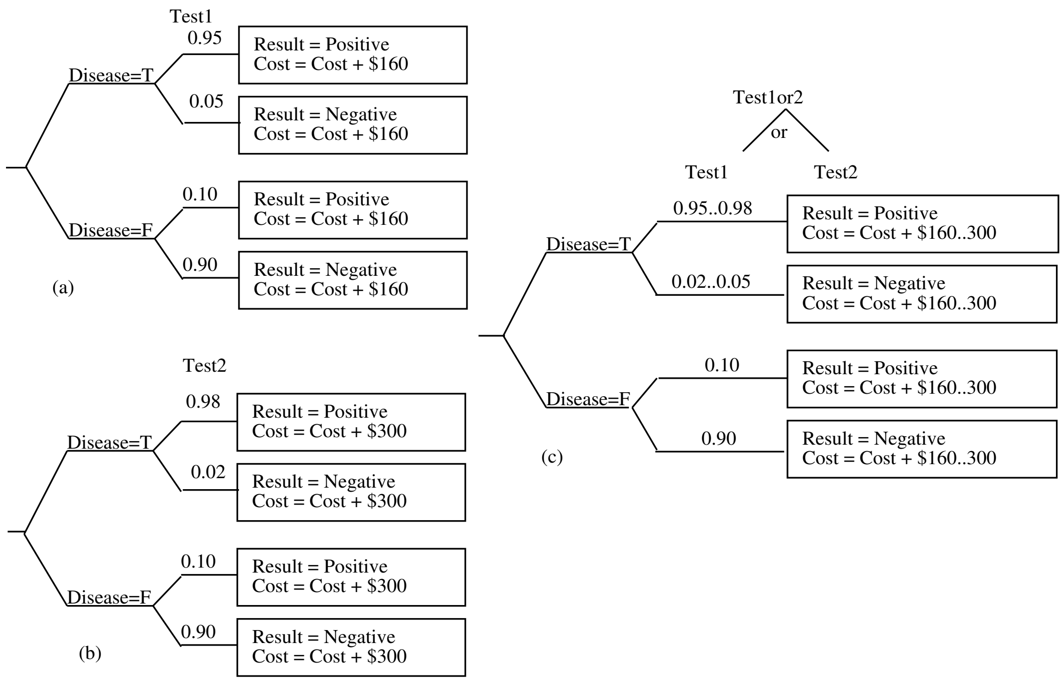

Figure 5.10: Test1 and Test2 are test actions that have different costs and accuracies. Test1or2 is an instance abstraction of the two tests.

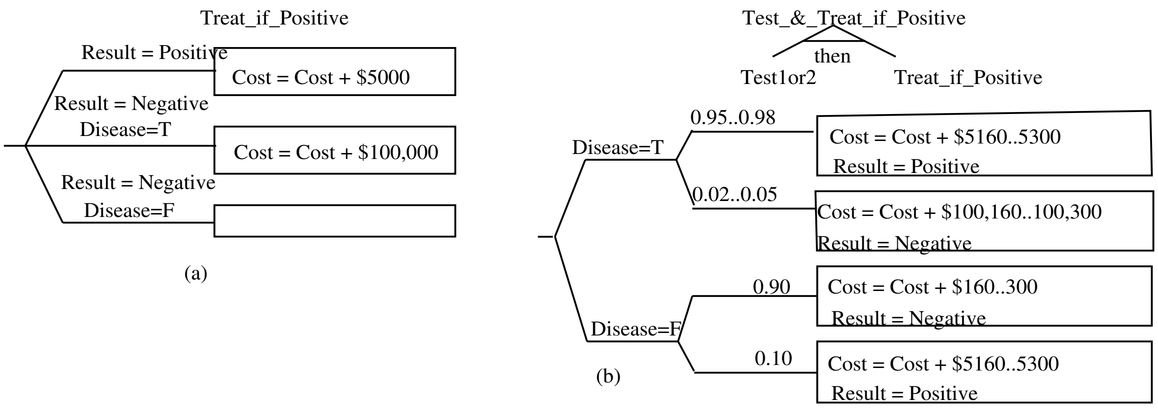

Figure 5.11: The sequence of a Test10r2 action (figure 5.10) and Treat_if_Positive action (a) are abstracted into a Test_&_Treat_if_Positive macro action (b).

gives the definition of the action that treats the disease if the test result is positive. The sequence of a test action followed by a Treat_if_Positive action is abstracted into the macro action given in figure 5.11b. The effects of the abstract action summarize the effects of the sequence, reducing the work needed to project the effects of the sequence. Fully expanding the Test_&_Treat_if_Positive action would give two plans, each with two actions. See [Haddawy and Doan, 1994] and [Doan and Haddawy, 1995] for more details on the abstraction mechanism.

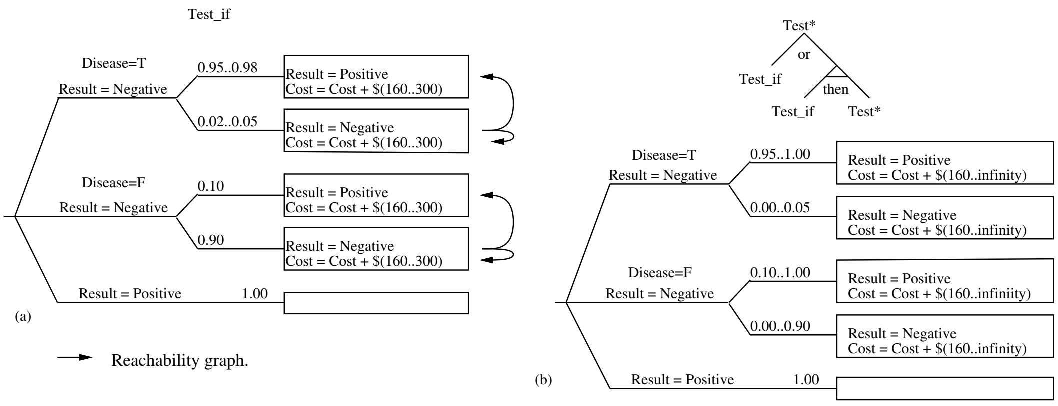

Figure 5.12: The Test_if action applies a Test action if there is no previous positive test result. Test*, a repeated action, is either a single Test_if action or a sequence of a Test_if action followed by a Test* action. The arrows show the reachability graph.

Infinite Plan Spaces

In the original version of the DRIPS planner, the space of plans represented in the abstraction hierarchy was finite, although arbitrarily large. The abstraction hierarchy did not allow recursion, a natural method of representing loops. This section describes how we have extended the planner to allow recursion [Goodwin, 1996].

Figure 5.12a defines a conditional action that applies a test unless a previous test result was positive. Figure 5.12b gives the recursive definition of the Test* action that includes one or more conditional tests. The effects of the Test* action give the range of possible outcomes. In the limit, the probability of detecting the disease is one, but the probability of a false positive is also one. The cost can range from the cost of one test to infinity for an infinite number of tests. These conditional effects are used by the planner to evaluate the bounds on the expected utility of a plan with the Test* action.

The planner begins with a plan that consists of the Test* action followed by the Treat_if_Positive action. This abstract plan represents all possible sequences of one or more tests followed by a treatment if one of the tests is positive. The planner proceeds by expanding the Test* action to create two plans: one with a single test and the other with two or more tests. Each plan is evaluated to determine the range of expected utility. If either plan is dominated, it is pruned from the search space. Expansion of the Test* action continues until the upper bound on the plan with the recursive action falls below the lower bound of a plans with a finite number of tests. In this example, the planner terminates with a plan that has two test actions, a Test1 action followed by a Test2 action, which is the optimal plan with an expected cost of $3325.

Method for Encoding Loops

A loop is a sequence of actions that is repeated one or more times until some condition is achieved. Figure 5.12b shows an action that represents a loop of test actions. It is an instance action that either applies a single test_if action or a test_if action followed by a recursive application of the loop action. The body of the loop is represented by a conditional action that applies the sequence of actions that form the body of the loop, unless the loop termination condition holds. In our example, the termination condition is a positive test result (figure 5.12a). If the condition holds, the action has no effect.

The conditional effects of the repeated action can be derived from the effects of the action that form the body of the loop. We have developed a method for automatically doing this using a reachability graph. A reachability graph is created by connecting a directed arc from each effect in the conditional action to every effect that could be reached in the next iteration of the loop, except for the null effect used to terminate the loop. The arrows in figure 5.12a give the reachability graph for the test_if action. Each of the effects with a negative result can loop back to itself or to the effect with the positive result on the next iteration of the loop. Effects with a positive result will terminate the loop on the next iteration and have no outgoing arrow. The reachability graph is used to create a set of recurrence equations. The solution to these equations gives the range of effects for the recursive action. See [Goodwin, 1996] for details.

Finding Optimal Plans with Finite Length

In our simple medical domain, the plan with the highest expected utility is a finite sequence of two tests. In general, the value of repeated information gathering actions, like a medical test, is limited by the value of perfect information. The planner will expand the recursive action until the value of information from the test is less than the cost of the test, and begins to lower the upper bound on expected utility. When the upper expected utility bound on the plan with the infinite action falls below the lower bound on the expected utility of another plan, it can be pruned. This method of pruning constitutes a proof that the infinite plan is not a the plan with the highest expected utility.

The planner will stop unrolling the loop only when $\overline{EU}_{loop} \leq \overline{EU}_{optimal}$. To ensure termination, the value of $\overline{EU}_{loop}$ must eventually be less than $\overline{EU}_{optimal}$. IN general, this may never occur. We can ensure that $\overline{EU}_{loop}$ is eventually less than or equal to $\overline{EU}_{optimal}$ by imposing conditions on the domain. The plan with the highest expected utility will be finite if, after some fixed number of unrollings, the marginal utility of unrolling the loop once more is negative and, for each additional unrolling, the marginal utility is non-positive. Under these conditions, the expected utility of the infinite plan will converge to minus infinity, or some sub-optimal finite value. The same condition can be imposed on the upper bound on expected utilities for plans, including plans with abstract actions. This ensures that plans with infinite recursion can be pruned after some number of unrollings of each loop. For the previous medical example, termination is ensured because the marginal benefit from each test decreases while the marginal cost remains the same.

There are many possible restrictions on the encoding of a domain and a utility function that can enforce the sufficient conditions necessary to ensure finite length plans with the highest expected utility. These can include restrictions on the form of the utility function and restrictions on the actions in the domain. One method is to have each iteration of a loop consume some finite amount of a limited resource. Plans that violate the resource limit are assigned low utility. Ngo and Haddawy have proposed using this method [Ngo and Haddawy, 1995]. Resource limits are used to calculate a limit on useful loop iterations, and the domain is encoded to allow only that many iterations of the loop. Williamson and Hanks use a more restrictive condition, strict asset-position monotonicity, and allow only deadline goals [Williamson and Hanks, 1994]. They use these restrictions to do optimal planning in Pyrrhus.

As with the Xavier Route Planner and the Pyrrhus planner, the DRIPS planner has two choice points, one related to plan generation and the other related to refinement selection. At any point in the search, the planner will have a set of partially elaborated, potentially optimal plans under consideration. Selecting the plan to work on and the abstract action to refine constitute the core of the meta-level control problem for the DRIPS planner. But, unlike the other planners, DRIPS can choose to satisfice rather than optimize. By accepting a plan that may not have the highest possible expected utility, the planner can improve overall expected utility and avoid getting caught in infinite loops. We explore the choice to satisfice in Chapter 8.

5.6 Robot-Courier Tour Planner

The robot-courier tour planner creates tours for a simplified robot courier that has to visit a set of locations to deliver messages and return to its initial location. The domain is taken from Boddy's dissertation where he used it as a vehicle for investigating deliberation schedules for anytime algorithms[Boddy, 1991b]. The problem differs from the problems tackled by the other planners since any random ordering of the locations is a valid plan. The problem for the planner is to find tours that are shorter and require less travel time. In this domain, the planner assumes that the delivery locations are located on an Euclidean plane and that the distance between locations is proportional to the travel time.

Tour planning for the simplified robot domain is equivalent to the classic traveling salesman problem [Lin and Kernighan, 1973], which is prototypical of an NP-hard problem. Finding an optimal solution that can take time exponential in the size of the problem[3]. Fortunately, this problem admits iterative improvement approximation algorithms that converge on good solutions. The most common of these techniques are edge-exchange algorithms that iteratively improve the tour by swapping small sets of edges.

The simplest of the edge-exchange algorithms is the two-opt algorithm that greedily selects pairs of edges to swap. Each pass of the algorithm scans all possible combinations of two edges and swaps the two that give the best improvement in the tour length. Figure 5.13 illustrates a swap graphically. The algorithm continues until no swaps that can improve the

[3] Assuming that $P \neq NP$.

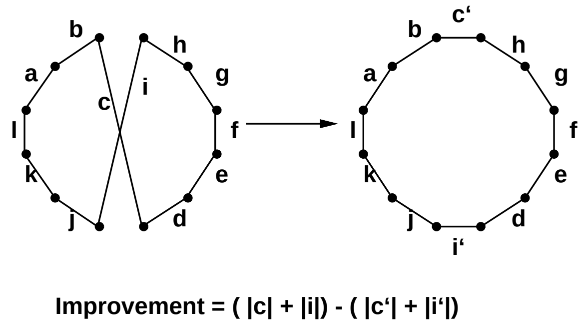

Figure 5.13: The two-opt algorithm repeatedly swaps the two edges that give the most improvement in the tour length.

tour length can be found. The result is not necessarily the shortest possible tour, but for large problems, with 100 locations, the result is within 8% on average [Lawler *et al.*, 1985].

The robot-courier tour planner does not have to choose which plan to work on or what part of the selected plan to refine. As designed, the tour improvement algorithm has only one plan under consideration and heuristically selects the edges to swap. If instead, the algorithm was extended to allow the planner to choose between improving the current plan and generating a new random tour, the planner would have a decision point analogous to the plan generation decision for the other planners. Similarly, if the planner could choose between performing an iteration of two-opt and an iteration of three-opt or a random swap, the planner would have a need for refinement guiding. But, adding these choices complicates the planner, making a mathematical model of the planner more difficult. Using the tour planner as proposed by Boddy allows us to focus on the decision of when to begin execution. The tradeoff is between finding a better tour and starting execution. This tradeoff is the topic of Chapter 8.

5.7 Planner Summary

The meta-level control question for each of the four planners is summarized in table 5.2. The plan generation problem, to be covered in Chapter 6, is common to each of the planners, except the robot-courier tour planner. It involves deciding whether to generate another plan or refine a partial plan. This decision is not applicable to the tour planner because the tour planner considers only a single plan at a time and does not attempt to search the entire space of possible plans. The tour planner is not complete in the sense that it may never find the plan with the highest expected utility, even given an infinite amount of time.

Refinement guiding concerns selecting which part of a partial plan to refine. Refinement guiding is also applicable to all of the planners except the tour planner. The two-op algorithm includes a domain specific heuristic that selects the swap that causes the best improvement in the tour length. Refinement guiding is covered in Chapter 7. The questions of whether to optimize or satisfice and when to begin execution are applicable to all the planners and are the subject of Chapter 8.

	Xavier	Pyrrhus	DRIPS	Robot-Courier
Plan Generation Chapter 6	Generate another Plan? Refine a partial Plan?	Select plan to refine.	Select plan to expand.	NA Only one plan[4].
Refinement Guiding Chapter 7	Select contingency or abstract node to refine.	Select flaw to fix.	Select action in the plan to expand.	NA Heuristicly determined.[5].
Commencing Execution Chapter 8	Decide when to execution while continuing to make contingency plans.	Nothing (Optimizing)	Take into account limited accuracy and the time cost of computation.	Decide when to begin execution of the next step.
Contingency Planning Implicit	Select contingencies to plan for.	NA (Deterministic)	Implicit in abstraction hierarchy.	NA (Deterministic)
Information Gathering Future Work	Nothing	NA (Deterministic)	Implicit in abstraction hierarchy.	NA (Deterministic)
Quality of Estimates Future Work			Use only a subset of the chronicles to do sensitivity analysis.	Using internal state to estimate future performance.

Table 5.2: Summary of meta-level control questions for each planner.

Other meta-level control questions involving contingency planning and information gathering are covered only implicitly. Plans that include actions that gather information, such as medical tests, are generated if information gathering increases expected utility, but we do not address the question of when to gather information explicitly. Similarly, contingency planning is done only to the extent that it is needed to distinguish plans with high expected utility from plans with low expected utility. Work on these questions and questions related to the quality of estimates needed for effective meta-level control is left for the future.

[4]Applicable if the planner is extended to allow generation of new random tours.
[5]Applicable if the planner is extended to allow a choose between two-opt and three-opt.

Chapter 6

Plan Generation

This chapter examines the questions of which plans to generate and when to terminate plan generation. In the case of a generative planner, the question is how many times to call the planner to generate a new plan. In the case of a planner that generates abstract and/or partial plans, the question involves selecting which plan to elaborate in order to generate more detailed plans. We begin by giving a sequence of examples from the Xavier route planner to illustrate the nature of the choice being made. We then present three strategies for making the choice: selecting the plan with the highest upper bound on expected utility, selecting the plan with the highest lower bound on expected utility and selecting the plan with the lowest upper bound on expected utility. We show that selecting the plan with the highest upper bound on expected utility is optimal under some circumstances. We also show that this strategy has the best possible worst case performance for any strategy that uses bounds on expected utility. The DRIPS planner is then used to show how the theoretical results apply to a hierarchical refinement planner and infinite plan spaces. Empirical results using the DRIPS planner for a large medical domain are used to show how the three strategies perform in practice. Finally, we conclude the chapter by describing how the theoretical results can be applied to other decision-theoretic planners.

6.1 Xavier Plan Generation

Given a set of partial plans with ranges of expected utility and a limit on the expected utility of the remaining plans, the Xavier planner must decide whether to generate another plan or elaborate one of the existing plans. The answer to this question depends on the meta-level objective. Is the planner trying to minimize the amount of computation needed to find the plan with the highest expected utility or is it trying to maximize the overall utility by possibly adopting a sub-optimal solution that can be found quickly? The two objectives are related, but not the same. If the optimal solution can be found quickly enough, then finding and using the optimal solution also produces the highest overall expected utility. In this section, we discuss how to find the optimal solutions with a minimum of effort. The question of how to maximize overall utility, by possibly adopting a sub-optimal plan, is addressed in Chapter 8.

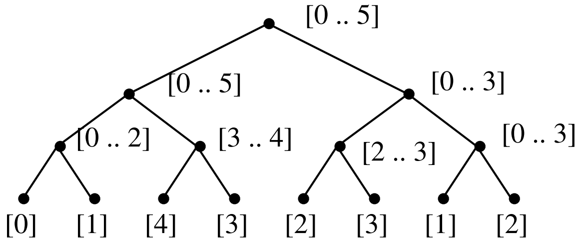

Figure 6.1: Representation of a search tree.

Rather than focusing only on the route planning problem, consider a generalization of the problem where we start with a single abstract plan that represents all possible plans and has a range of expected utility. The planner proceeds by refining the abstract plan into some number of more detailed sub-plans. Each of these sub-plans is either a completely elaborated primitive plan or a more refined abstract plan. The abstract sub-plans will have ranges of expected utility within the range of the parent plan whereas the primitive plans will have a point-value within the same range. The planner continues to select an unrefined abstract plan and refine it until the plan with the highest expected utility is found. Figure 6.1 shows a representation of the plan hierarchy. Each interior node represents an abstract plan and is labeled with the range of expected utility. Leaf nodes are completely elaborated plans. The state of the search in this space is represented by the set of nodes in the tree that have been generated. The problem is to search this tree and find the optimal solution while expanding a minimum number of nodes.

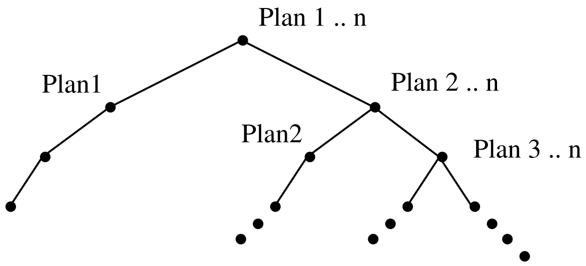

Figure 6.2: In the Xavier search tree, the set of ungenerated plans is a node that is expanded by generated a new partial plan. When a partial plan is refined, the result is a single, more refined plan.

The Xavier route planning problem can be mapped to this model by considering the set of ungenerated plans to be a single abstract plan. The bounds on the initial abstract plan are given by the upper bound on recovery costs, $2\sum length(corridor_i)/aveSpeed$ and the straight line distance to the goal divided by the speed of the robot, $distance/speed$. The abstract plan is refined by generating a new route that produces a new plan with a nominal route and a new abstract plan, representing the plans remaining to be generated. The lower bound on route length for the new abstract plan is the nominal length of the new route divided by the average speed of the robot, since this is limit on how good any of the remaining plans can be. The upper bound on length is again twice the length of all the corridors. The reason is that the robot could completely traverse a spanning tree, routed at its current location, and visit every possible location by traversing each corridor two times[1]. The new plan can be refined by planning for contingencies and the new abstract plan can be refined by generating a new route. Eventually, the tree bottoms out when there are no more routes and each plan has been completely refined. Figure 6.2 shows an Xavier route planner search tree graphically.

6.1.1 Strategies

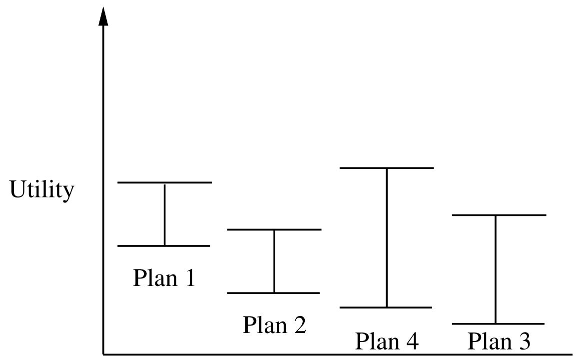

Figure 6.3: Plans selected by each of the four strategies.

Casting the plan generation problem for the Xavier route planner as a search through a tree of abstract plans allows us to ignore irrelevant details and focus on issues that are common to decision-theoretic planners. In this section, we present four general strategies for controlling search in the trees of abstract plans when plans are characterized by ranges of expected utility. We discuss the rationale used to support each of these strategies. In the next section, we will prove that the strategy of always selecting the plan with the highest upper bound on expected utility produces the best possible guarantee on the amount of work required to find a plan with the highest expected utility.

Strategies for selecting which plan to refine either focus effort on pruning the search space or on exploring parts of the search space likely to contain plans with high expected utility. One approach is to select the plan with the greatest lower bound on expected utility, $max(\underline{EU})$. In figure 6.3, this strategy corresponds to selecting plan 1. There are two lines of argument that support this choice. The first is that plan 1 gives the best guarantee on the minimum utility. It minimizes the maximum regret in the difference between the expected utility that it achieves and the expected utility that is possible[2]. This argument is more applicable when trying to find a nearly optimal solution with minimal effort, but not when

[1] For details see section 5.3.

[2] See [Loomes and Sugden, 1982, Loomes, 1987] for information on regret theory, an alternative to maximizing expected utility. Also see section 4.3.1.

Strategy	Plan Selected	Argument
1 $max(\underline{EU})$	plan 1	Raise the best lower bound to prune other plans.
2 $min(\overline{EU})$	plan 2	Try to prune nearly pruned plans.
3 $max(\overline{EU})$	plan 3	Focus on potentially optimal plans.
4 $min(\underline{EU})$	plan 4	No rationale.

Table 6.1: Plans selection strategies.

looking for a plan with the highest expected utility. The second argument to support the choice of plan 1 has to do with the likelihood of pruning other plans. If the lower bound on plan 1 can be raised slightly, then plan 2 can be pruned, reducing the search space. Although this is true, the effort expended to refine plan 1 is not necessarily needed to find the optimal solution. This strategy was originally used in the DRIPS planner and its implications are discussed as part of the empirical results in section 6.5.1.

A related strategy, which also attempts to improve pruning, prefers refining "nearly pruned" plans, like plan 2 in figure 6.3. It selects plans with the minimal upper bound on expected utility, $min(\overline{EU})$. The argument is that plan 2 can almost be pruned and a small amount of work will likely lower the upper bound on expected utility enough to allow the plan to be pruned. The problem with this strategy is that it expends effort on plans that are probably sub-optimal. It also suffers when there are many nearly pruned plans equivalent to plan 2. Rather than raising the lower bound on a plan like plan 1 that could prune them all at once, the strategy expends effort on each plan to prune it.

A third strategy takes an optimistic view and works only on plans with the greatest upper bound on expected utility, $max(\overline{EU})$. The idea is to concentrate the effort on the potentially optimal plans rather than trying to prune the search space. This optimistic strategy is not particularly applicable when looking for nearly optimal solutions using a minimum of computation. Selecting plan 4 in figure 6.3 for refinement can lead to plans that are less optimal than either plan 1 or plan 2. However, using the optimistic strategy when looking for a plan with the highest expected utility turns out to be the best strategy. The next section presents the proof of this result and the bounds on worst case performance.

The three strategies and their arguments are summarized in table 6.1. A fourth strategy that selects the potentially worst plan, the one with the minimum lower bound on expected utility, $min(\underline{EU})$, is included for completeness. This strategy would neither focus effort on pruning the search space nor on potentially optimal plans, and thus there is no rationale for choosing to work on such plans.

6.2 Proofs of Optimality

This section presents and proves two theorems on the optimality of the optimistic plan selection strategy. We show that, when looking for all plans with the highest expected utility, the optimistic strategy is optimal. This result does not hold when looking only for a single plan with the highest expected utility. A lucky guess when breaking ties between abstract plans that have the same upper bound on expected utility as a primitive plan with the highest expected utility, can lead to less work. For this to happen, the parent nodes of primitive plans with the highest expected utility have to have an upper bound on expected utility that is tight. Often, bounds are not tight, but can be arbitrarily close. Under this restriction, the optimistic strategy is again optimal. We then give the worst case performance of the optimistic strategy in the unrestricted case and show that all possible strategies have, at best, the same worst case performance. We show that the strategies that focus on pruning the search space can be arbitrarily worse than the optimistic strategy, and for infinite search trees, can fail to terminate.

6.2.1 Optimal Strategy for Finding All Optimal Plans.

Theorem 1 *Selecting the plan with the greatest upper bound on expected utility for expansion expands the fewest nodes when finding all plans with the highest expected utility.*

Proof: The proof relies on showing that every abstract plan that, at some point in the search, has a maximal upper bound on expected utility must be refined. Since the method expands only the plans that are required to be expanded, the method expands the fewest number of nodes. The proof that such plans need to be expanded can be shown by contradiction.

1. The optimistic strategy, $\mathcal{O}$, refines plans in a monotonically non-increasing order of the upper bound on expected utility, $\overline{EU}$.
   - The abstract plans at the frontier of the search are sorted by $\overline{EU}$, and the plans with the highest $\overline{EU}$ are refined first. [By definition of the optimistic strategy.]

     It is not possible for a plan at the frontier of the search to have a higher $\overline{EU}$ than the next plan to be refined, since they are sorted.
   - No node beyond the frontier of the search has an $\overline{EU}$ higher than the next node to be refined.

     Any node beyond the frontier of the search is a descendent of one of the nodes at the frontier.

     All descendents of a node have $\overline{EU}$ less than or equal to the $\overline{EU}$ for all ancestors. (Otherwise, the bounds are not valid.)
   - Therefore, no unrefined plan can have a higher $\overline{EU}$ than the next plan to be refined and plans are refined in monotonically non-increasing order of $\overline{EU}$.
2. All plans with $\overline{EU} \geq \overline{EU}_{optimal}$ must be expanded in order to show that all optimal plans have been found.
   - Suppose it were possible to determine that there were no more optimal plans without expanding an abstract plan with an $\overline{EU} \geq \overline{EU}_{optimal}$. This is a contradiction since the unexpanded plan could include an optimal refinement.

3. Any plans with $\overline{EU} < \overline{EU}_{optimal}$ are pruned by $\mathcal{O}$ when a plan with $EU_{optimal}$ is found. This happens before any plans with $\overline{EU} < \overline{EU}_{optimal}$ are refined since plans are refined in monotonically non-increasing order of $\overline{EU}$.

4. Since the $\mathcal{O}$ strategy refines only the set of plans that are required to be refined, it refines the fewest total nodes.

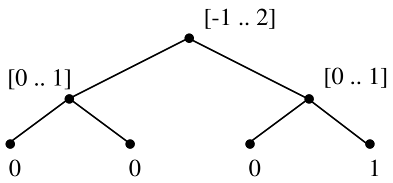

Figure 6.4: If the right sub-tree is refined, an optimal solution is found without refining the left sub-tree. If the left sub-tree is refined, then the right sub-tree must also be refined.

Generally, a planner has to find only one plan with the highest expected utility, since executing one such plan will achieve its objectives. When looking for a single plan with the highest expected utility, the optimality of the optimistic strategy has problems with the boundary cases where multiple abstract plans have the same greatest upper bound on expected utility equal to $\overline{EU}_{optimal}$. Figure 6.4 shows a search tree where the two children of the root node are abstract plans with the same upper bound on expected utility. One of the abstract plans has a primitive sub-plan with the same upper bound on expected utility, but the other one does not. If the search method guesses correctly, it can expand the right sub-tree and determine it has a plan with the highest expected utility without expanding the left sub-tree. If the search method guesses incorrectly, then it has to expand both sub-trees and do more work.

The boundary case that causes problems for the optimistic plan selection method arises only when the parent node of a plan with the highest expected utility has an upper bound on expected utility that is tight. If the right sub-tree in figure 6.4 had an upper bound that was greater than 1, then the optimistic strategy would be optimal, independent of the value of the upper bound on the left sub-tree. If the left sub-tree had an upper bound greater than 1, it would have to be refined before the planner could conclude that it had a plan with the highest expected utility. Both sub-trees would have to be expanded and the order would not make a difference. If the left sub-tree had an upper bound equal to or less than 1, the right sub-tree would be expanded first. The best primitive plan would be found and the left sub-tree would not have to be expanded.

There are two questions we need to address: Are the upper bounds on expected utility for ancestors of a plan with the highest expected utility ever tight and how big a difference can lucky guesses make. We look first at the question of tight bounds and argue that, for some planners, the expected utility bounds on partially refined plans are never tight. For such planners, the optimistic strategy is optimal even when looking for a single plan with the highest expected utility. Without this restriction, we can still show that the difference between the optimistic strategy and lucky guessing is bounded. Such is not the case for the other plan selection strategies from section 6.1.1, and we show search trees where these strategies can fail to terminate.

6.2.2 Optimal Strategy in Restricted Domains

In some domains, the upper bound on expected utility for abstract plans is never tight, and a fully refined plan will always have an expected utility that is less than the upper bound on its abstract parent. The expected utility bound on the parent can, however, be arbitrarily close to the real value. This situation arises in domains where refining an abstract plan resolves a tradeoff and the upper bound on the abstract plan was created by ignoring the tradeoff. A simple example would be a tradeoff between fuel consumption and travel time. The bound on the abstract plan would not be calculated using the optimal values for this tradeoff, since it could be too computationally expensive to compute. Instead, the bound would be calculated using the minimum amount of fuel and the minimum time, an outcome that is optimal, but unachievable. Refining the plan would resolve this tradeoff and would necessarily lower the upper bound on expected utility. The amount by which the bound is lowered could be arbitrarily small, depending, for example, on the length of the trip. Note that if you could show that the bound was always lowered by a minimum amount, the original bound could just be lowered by this amount.

It turns out that in domains where the upper bound on expected utility for abstract plans is never tight, but can be arbitrarily close, the optimistic strategy is still optimal.

Theorem 2 *If the upper bound on expected utility for abstract plans is never the same as the bound on the expected utility of a completely expanded plan, but can be arbitrarily close, then selecting the plan with the greatest upper bound on expected utility expands the fewest number of nodes when finding a single plan with the highest expected utility.*

Proof: The idea behind the restriction is to remove the possibility that a lucky guess can be confirmed to end the search. If there is a tie in the upper bounds on two abstract plans, there is no way to confirm that you have a plan with the highest expected utility by refining only one of them. Refining one of the plans will always give a lower expected utility for the primitive plan and the other abstract plan may contain a primitive plan that has slightly higher expected utility. The proof proceeds by showing that the optimistic strategy refines only the plans that are required to be refined in order to find the optimal solution.

1. The optimistic strategy refines plans in a monotonically non-increasing order of the upper bound on expected utility, $\overline{EU}$.

    - From the proof of theorem 1

2. The parent of an optimal primitive plan will be refined before any abstract plan with $\overline{EU} = \overline{EU}_{optimal}$

- The parent plan of an optimal plan must have $\overline{EU} > \overline{EU}_{optimal}$, since the bounds are not tight.
- Since plans are refined in monotonically non-increasing order of $\overline{EU}$ the parent plan must be refined before plans with $\overline{EU} = \overline{EU}_{optimal}$.

  This ensures that the optimal primitive plan is in the frontier set before any plans with $\overline{EU} = \overline{EU}_{optimal}$ are refined.

3. All plans with $\overline{EU} > \overline{EU}_{optimal}$ must be expanded in order show that the optimal plan has been found.

   - Suppose it were possible to determine the optimal plan without expanding an abstract plan with an $\overline{EU} > \overline{EU}_{optimal}$. Let the difference between the unrefined abstract plan and the upper bound on expected utility of the optimal plan be δEU. To conclude that the optimal plan has been found is to conclude that refining the abstract plan would reduce the upper bound on expected utility by at least δEU. But this contradicts the assumption that the bounds can be arbitrarily tight.

4. When the optimal primitive plan is in the frontier set, planning stops and only plans with $\overline{EU} > \overline{EU}_{optimal}$ have been refined.

   - The stopping criteria is part of the definition of the optimistic strategy. The fact that no plans with $\overline{EU} \leq \overline{EU}_{optimal}$ are refined follows from the fact that plans are refined in order and the optimal primitive plan will appear in the frontier set before any plans with $\overline{EU} \leq \overline{EU}_{optimal}$ are refined.

5. Since the strategy refines only the set of plans that are required to be refined, it does the minimum amount of computation.

6.2.3 Bounding the Advantage of Lucky Guesses

With no restrictions on the bounds of partially refined plans, the optimistic strategy can lead to expanding every abstract plan with an upper bound on expected utility that is greater than or equal to $\overline{EU}_{optimal}$. Although the plans with upper bounds that are greater than $\overline{EU}_{optimal}$ must be expanded in order to prove the optimal plan has been found, not all the plans with upper bounds equal to $\overline{EU}_{optimal}$ need be expanded. The planner needs to expand only direct ancestors of a plan with the highest expected utility.

If we modify the optimistic strategy to break ties such that nodes closest to the root of the search tree are refined first, then we can bound the difference between a lucky guess and our systematic strategy. It is also possible to show that any complete strategy has, at best, the same worst case performance. A complete strategy is one that is guaranteed to eventually find a primitive plan with the highest expected utility and show that it has the highest expected utility, whenever it is possible to do so.

Theorem 3 *In the worst case, the $\mathcal{O}$ strategy of selecting the plan with the greatest upper bound on expected utility for expansion and breaking ties in a FIFO order expands $O(B^n)$ more nodes than an optimal, omniscient strategy. Here B is the maximum branching factor and n is the minimum depth of a plan with the highest expected utility.*

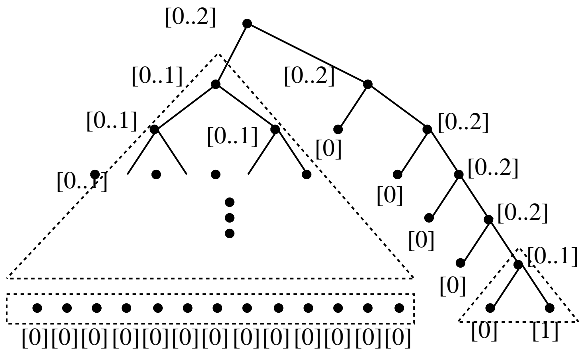

Figure 6.5: Worst case search tree.

Proof: To show that the worst case is at least $O(B^n)$, we need give only an example. To show that the bound is tight, we can show that the algorithm will always find the optimal plan in $O(B^n)$.

Figure 6.5 shows a worst case example for the optimistic strategy. The dashed triangles show the part of the tree that will be unrefined when all plans with $\overline{EU} > \overline{EU}_{optimal}$ have been refined. An optimal strategy would just refine the node at the lower right of the tree and find the optimal solution. The optimistic strategy will refine every other node before this node. The difference is:

$$\begin{aligned} O(optimistic) - O(omniscient) &= ((B-1)(B^{n-1}-1)+1)-1 \\ &= B^n - B^{n-1} - B + 1 + 1 - 1 \\ &= O(B^n) \end{aligned}$$

We now show that the maximum possible difference is $O(B^n)$.

1. Any strategy must expand all nodes with $\overline{EU} > \overline{EU}_{optimal}$, from the proof of theorem 2

2. When all the plans with $\overline{EU} > \overline{EU}_{optimal}$ have been expanded, either the optimal plan has been found, or there are one or more abstract plans with $\overline{EU} = \overline{EU}_{optimal}$.

   - Since all strategies must refine all plans with $\overline{EU} > \overline{EU}_{optimal}$, this point will be reached eventually. The optimistic strategy ensures that this point is reached before any plans with $\overline{EU} = \overline{EU}_{optimal}$ have been evaluated.

3. If the optimal plan has been found, then the optimistic strategy terminates, otherwise it does a breadth-first search through the plans with $\overline{EU} = \overline{EU}_{optimal}$.

- Since the nodes with equal $\overline{EU}$ are sorted by inverse depth, they will be explored breadth-first.

4. The number of node expansions to completely expand a tree using breadth-first search, to a depth of n, is bounded by $O(B^n)$, and will necessarily find any leaf node at depth n with $EU = EU_{optimal}$

5. In the best case, an optimal strategy could refine a single plan and find an optimal solution. $O(1)$

6. The worst case difference is $O(B^n) - O(1) = O(B^n)$

Given theorem 3, we know that the optimistic strategy can be $O(B^n)$ worse than optimal, but what about other strategies? It turns out that any strategy is $O(B^n)$ worse than an optimal strategy for some search trees.

Theorem 4 *For Every strategy there exists a tree such that the strategy expands $O(B^n)$ more nodes that an optimal omniscient strategy.*

Proof: An adversarial argument can be used to prove this theorem.

1. Given a search strategy, an adversary can determine the order that the leaves in the tree in figure 6.5 will be visited.

2. The adversary can then rearrange the labels on the leaves so that the plan with the highest expected utility is visited last.

3. For any arrangement of labels on the leaf nodes, an optimal omniscient strategy can find the plan with the highest expected utility in at most n expansions, where n is the depth of the plan with the highest expected utility.

4. The minimum difference is:

$$\begin{aligned} Min(\text{difference}) &= (B-1)(B^{n-1}-1)+1-n \\ &= B^n - B^{n-1} - B + 1 + 1 - n \\ &= O(B^n) \end{aligned}$$

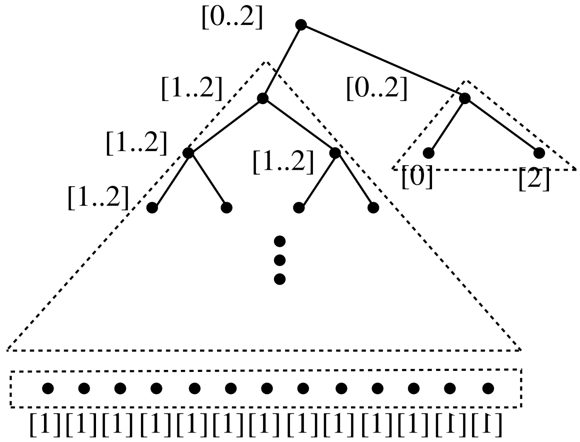

Figure 6.6: Worst case tree for selecting the plan with greatest lower bound on expected utility.

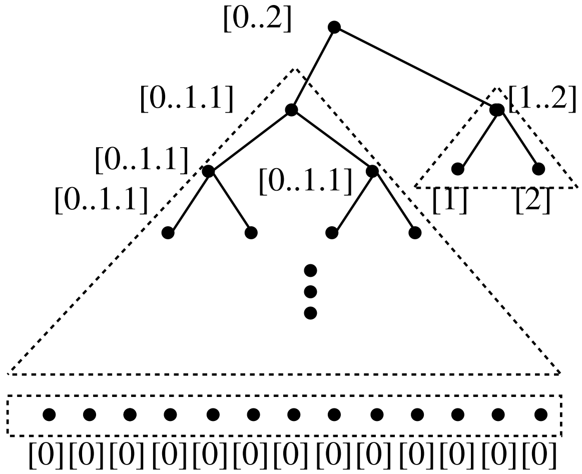

Figure 6.7: Worst case tree for selecting nearly pruned plans.

6.3 Worst Case Performance

We have shown that any complete strategy can be $O(B^n)$ worst than an optimal strategy for some trees. Obviously, an incomplete strategy, can fail to ever find the optimal plan. It is also interesting to note that a search tree may not be balanced or even finite. A strategy that performed depth-first search could be forced to search an arbitrarily large, or even an infinite space before finding the optimal solution, where a bread first search would terminate after $O(B^n)$ refinements.

Consider the first two strategies proposed in section 6.1.1, one that selects the plan with the greatest lower bound on expected utility and the other that selects the plan with the minimum lower bound on expected utility. Neither of these strategies systematically refines the plans with $\overline{EU} > \overline{EU}_{optimal}$ before expanding plans with $\overline{EU} = \overline{EU}_{optimal}$. In fact, selecting nearly pruned plans can expand plans with $\overline{EU} < \overline{EU}_{optimal}$ before plans with $\overline{EU} > \overline{EU}_{optimal}$. Using this observation, we can construct examples where these strategies will explore an arbitrarily large space before finding the optimal solution. For infinite search trees, we can construct examples where these strategies never terminate.

Consider the search tree in figure 6.6 where the optimal solution can be found by refining the top three nodes. The strategy of selecting the plan with the greatest lower bound on expected utility will completely expand the sub-tree on the left before refining the sub-tree on the right. The amount of work required is $O(B^{max(Depth)})$, which can be arbitrarily worse than $O(B^n)$. In fact, if the tree is infinite, then the maximum depth is also infinite and the planner will never terminate.

A similar example can be constructed for the strategy of selecting almost pruned plans for refinement. Figure 6.7 shows an example where an arbitrarily large or infinite amount of work is required before the planner finds the optimal plan. The worst case bounds for this strategy are the same as for the strategy of always selecting the plan with the greatest lower bound, $O(B^{max(depth)})$.

6.4 Pyrrhus

The plan generation question for the Pyrrhus planner can be mapped to the search tree model used for the Xavier route planner. The root of the tree is the null plan, specifying only the start and goal states. The children of a node are created by adding actions, ordering constraints or variable bindings to the plan that the node represents. The plan generation decision is to select a partial plan to refine. When the selected plan is refined, its children are added to the search tree. The size of the search tree is limited by the constraints on resources, including time that the planner imposes.

Williams and Hanks originally proposed using heuristic search control for selecting the plan to refine [Williamson and Hanks, 1994]. They suggest that techniques from classical partial order planning will be applicable. After experimenting with a number of strategies, they settled on the optimistic strategy because it had the best performance[3]. As with the Xavier planner, the optimistic strategy is the preferred strategy. The proofs of optimality in section 6.2 also hold for the Pyrrhus planner.

6.5 DRIPS Plan Generation

In this section, we apply the theoretical results on plan selection from the previous sections to the DRIPS planner. As described in section 5.5, the DRIPS planner is a decision-theoretic, hierarchical-refinement, domain-independent planner, that is very different from the Xavier route planner and the Pyrrhus planner. However, it still has to address the same plan generation problem.

[3]Personal communications with Mike Williamson.

The theoretical results from the Xavier route planner apply directly to the DRIPS planner. The abstraction hierarchy defines the search tree and the planner expands the search tree only until it has found a plan with the highest expected utility. To augment our theoretical results, in this section we present an empirical evaluation of the plan selection strategies proposed in section 6.1.1.

6.5.1 Empirical Comparison of Strategies

To get an idea of how the plan selection strategies compare in practice, we modified the DRIPS planner so that it could use any of the first three strategies in table 6.1. We then tried each strategy on a sequence of problems from the DVT medical domain. The DRIPS planner includes an example domain for analyzing treatment policies for deep venous thrombosis (DVT) [Haddawy *et al.*, 1995]. The domain model is taken from a study in the medical literature that evaluated treatment policies, including repeated tests [Hillner *et al.*, 1992]. The domain consists of a model of how the disease can progress over time and the actions available to the doctor that include a number of tests and a treatment. The planning objective is to create a testing and treatment policy with the highest expected utility. The original encoding of the domain for the DRIPS planner allowed up to three tests before a decision to treat was made. We extended the domain to use loops so that the planner can determine how many tests are useful.

The set of problems from the DVT domain are generated by varying the cost of fatality. For low costs of fatality, the optimal plan is not to test or treat anyone. Very few plans need to be refined in order to determine the optimal plan. As the cost of fatality rises, the optimal plan includes more tests to improve the likelihood of detecting the disease and treats the patients with positive test results. Adding more tests increases the probability that all patients, including healthy patients, will suffer side effects from the tests, which can lead to death in some cases. As the cost of fatality rises, the number of tests that must be considered and the tradeoffs between not treating the disease and side-effects from multiple tests become more difficult. The planner must refine an ever increasing number of plans to determine the optimal solution, even in the best case.

The results of the experiment are shown in table 6.2. The number of plans generated and the CPU time are displayed for each strategy. The percentage differences from the optimistic strategy are also given for the other two strategies. From the results, we see that for low costs of fatality, all the strategies have the same performance and refine relatively few plans. The differences in CPU time for these problems are not significant. As the cost of fatality rises, the number of plans that must be expanded increases, giving the planner more opportunity to make bad plan selection choices. When the cost of fatality has reached \$650,000, the first strategy, $max(\underline{EU})$, which was originally used in DRIPS refines about 10% more plans and uses 10% more CPU time than the optimistic strategy. This extra effort is spent refining plans in an attempt to prune the search space rather than focusing on potentially optimal plans. Above a \$650,000 cost of fatality, this strategy crosses a threshold where it becomes exceedingly sub-optimal. For a cost of fatality of \$850,000, the planner runs for over 5 CPU days and still fails to find a solution when the optimal

Cost of Fatality in \$000	Strategy $max(\underline{EU})$ # Plans		Time		$min(\overline{EU})$ # Plans		Time		$max(\overline{EU})$ # Plans	Time
50	9	0%	5.0 sec	0%	9	0%	5.0 sec	0%	9	5.0
100	33	0%	44.7 sec	2%	39	18%	47.5 sec	9%	33	43.7 sec
150	45	0%	63.0 sec	2%	51	13%	66.0 sec	7%	45	62.0 sec
200	67	0%	116.6 sec	1%	73	12%	119.8 sec	3%	67	115.9 sec
300	157	0%	433.6 sec	1%	161	3%	433.5 sec	1%	157	429.8 sec
500	471	7%	34.7 min	6%	553	25%	41.8 min	28%	441	32.6 min
650	1025	10%	99.0 min	10%	1293	39%	128.9 min	43%	933	90.1 min
850	>10007	>222 %	>5.45 days	>1574%	8481	173%	22.85 hrs	175%	3105	8.31 hrs

Table 6.2: Results from the DVT domain comparing the number of plans generated and the run time for three strategies. Percentages are differences relative to the optimal strategy. Tests run under Allegro 4.2 on a Sun 4 Sparc.

strategy finds a solution in 8.31 CPU hours. The planner stopped after 5.45 days because the computer ran out of virtual memory.

To understand why the $max(\underline{EU})$ strategy does so poorly, consider how it interacts with the test loop in the DVT plan hierarchy. The strategy will first unroll the loop, because it is the only thing that can be done. This will create two new plans, one with a loop and one without. The one without the loop will have the $max(\underline{EU})$ because it does not have an infinite number of tests. The planner will select this plan for expansion and create one or more new plans. These new plans will have $\underline{EU}$ greater than that of the plan with the loop and will be expanded first. The plan with the loop will be expanded only when all the plans without the loop have been fully refined or pruned. This works well if the optimal plan is found with the first or second loop unrolling since the optimal plan will be found quickly and can be used to effectively prune the search space. However, when the optimal plan takes three or more unrollings to find, the strategy almost completely refines all plans with one of two tests before considering any plan with three tests. Even then, it focuses attention not on plans that are likely optimal, but on plans that are most refined, and thus have tighter lower bounds. Even when the optimal plan is found, the planner must still eliminate plans with 4 or more tests. Again, it will nearly completely refine all the plans with 4 tests before considering plans with 5 tests. Since the number of plans is exponential in the number of tests, the value of this strategy quickly plummets as the number of tests in the optimal solution increases from two to three. The $\underline{EU}$ is strategy was originally the default strategy used by the DRIPS planner.

The second strategy for plan selection, $min(\overline{EU})$, does worse than the $\underline{EU}$ or optimistic strategy for small costs of fatality. The relative performance continues to worsen as the cost of fatality and the difficulty of the planning problem increases. It does not have the same dramatic threshold effect that the $\underline{EU}$ strategy has, but the performance does degrade significantly. At \$850,000, this strategy refines 173% more plans and uses 175% more CPU time.

Although strategies $min(\overline{EU})$ and $max(\underline{EU})$ can be arbitrarily worse than $max(\overline{EU})$, in practice the differences are mitigated by other factors. One thing to note is that the choice is limited to plans that are potentially optimal. Plans with very low $min(\overline{EU})$ are pruned from the search space. Once an optimal plan has been found, plans with $min(\overline{EU})$ have $\overline{EU} > \overline{EU}_{optimal}$, or they would be pruned. These plans need to be refined anyway. The difference between $max(\overline{EU})$ and $min(\overline{EU})$ is limited by the $max(\underline{EU})$ and gradually improves as more nearly optimal plans are found. The $max(\underline{EU})$ strategy tends to be correlated with the $max(\overline{EU})$ strategy since plans with higher lower bounds also tend to have higher upper bounds.

The results in table 6.2 are consistent with our theoretical results and show that using the optimistic plan selection strategy can have a significant effect on performance. The optimistic strategy has now been adopted by the creators of the DRIPS planner as the default plan selection strategy.

6.6 Robot-Courier Tour Planning

The plan generation problem for the robot-courier tour planner is moot since the planner considers only a single plan at a time. Modifying the planner to allow it to generate random tours, would make the plan generation question much more interesting. If the planner could choose to generate a random plan, then the planner could move its search to another region of the plan search space. Subsequent use of the two-opt algorithms would hill climb to a maximum in the new region. Jumping to another region could help the planner escape a local minimum or a region where the slope is very flat. Extending the planner so that it can generate random tours raises the question of when it should use two-opt to improve a tour it has and when it should generate a new random tour. However, the nature of the question would be different because plan generation would be done via a stochastic process. Plans would not be generated in any particular order. Since the random plans could repeat in the sequence, it could take an arbitrarily long time to generate a complete set of plans.

Some heuristic methods for deciding when to generate a new plan for a stochastic plan generator have been developed for machine shop scheduling domains where plan improvement is done using simulated annealing[Nakakuki and Sadeh, 1994]. We leave a more formal analysis of meta-level control for plan generation to future work.

Chapter 7

Refinement Guiding

The results in the previous chapter suggest which plan to select for further refinement. The next problem to address is to decide which part of the selected plan to refine. In this chapter, we look at three planners that produce different types of partial plans and have different information available when deciding which part of a plan to refine. We begin with the Xavier route planner, where the possible refinements include planning for contingencies, like closed doors, and expanding an abstract route through a room or corridor. The sub-parts of a problem in this domain can be solved independently and joined together to produce a plan with the highest expected utility. Only the best solution to each sub-problem needs to be retained in the refined plan. For problems of this type, we will show that there is an optimal idealized algorithm for selecting refinements that can be approximated using sensitivity analysis. The Pyrrhus planner, on the other hand, is used to solve problems that are generally not separable into sub-problems that can be solved independently. When refining a plan by resolving a conflict, binding a variable, or selecting an operator, the planner cannot just keep the single best refinement. Instead, the planner needs to keep all the possible solutions to a particular sub-problem. Refining part of a plan results in a set of plans, one for each solution to the sub-problem. In order to select which refinement to do, the current implementation of the Pyrrhus planner performs each possible refinement, essentially doing a complete one ply expansion of the search tree, to create a set of sets of plans. The planner then evaluates each set of plans and heuristically selects one set and continues planning with it. Although it is certainly not very efficient, this approach acquires perfect information about the results of computations by actually performing all of them. The perfect information obtained is not sufficient to allow the optimal, idealized refinement selection algorithm to be used. However, we will show that a heuristic, found to be effective for the Pyrrhus planner, approximates our idealized algorithm. The DRIPS planner, like the Pyrrhus planner, is generally used on problems that are not separable. For this planner, plan evaluation is the most expensive part of planning, so evaluating all possible refinements at each stage and accepting the best one is far too computationally expensive. Instead, we use a sensitivity analysis to suggest the effect of a computation and perform only the one that appears best. Our theoretical results are supported with empirical results from each of the planners.

7.1 Idealized Algorithm

If a meta-level controller knew the result and duration of every possible refinement, then it could search for a sequence of refinements that found a plan with the highest expected utility with the least amount of computation. The obvious problem is that a meta-level controller does not have perfect information about the possible refinements. But, even if it did, the search for the best sequence of refinements would, in general, be prohibitively expensive for use in meta-level control. However, some planners and domains have properties that allow for a simpler idealized algorithm. In the next section, we describe a simplified idealized algorithm that is applicable when sub-parts of a problem can be solved independently. The Xavier route planner is such a planner and we will adapt the simplified idealized algorithm for use in this planner. Other decision-theoretic planners, such as the Pyrrhus planner and the DRIPS planner, are not restricted to domains where the sub-problems can be solved independently. These planners introduce complications that require a more complex idealized algorithms. As this chapter progresses, we introduce versions of the idealized algorithm needed to address these added complexities.

7.2 Idealized Algorithm for Refining Separable Plans

In this section, we present an idealized algorithm for selecting which refinement to do first when refining a partial plan where each sub-problem can be solved independently. The Xavier route planner falls into this category and, in the next section, we will show how to use a sensitivity analysis to approximate the algorithm for the Xavier route planner. The algorithm we present in this section is idealized because it relies on knowing the duration and effect of each possible refinement computation. This information could be obtained by performing each computation and measuring the effect and duration, but this would defeat the purpose of meta-level control, which is to do the minimal amount of computation. We begin by stating our assumptions and justifying them. We then give a description of the algorithm and an intuitive argument for why it is optimal. We then prove that the algorithm is optimal, in some circumstances, and prove some complexity results.

The problem we address here is how to select refinements to completely elaborate one plan with the highest expected utility and prove that the plan does in fact have the highest expected utility. To understand this problem, consider the set of plans represented in figure 7.1. Each plan has a range of expected utility and, unknown to the planner, an exact expected utility somewhere in this range, represented by the black dot. The objective of the planner is to do as little work as possible to fully refine an optimal plan, like the first one in the figure, and show that it is optimal. To show that a plan is optimal, the upper bound on every other plan must be made less than or equal to $EU_{optimal}$, the dashed line in the figure. The four plans in figure 7.1 represent the four cases we need to consider in our analysis.

In this section, we consider refinement selection only for partial plans where the sub-problems can be solved independently. If there is more than one way to refine a particular

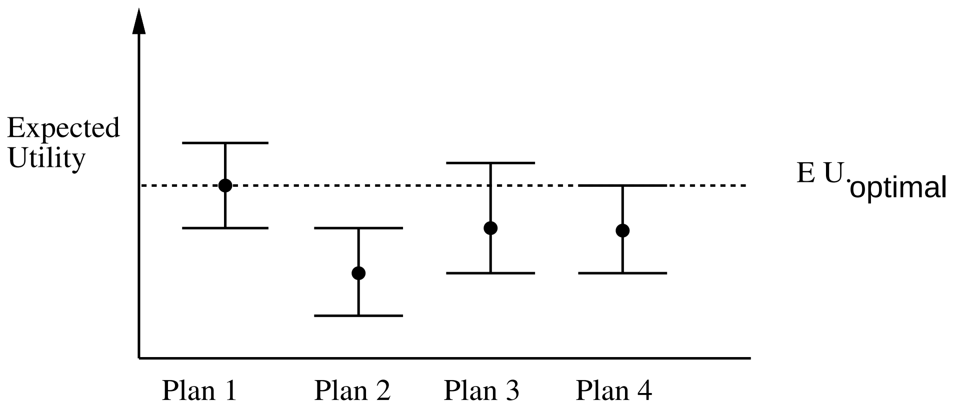

Figure 7.1: Plans can be characterized according to their relationship to $EU_{optimal}$.

part of a plan, our independence assumption permits us to retain only the best one. Refining a plan results in a single, more fully elaborated plan. Our independence assumption also makes the following assumption reasonable.

Assumption 1 *Permuting the order of a set of refinements does not affect the amount of computation needed.*

The idea is that we are selecting from a set of refinements that elaborate different parts of a plan. Since we can solve each sub-problem independently, the amount of computation needed to refine a particular part of a plan should not depend on the detail in the rest of the plan. Although this assumption is generally true for separable problems, it does not hold in every situation. It could be the case that the cost of evaluating the expected utility bounds on a plan depends on which parts have been refined. Refining a part of a plan that causes bounds evaluation to be expensive before other parts of a plan, could lead to a lower cost for evaluating the plan after each subsequent refinement. However, this type of affect should be small and the assumption is valid as a close approximation.

To begin our analysis, let's restrict our attention to the situation where the planner is looking for all the optimal plans. Obviously, the optimal plans must be fully refined and the order in which the refinements are done does not matter, given our assumption. The plans like plan 1 in figure 7.1 can thus be refined in any order. The remaining sub-optimal plans can be divided into three groups, which depend only on the upper bound on expected utility, $\overline{EU}$: plans with $\overline{EU} < EU_{optimal}$, plans with $\overline{EU} > EU_{optimal}$ and plans with $\overline{EU} = EU_{optimal}$. These groups correspond to plans 2 to 4 in figure 7.1. Plans in the first group (plan 2) will never be selected for refinement using the optimistic plan selection strategy, so we can ignore the problem of selecting refinements for them. Plans in the other two groups (class 3 and plan 4) must be expanded until $\overline{EU} < EU_{optimal}$, in order to prove that we have the full set of optimal plans. Changing the lower bounds on these plans will not prune them. To refine each of these plans efficiently, we choose the sequence of refinements that reduces $\overline{EU}$ to below $EU_{optimal}$ with the least amount of work. If each refinement takes the same amount of computation, this will be the shortest sequence. This choice of refinements can be done independently for each plan since the $\overline{EU}$ are independent and pruning depends only on the $\overline{EU}$ of each plan.

Theorem 5 *When finding all plans with the highest expected utility, given an optimistic plan selection strategy,* $\mathcal{O}$*, selecting the sequence of refinements that reduces* $\overline{EU} < EU_{optimal}$ *with the least amount of work and selecting any sequence of refinements for plans with the highest expected utility requires the least amount of work to completely refine the set of plans with the highest expected utility.*

Proof:

1. All optimal plans must be completely refined. The order does not matter, given our assumption.
2. All non-optimal plans must be refined until $\overline{EU} < EU_{optimal}$. Selecting the sequence that does this with the least amount of work is by definition optimal.
3. Only the minimum amount of work is done, so the process is optimal.

It is important to note that we refer to effects of sequences of refinements in theorem 5 and not the sum of the effects of a sequence of refinements. The reason is that utility might not be linear. Consider an example where there is a route consisting of two abstract nodes, each with a range of expected travel time of [10..100] seconds and actual travel times of 80 seconds. Suppose further that there is a hard deadline of 100 seconds. The utility function is positive and linearly decreasing with arrival time until the deadline. Arriving after the deadline has zero utility.

$$U(t) = 100 - t, t < 100, 0 \text{ otherwise}$$

Refining the first action reduces the range of utility from [0..80] to [0..10], a reduction on the upper bound of 70. Similarly, if the second action is refined instead, the reduction is also 70. However, performing both refinements results in a reduction of 80, which is much less than 70 + 70. The reason is that the refinement of each action is not utility independent. Each refinement reduces the range in travel time by the same amount, but utility does not decrease linearly with travel time after the deadline. Actions can also be resource dependent.

Not all domains have utility and resource dependence between refinements. In the Xavier route planning domain, the refinements are resource and utility independent. The reduction in $\overline{EU}$ for a sequence of refinements equals the sum of the $\overline{EU}$ changes for the individual refinements. To find an optimal sequence, simply find a set of refinements whose individual effects on $\overline{EU}$ sum to $\geq \overline{EU} - EU_{optimal}$ and has the minimum amount of work. Unfortunately, the integer packing problem can be reduced to this problem, meaning that the problem is NP complete. However, this problem and the integer packing problem has an approximate solution that is at worst two times the optimal solution [Garey and Johnson, 1979]. Simply use a greedy algorithm that selects refinements with the largest ratio of $\triangle\overline{EU}$ to work. If the work for each refinement is equal, then the packing problem becomes trivial and the greedy algorithm, which simply selects the refinement with the largest $\triangle\overline{EU}$, is optimal.

7.2.1 Finding One Plan with the Highest Expected Utility

The optimality of the idealized algorithm does not hold when looking for only a single plan with the highest expected utility. Again the problem is partial plans with $\overline{EU} = EU_{optimal}$. Suppose an abstract plan has $\overline{EU} > EU_{optimal}$ and there are two possible refinements available. The first one reduces $\overline{EU}$ to exactly $EU_{optimal}$. The second refinement reduces $\overline{EU}$ to less than $EU_{optimal}$, but requires more computation than the first refinement. The refinement to choose depends on whether the plan will be re-selected for refinement. If the second refinement is selected, then the plan will never be re-selected for refinement because it will never have the highest upper bound on expected utility. If the first refinement is selected, then the plan may be re-selected because it can be tied for the highest upper bound on expected utility with an optimal plan. In the worst case, if we select the refinement with the larger effect when we could have selected the less expensive refinement, the extra work required is equal to the difference between the work required for each refinement. On the other hand, the worst case for selecting the smaller refinement when the plan is again selected for refinement requires the planner to do both refinements and uses extra work equal to that of the smaller refinement.

As we did for the plan selection problem, we can restrict the relationship between bounds on expected utility for abstract plans and their primitive refinements. If we use the same restriction as in section 6.2.1, then the planner never has to refine plans with $\overline{EU} = EU_{optimal}$.

Theorem 6 *Given an optimistic plan selection strategy, $\mathcal{O}$, and bounds on expected utility for partial plans that are arbitrarily close but not equal to the expected utility of a fully elaborated plan, then selecting the sequence of refinements that give $\triangle\overline{EU} \geq \overline{EU} - EU_{optimal}$ with the minimum amount of work and selecting any sequence of refinements for plans with the highest expected utility requires the least amount of work to completely refine the set of plans with the highest expected utility.*

Proof: Same as for theorem 5, except that plans with $\overline{EU} = EU_{optimal}$ are never refined, according to theorem 2 in section 6.2.2.

In the unrestricted case, we need to worry about refining plans with $\overline{EU} = EU_{optimal}$ and the fact that a single refinement may not produce any change in $\overline{EU}$. Since the changes in $\overline{EU}$ are not necessarily additive, a sequence of changes may produce a change equal to the largest change due to a single refinement. Let a sequence S1 produce a change $\triangle\overline{EU} = \overline{EU} - EU_{optimal}$ with minimal work, and a second sequence S2 produce a change $\triangle\overline{EU} > \overline{EU} - EU_{optimal}$ with minimal work. Selecting S1 gives a worst case extra work equal to the work needed for S1, work(S1), whereas selecting S2 gives a worst case extra work equal to the difference between the work needed for S2 and the work needed for S1, work(S2) - work(S1).

7.3 Xavier Route Planner Refinement Selection

In this section we show how to approximate the idealized algorithm and apply it to the Xavier Route planning problem. We then present some empirical results to demonstrate the effectiveness of our approach.

7.3.1 Approximation using Sensitivity Analysis

The two obvious problems with the idealized algorithm are that it requires knowing the amount by which $\overline{EU}$ will be reduced by a sequence of refinements and the amount of computation time needed. To approximate the idealized algorithm, we use a greedy algorithm that selects refinements with the highest ratio of change in $\overline{EU}$ to computation time. To estimate this ratio, we use a sensitivity analysis.

In this section, we review two approaches to performing the required sensitivity analysis, one that uses the derivative of the utility function with respect to computation and the other that uses the utility function and an estimate of the duration of a refinement. The choice of which method to use depends on the information available and the type of computation. For the Xavier planner, the method that uses the utility function and an estimate of the required work is most appropriate.

If a formula for $\overline{EU}$ as a function of computation time for each refinement were available, then we would simply take the partial derivative to get the ratio we need. For most refinement algorithms, this formula will not be available and certainly not in a closed form that could be differentiated. Instead, we could determine the relationship between $\overline{EU}$ and computation for each refinement empirically, plot the results, fit a line and find its slope. This is the basic idea behind creating performance curves. The problem with this approach for refinement guiding is that it may require a performance curve for each refinement[1]. A way around this problem would be to create performance curves parameterized on the characteristics of a refinement.

An alternative approach is to calculate the possible change in $\overline{EU}$ and estimate the computation time required separately and then divide to get the required ratio. The utility function can be used to find $\triangle\overline{EU}$ and a model of the planner can be used to estimate the computation time. We illustrate the method using an example from the Xavier route planner.

Consider the example in figure 7.2. If $P(A)$ is the probability that door A is open and $P(\overline{A})$ is the probability that door A is closed then the utility of the plan is given by:

$$\begin{aligned} EU(plan1) &= -\{P(A)P(B)Time(plan1|A,B)+ \\ &\quad P(\overline{A})Recovery(plan1|\overline{A}) + \\ &\quad P(A)P(\overline{B})Recovery(plan1|A,\overline{B})\Big\} \\ Time(plan1|A,B) &= \sum_{i=0}^{4} Time(Node_i) + Time(corner(Node2, Node3)) \end{aligned}$$

[1] If the same curve were used for all refinements then it would give the same ratio for all refinements.

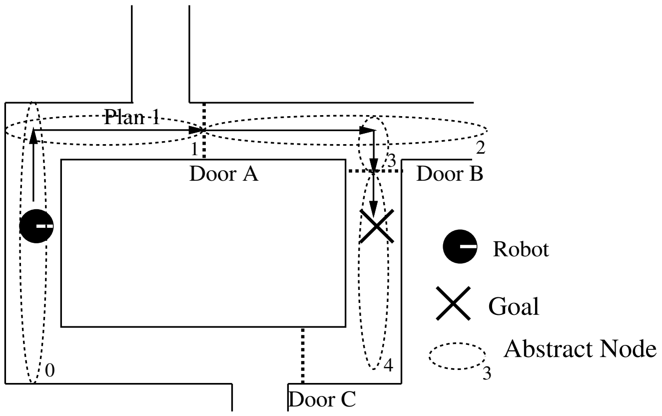

Figure 7.2: Xavier Example with doors

In the Xavier domain, the utility of a route is inversely proportional to the expected travel time. Plan 1 has a range of utility due to the range of distances for each abstract node and the unplanned contingencies. Planning the route through an abstract node reduces the range of travel time to a point. The effect on the range of expected utility is simply the range of travel time for the abstract node, weighted by the probability that the node is traversed. The potential effect on $\overline{EU}$ is the same as the potential reduction in the range of EU. The sensitivity for expanding an abstract node 2 (figure 7.2) is:

$$Sensitivity(Expand(Node_2)) \quad = \quad Range(Time(Node_2)) * P(A)$$

Similarly, the sensitivity for an unplanned contingency depends on the range of travel times for the contingency and the probability that the contingency plan is needed. The sensitivity for door B is:

$$Sensitivity(Recovery(B)) \quad = \quad Range(Time(Recovery(B)) * P(A) * P(\overline{B})$$

Note that the resulting sensitivity formulas for each refinement depend only on the range of time for the part of the plan the refinement affects. This is a consequence of the fact that the utility of a plan is simply the sum of the utilities for each sub part.

To select refinements, we divide the sensitivity of each refinement by the expected computation time and perform the refinement with the highest ratio. For the Xavier planner, each refinement requires one call to the A* search algorithm and each call requires approximately constant time. The reason each call takes about the same amount of time is that most of the searches for routes are relatively short and the overhead of setting up the search problem tends to dominate.

Using the sensitivity of $\overline{EU}$ to approximate $\triangle\overline{EU}$ is only a heuristic, since the real value is unknown. In fact, the sensitivity gives the maximum change possible, and not the expected change. However, the sensitivity can provide an estimator of the mean change.

	Refinements		Time	
Sensitivity Analysis	12.8333	0.0%	0.129584	0.0%
Heuristic	13.5161	+5.3%	0.146171	+12.8%
Random	14.2083	+10.7%	0.148478	+14.6%

Table 7.1: Average performance of the Xavier route planner on 9890 routes on the fifth floor of Wean hall at CMU.

Suppose the actual $\triangle\overline{EU}$ is chosen from a fixed distribution that is scaled to cover the range of possible refinement values. Let the distribution be defined over the range [0..1]. The minimum change is zero, otherwise the original bounds should be lowered by the amount of the minimum change. To cover the range of possible values, the distribution is scaled by $\triangle\overline{EU}_{max}$. The mean of the distribution is also scaled by the same factor. This suggests that $\triangle\overline{EU}_{max}$ should be proportional to the mean $\triangle\overline{EU}$ and therefore useful for selecting refinements.

Theorem 7 *If the actual change in $\overline{EU}$ is drawn from a fixed, non-degenerate distribution scaled to the range of allowed values, then $\triangle\overline{EU}_{max}$ is proportional to the expected change in $\overline{EU}$.*

Proof: Simply follows from the properties of scaling a probability distribution.

7.3.2 Empirical Results

To evaluate the effectiveness of our sensitivity analysis based meta-level control strategy for the Xavier domain, we compared its performance against a domain specific heuristic and a strategy that randomly selected refinements. The heuristic strategy simply traverses the plan depth-first and selects the first abstract node or unplanned contingency that it encounters. This approach tends to select refinements along the nominal route of a plan, closest to the robots start location. In some ways, it approximates the sensitivity analyze approach. In figure 7.2, if both doors had the same probability of being closed, then both the default heuristic and the sensitivity analysis method would choose to make the contingency plan for Door A before one for Door B. The random method randomly selects one of the available refinements.

We used the fifth floor of Wean Hall, where Xavier normally operates, as the environment to do our comparisons of the three refinement selection methods. The tests were performed on a Sun Sparc 5 computer using SunOS 4.1. The code was written in C and compiled using the gcc compiler. We created a test program that plans routes between each pair of distinct nodes in the fifth floor map, for a total of 9890 routes. The results of our comparison are summarized in table 7.1. The table shows the average number of refinements performed and the average running time for each of the three methods. In terms of refinements, the sensitivity analysis method performs on average 5% fewer refinements than the heuristic method and 11% fewer refinements than the random strategy. In terms of running time, the sensitivity analysis method does even better. On average, it requires 13% less time than the heuristic and 15% less time than random selection. The sensitivity analysis method outperforms the other strategies even thought it incurs an average overhead of 11% to perform the sensitivity analysis. This overhead could be reduced by optimizing the sensitivity analysis code and by caching. Currently, the planner performs a full sensitivity analysis for the plan with the highest upper bound on expected utility before each refinement. The planner could cache the results from one sensitivity analysis and reuse the values for the parts of the plan that are not changed by a refinement. Caching sensitivity analysis values works in this domain because the problem is separable and refining one part of a plan does not affect the sensitivity of the expected utility bounds to refining another part of the plan.

7.4 Abstract plans with Multiple Sub-plans (Pyrrhus)

To this point in the chapter, we have considered only partial plans where each part of an abstract plan can be refined independently using local dominance to select only a single refinement. When local dominance does not lead to global dominance, multiple potentially optimal instantiations of each refinement have to be retained. Refining part of a plan leads to a set of more refined sub-plans. Selecting a particular abstract action or contingency to refine determines which set of sub-plans are created. The tradeoff is between the number of sub-plans created and the amount by which $\overline{EU}$ is lowered. Creating many sub-plans can lead to more detailed plans with tighter bounds, promoting pruning. It can also lead to a large number of plans, each of which must be refined before it can be pruned. In this section, we examine how to make this tradeoff.

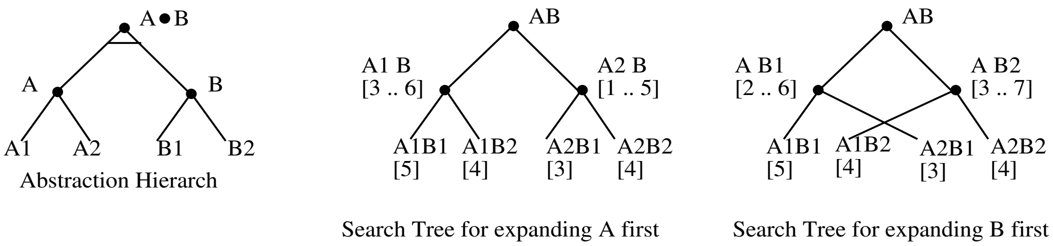

Figure 7.3: Order of abstraction refinement determines which tree is searched.

Selecting a particular refinement can be viewed as selecting the search tree that the plan selection strategy must search. Consider the example shown in figure 7.3. On the left is the abstraction hierarchy. A macro action, $A \bullet B$, consists of a sequence of abstract action A followed by abstract action B. Actions A and B each have two instantiations, A1, A2 and B1, B2 respectively. The question is whether to refine action A or B first. The search trees on the left of figure 7.3 show the two possible search trees. In the left most search tree, corresponding to refining action A first, only two of the interior nodes have

$\overline{EU} \geq EU_{optimal}$ and need to be refined. In the other search tree, all the interior nodes have $\overline{EU} > EU_{optimal}$ and need to be refined.

The objective of a good refinement selection strategy is to pick a search tree that the plan selection strategy can search efficiently. In the case of the optimistic plan selection strategy, this amounts to selecting a tree such that the number of nodes with $\overline{EU} > EU_{optimal}$ plus the number of nodes with $\overline{EU} = EU_{optimal}$, down to the depth of the first optimal solution is a minimum. The branching factor affects the number of nodes in the tree at a given depth and the amount by which $\overline{EU}$ is lowered on each refinement affects the depth to which the tree must be searched.

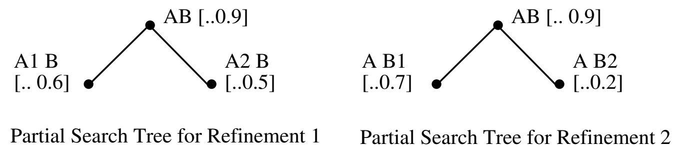

Figure 7.4: Refinement selection affects the distribution of expected utility values of the sub-plans.

In addition to the tradeoff between branching factor and changes in $\overline{EU}$, we need to consider the distribution of $\overline{EU}$ in the set of sub-plans created. Consider the problem proposed by Williamson and Hanks [Williamson and Hanks, 1996] and illustrated in figure 7.4. Refinement 1 leads to two sub-plans with moderate and nearly equal $\overline{EU}$. The second refinement leads to two sub-plans with vastly different $\overline{EU}$, one that is much better than either of the two sub-plans produced by refinement 1 and the other that is somewhat worse. Given that we know the bounds that each refinement would produce, but not the number of subsequent refinements needed or the exact value of $EU_{optimal}$, which refinement is better?

To answer this question, we propose creating a simple model to estimate the performance of the planner on the rest of the problem. We need to model the amount of work needed to lower the $\overline{EU}$ on the sub-plans to be less than or equal to $EU_{optimal}$, even though we do not know $EU_{optimal}$ when making the decision. To simplify the model, we assume that each refinement has a constant cost and that the expected change in $\overline{EU}$ for each refinement is also constant. These are both only first order approximations. We let the average branching factor be B and the amount that each refinement lowers $\overline{EU}$ be δ. Then the work required to lower the upper bound on expected utility $\triangle\overline{EU} \propto B^{\triangle\overline{EU}/\delta}$. To eliminate a plan, the $\overline{EU}$ must be lowered to $EU_{optimal}$, and the estimated work is $= B^{(\overline{EU}-EU_{optimal})/\delta}$. Given the choice between two possible refinements, we select the one with the least amount of expected work, $\sum_{i=1}^{n} B^{(\overline{EU}_i-EU_{optimal})/\delta} = B^{(-EU_{optimal})/\delta}\sum_{i=1}^{n} B^{(\overline{EU}_i)/\delta}$, where n is the number of sub-plan created. Since the $B^{(-EU_{optimal})/\delta}$ term is common to all estimates, we can eliminate it and the need to know or estimate $EU_{optimal}$. The branching factor, B, can be estimated from the abstraction hierarchy or on-line by keeping track of the average number of sub-plans that each refinement creates. The expected reduction in $\overline{EU}$, δ, can also be estimated on the fly or from previous examples.

Consider again, the Williamson and Hanks example. The branching factor for both refinements is 2, so we will assume that this is characteristic of the domain and use $B = 2$.

Estimating δ from this limited example is more problematic since not all the changes in $\overline{EU}$ are the same. As our estimate of δ, we use the average change in $\overline{EU}$, which is $((9-6)+(9-5)+(9-7)+(9-2))/4 = 4$. The expected work for refinement 1 is $2^{6/4} + 2^{5/4} = 5.2$ and the expected work for refinement 2 is $2^{7/4} + 2^{2/4} = 4.77$. This result suggests that adopting refinement 2 should require less work. Williamson and Hanks reach the same conclusion using their "sum of upper bounds on value", (SUBV), heuristic that selects the refinement with the lowest $\sum \overline{EU}$. The sum for refinement 1 is 1.1 and for refinement 2 is 0.9.

The SUBV heuristic can be viewed as an approximation of our method using expected work. Rather than summing $\overline{EU}$, divided by δ and exponentiated by B, the heuristic simply sums the values. This eliminates the need to estimate δ and B. The two formulations are somewhat correlated and in some cases approximate each other.

$$\begin{aligned} work &= \sum_{i=1}^{n} B^{(\overline{EU})/\delta} \\ &\doteq nB^{average(\overline{EU})/\delta} \\ work_1 - work_2 &= n_1 B^{average(\overline{EU_1})/\delta} - n_2 B^{average(\overline{EU_2})/\delta} \end{aligned}$$

If $n_1 = n_2$ then

$$work_1 - work_2 = n\big(B^{average(\overline{EU_1})/\delta} - B^{average(\overline{EU_2})/\delta}\big)$$

Work1 is greater than work2 if $average(\overline{EU_1}) > average(\overline{EU_2})$. Since we assume that $n_1 = n_2$, this reduces to whether $\sum \overline{EU}_1 > \sum \overline{EU}_2$, the same as the SUBV heuristic.

Williamson and Hanks compare their heuristic to three heuristics taken from the partial-order, causal-link planning literature and a fourth heuristics designed to minimize the least upper bound on EU, "Least upper bound value", LUBV. The LUBV heuristic selects the refinement with the maximum change in $\triangle\overline{U}$, regardless of the number of plans created. The other three heuristics are designed to enable a planner to quickly find a solution to the problem, without regards to plan quality. Results from their experiments show that the SUBV heuristic does significantly better than the other four heuristics [Williamson and Hanks, 1996]. This is the expected result given our analysis that shows that the SUBV heuristic approximates the optimal strategy.

7.5 DRIPS Refinement Selection

The Pyrrhus planner relies on quickly performing all the available refinements and evaluating the resulting plans before it decides which refinement to keep. This method depends on computationally cheap refinement and evaluation. For some planners, plan evaluation is the most expensive part of planning and needs to be avoided as much as possible. This is especially true for probabilistic domains where a planner may have to determine a large number of possible outcomes and their probabilities in order to evaluate the expected utility of a plan. The Pyrrhus planner avoids this problem since it is restricted to deterministic domains.

In this section, we look at the refinement selection problem for the DRIPS planner, which finds optimal plans for probabilistic domains. Plan evaluation is expensive and must be avoided when possible. The planner also presents some complications and violates some of the assumptions used to analyze the Xavier and Pyrrhus planners.

The DRIPS planner evaluates a plan by projecting the initial state through the sequence of actions in the plan. Each action can have probabilistic effects that cause the chronicle to branch, one branch for each possible outcome of the action. The longer a sequence of actions, the more levels of branching and the more chronicles to evaluate. The evaluation of a chronicle may depend on the entire chronicle or certain sub-parts. For example, a maintenance goal where the utility depends on how close the temperature is to 20 degrees at each point in time would require evaluating every state in the chronicle On the other hand, goals of attainment, such as delivering a package to the post office by 5pm depends only on the final state of the chronicle. In general, the evaluations tend to be either constant or linear in the length of the chronicle. The number of chronicles, however, is exponential in the number of actions in the plan. If the average branching factor per action is B_{branch}, then the number of chronicles is $B_{branch}^{length(plan)}$. Refining an instance abstraction creates a set of plans with the same length. Expanding a macro action creates a single plan that is longer and will tend to have more chronicles than the original plan. Because different refinements can create plans with different lengths, and hence different expected evaluation times, we can no longer assume that all plan evaluations take the same amount of work. Different evaluation costs also violates the assumption that reordering a sequence of refinements does not make a difference in the total computation needed. Without these assumptions, estimating the work needed to refine an abstract action and the work needed to refine the resulting plans is more complex.

In the rest of this section, we show how to apply our search control techniques to the DRIPS planner. We begin by considering deterministic domains. We show how to calculate the sensitivity of $\overline{EU}$ to refinement of a particular abstract action and how to calculate the expected work needed to perform the refinement. The results are used to guide search in a simple beer brewing domain and the performance is compared to several simple heuristic methods. We then extend our approach to deal with probabilistic domains and show how it performs in a probabilistic medical domain.

7.5.1 Sensitivity Analysis for Deterministic Domains

In this section, we present a sensitivity analysis for a general utility model proposed by Haddawy and Hanks [Haddawy and Hanks, 1993] and used in the DRIPS planner. In this model, expected utility of a plan, p, is the sum of the utilities of the possible chronicles weighted by their probability.

$$\mathbf{EU}(p) = \sum_{c \in chronicles(p)} U(c) \cdot P(c)$$

The utility of each chronicle, $U(c)$, is the weighted sum of the utility of goal satisfaction, UG, and residual or resource utility, UR. The residual utility measures the value of non-goal related resources consumed or produced in a chronicle. These resources include things like fuel and money that are consumed to achieve a goal and as a result are unavailable for other tasks. The UR function is used to express the preference for conserving resources like fuel and money.

$$U(c) = \mathsf{UG}(c) + k_r \mathsf{UR}(c)$$

The utility of goal achievement is further broken down into temporal and atemporal components. The atemporal component gives the degree to which a particular state satisfies the goal, $DSA(c,t)$. It takes a chronicle and a point in time. The temporal component gives the degree to which the temporal component of the goal is satisfied. For deadline goals, it is a function only of time, $CT(t)$ and in the case of maintenance goals it is a function of time intervals $CP(i)$. The temporal and atemporal components are combined in different ways to express maintenance and deadline goals. For example, a deadline goal, with a deadline time of t_d is expressed using the formula:

$$\mathsf{UG}(c) = DSA(c,t_d) + \sum_{\{t>t_d : \neg\exists t'(t_d \le t' < t) \wedge DSA(c(t')) \ge DSA(c(t))\}} (DSA(c(t)) - \mathsf{max}_{t_d \le \tau < t} DSA(c(\tau)))\dot{C}T(t)$$

In the formula, $c(t)$ refers to the chronicle, c, up to time t. This formula rewards the degree to which the goal is accomplished at the deadline plus any increases in DSA after the deadline, discounted by the temporal degree of satisfaction, CT.

Maintenance goals depend on the degree to which the goal is satisfied over an interval. The formula for representing maintenance goals is:

$$\int_0^1 \sum_{\{I : \forall t \in I \mathsf{DSA}(t) \ge x \wedge \nexists I' \supset I \mathsf{DSA}(t) \ge x\}} CP(I)dx$$

This formula rewards intervals of interest when the DSA is maintained above each possible value. See [Haddawy and Hanks, 1993] for more details on how this can be used to express various forms of maintenance goals.

The DRIPS planner can handle both deadline and maintenance goals using its standard $\mathsf{UG}(c)$ functions. The remaining utility functions, DSA, CT and UR, form part of the problem description input to the planner. The sensitivity analysis requires two additional functions that give the possible change in the upper bound of the utility functions as a result of expanding an action. The $\triangle DSA^{+}(chronicle, action, plan)$ function returns the maximum change in the upper bound on utility of goal achievement for a chronicle if the given action in the given plan is expanded. A second function, $\triangle UR^{+}(chronicle, action, plan)$, similarly returns the maximum change in the upper bound on the residual utility. The $\triangle DSA^{+}$ and $\triangle UR^{+}$ functions can be derived from the DSA and UR functions respectively. However, since the DSA and UR functions can be arbitrarily complex lisp functions, the two additional functions must currently be supplied by the domain designer. To show what is involved in generating the required functions, we give a series of examples.

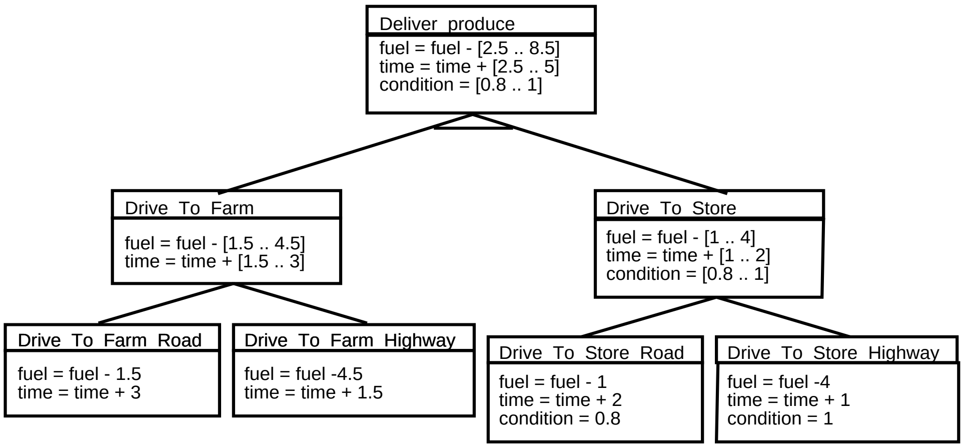

Figure 7.5: Abstraction leads to a reduced number of qualitatively different routes.

To illustrate how the sensitivity analysis is performed, we will analyze the simple delivery domain shown in figure 7.5. The plan is to drive a truck from a depot to the farm, pick up a trailer-load of goods and deliver them to the store. The truck can either drive on the interstate, which, although it is further, saves time because speeds are higher. The price that must be paid is higher fuel consumption. Also, the condition of the produce deteriorates while it is in transit to the store. The longer the trip, the more the deterioration. Figure 7.5 gives the action hierarchy. The top level "deliver_produce" is a macro action consisting of two instance actions "drive_to_farm" and "drive_to_store". The utility function for the domain is:

$$\begin{aligned}
\mathsf{UR}(c) &= 10 - \mathsf{fuel} \\
\mathsf{DSA}(c) &= 10 * \mathsf{condition} - 0.1 * \mathsf{time} \\
\mathsf{CT}(c) &= 1 \\
U(c) &= \mathsf{DSA}(c) * \mathsf{CT}(c) + 0.5\mathsf{UR}(c) \\
&= 10 * \mathsf{condition} - 0.1 * \mathsf{time} + 5 - 0.5\mathsf{fuel} \\
EU &= U(c)
\end{aligned}$$

Suppose the top macro action has been refined to create a plan with the two drive actions ("drive_to_farm", "drive_to_store"). The utility of the plan is:

$$\begin{aligned}
U(c) &= 10 * \mathsf{condition} - 0.1 * \mathsf{time} + 5 - 0.5\mathsf{fuel} \\
&= 2 * [0.8 \ldots 1.0] - [2.5 \ldots 5.0] + 5.0 - 0.5 * [2.5 \ldots 8.5] \\
&= [8.25 \ldots 13.5]
\end{aligned}$$

The planner now must select one of the instance actions for refinement. To use sensitivity analysis based refinement selection, the sensitivity of $\overline{EU}$ with respect to each action refinement must be calculated. This can be done using the structure of the actions and the utility function. Since the utility function for this domain is linear in the value of the attributes, we can generate the sensitivity functions using partial derivatives.

$$\begin{aligned}
\triangle\overline{\mathsf{UR}(c)}_{\mathsf{action}} &= \sum_{\mathsf{A:attributes}} \left|\frac{\partial \mathsf{UR}(c)}{\partial A}\right| \triangle A \\
&= \triangle\mathsf{fuel}_{\mathsf{action}} \\
\triangle\overline{\mathsf{DSA}(c)}_{\mathsf{action}} &= \sum_{\mathsf{A:attributes}} \left|\frac{\partial \mathsf{DSA}(c)}{\partial A}\right| \triangle A \\
&= 10 * \triangle\mathsf{condition}_{\mathsf{action}} + 0.1 * \triangle\mathsf{time}_{\mathsf{action}} \\
\triangle\overline{\mathsf{CT}(c)}_{\mathsf{action}} &= 0 \\
\triangle\overline{U(c)}_{\mathsf{action}} &= \triangle\overline{\mathsf{DSA}(c)}_{\mathsf{action}} + 0.5 \triangle \overline{\mathsf{UR}(c)}_{\mathsf{action}} \\
&= 10 * \triangle\mathsf{condition}_{\mathsf{action}} + 0.1 \triangle \mathsf{time}_{\mathsf{action}} + 0.5 * \triangle\mathsf{fuel}_{\mathsf{action}}
\end{aligned}$$

In the sensitivity formulas, $\triangle A$ refers to the range of attribute values assigned by the actions. For example, $\triangle\mathsf{fuel} = 3.0$ in the "drive_to_farm" action, since the effect of the action is $\mathsf{fuel} = \mathsf{fuel} - [1 \ldots 4]$. It is also important to note that the absolute value of each partial derivative is used in the sensitivity formulas. This is because the range of attribute values can be reduced by $\triangle A$ by either raising the lower bound of decreasing the upper bound. For resources attributes, like fuel, raising the lower bound decreases the upper bound on utility whereas the converse is true for desirable attributes like "condition". Using the absolute value of each derivative gives the desired contribution for both types of attributes.

$$\begin{aligned}
\triangle\overline{U(c)}_{Drive_to_Farm} &= 10 * 0 + 0.1 * 1.5 + 0.5 * 3 && (7.1) \\
&= 1.65 \\
\triangle\overline{U(c)}_{Drive_to_Store} &= 10 * 0.2 + 0.1 * 1 + 0.5 * 3 && (7.2) \\
&= 3.6
\end{aligned}$$

Using our sensitivity formula, we can calculate the sensitivity of $\overline{EU}$ to each of the two action refinements. The results, equations 7.2 and 7.3, suggest that $\overline{EU}$ is most sensitivity to refinement of the second action. When the first action is refined, the resulting plans have utility values of $[9.75 \ldots 13.35]$ and $[8.4 \ldots 12]$, an average reduction in $\overline{EU}$ of 0.825. Refining the second action results in plans that have utility values of $[8.25 \ldots 11.4]$ and $[10.5 \ldots 12]$, an average reduction of 1.8, which is more than twice the average reduction for refining the first action. This is the expected result given that the sensitivity of $\overline{EU}$ to refining of the second action is more than twice that for the first action.

The sensitivity analysis done for the delivery example relies on the fact that utility is a linear combination of functions of a single attribute and that each of these functions has a constant derivative with respect to the attribute. If each function did not have a constant derivative, but was simply monotonicity non-decreasing (or non-increasing), applying the derivative method would be problematic. One potential problem is that the function may not be continuously differentiable. However, even if it were possible to calculate the derivative, there would be the problem of which attribute values to use when evaluating the derivative. As an alternative to using the differential method, we propose using an interval arithmetic based method.

If we do not know the closed form of the utility function, but simply know that it is a weighted sum of monotonically non-decreasing (or non-increasing), functions, we can still calculate the effect that a particular refinement can have on the utility. For each attribute, we simply restrict either the upper or lower bound by $\triangle A$ and evaluate $\overline{U}(\mathsf{condition}, \mathsf{fuel}, \mathsf{time}) - \overline{U}(\mathsf{condition}', \mathsf{fuel}', \mathsf{time}')$. In the delivery example, the utility is a monotonically non-decreasing function of "condition" and monotonically non-increasing function of "fuel" and "time". The sensitivity of $\overline{EU}$ to refinement of the first action is

$$
\begin{aligned}
&\triangle\overline{U(c)}_{Drive\_to\_Farm} \\
&= \overline{U}([0.8\ldots1.0],[2.5\ldots8.5],[2.5\ldots5.0]) - \overline{U}([0.8\ldots1.0],[5.5\ldots8.5],[4.0\ldots5.0]) \\
&= 13.5 - 11.85 \\
&= 1.65 \\
&\triangle\overline{U(c)}_{Drive\_to\_Store} \\
&= \overline{U}([0.8\ldots1.0],[2.5\ldots8.5],[2.5\ldots5.0]) - \overline{U}([0.8\ldots0.8],[5.5\ldots8.5],[3.5\ldots5.0]) \\
&= 3.6
\end{aligned}
$$

The results, as expected, are the same as for the derivative based method. The advantage of the interval arithmetic method is that we do not need access to the form of the utility function in order to take the derivative. The only information required is whether the utility is monotonically increasing or decreasing in each attribute. Even this requirement can be relaxed at the expense of additional computation. Knowing whether a function is monotonically non-increasing or non-decreasing is used to decide whether to restrict the upper or lower bound on the attribute. If this information is unavailable, but we still know that the function is monotonic, we can restrict each bound in turn to see which restriction has the largest effect on $\overline{EU}$. The sensitivity is the sum of the larger changes for each attribute. Additional problems arise if the utility function is not a weighted sum of functions of a single attribute. If however, we know simply that U(c) is a monotonic function of each attribute, then we can simply try all combinations of bounds restrict on the attributes. The combination that lowers $\overline{EU}$ the most is subtracted from the original $\overline{EU}$ to get the sensitivity. This method involves 2^n evaluations of the utility function, where n is the number of attributes appearing in the utility function. In general, the number of attributes in the utility function is relatively small and the utility function is relatively inexpensive to compute. So, although the computation is exponential in the number of attributes, the overall cost is not prohibitive.

In the delivery example, if the utility function is separated into temporal and atemporal components and represented as an immediate soft deadline goal, the utility function becomes:

$$\begin{aligned}
\mathsf{UR}(c) &= 10 - \mathsf{fuel} \\
DSA(c,t) &= 10 * \mathsf{condition} \\
CT(t) &= 10 - 0.10 * \mathsf{time} \\
\mathsf{UG}(c) &= max_{t\in c}(DSA(c,t) \cdot CT(t)) \\
U(c) &= \mathsf{UG}(c) + 0.5\mathsf{UR}(c) \\
EU &= U(c)
\end{aligned}$$

With this formulation of the utility function, the sensitivity cannot be calculated on a per attribute basis and simply summed. The effects of changing "time" and "condition" must be multiplied rather than added. Also, taking the maximum value over the time range introduces a non-linearity where changing DSA at one point in time may have no effect on UG. Even with these complications, the example still maintains monotonicity of $U(C)$ with respect to changes in the attribute values. The sensitivity of $\overline{EU}$ can be found by evaluating $\mathsf{UG}(c)$ using the 4 possible combinations of restrictions on condition and time and using the one that produces the largest change. Since $U(c)$ is linear in $\mathsf{UG}(c)$ and $\mathsf{UR}(c)$, the results can be combined with the sensitivity of $\mathsf{UR}(c)$ using a weighted sum. If it is known that the utility function is monotonically increasing with increasing "condition" and monotonically decreasing with increased "time", then only the restriction that decreases the upper bound on "condition" and raises the lower bound on "time" needs to be evaluated, since it will be the one that produces the largest change in $\overline{EU}$.

The sensitivity analysis calculations get more difficult when the utility functions are non-monotonic. Simply restricting one of the bounds does not allow for any conclusion about the maximum possible effect of refining an abstract action. In such situations, either more knowledge about the utility function is required or weaker bounds must be calculated.

To illustrate the methods for dealing with discontinuously differentiable, non-monotonic utility functions, we use the beer brewing domain, which is one of the domains supplied with the DRIPS planner as an example. In this domain, the planner selects malts and grains to use in a sequence of steps in the beer brewing process. The grains and malts selected effect the sweetness, hoppyness, colour and alcohol content of the resulting beer. For example, a stout is a dark, malty beer with high alcohol whereas an Indian pale ale is light, hoppy and has moderate alcohol. The utility function for the beer brewing domain is a weighted sum of saw-tooth functions, where each saw-tooth function depends only on a single attribute and is centered around the desired value for that attribute. The weights for all the attributes are positive.

$$
\begin{aligned}
\mathsf{UR}(c) &= -cost \\
\mathsf{UG}(c) &= \sum_{A:Attribute} W_A * \mathsf{sawTooth}(A, \mathsf{low_A}, \mathsf{desired_A}, \mathsf{high_A}) \\
U(c) &= \mathsf{UG}(c) + \mathsf{UR}(c)
\end{aligned}
$$

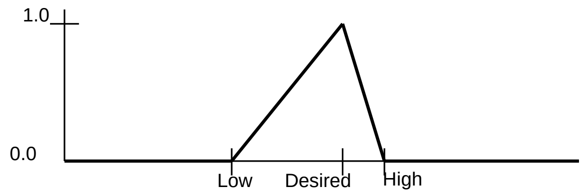

Figure 7.6: The saw-tooth function increases linearly from zero at the low value to one at the desired value and then decreases linearly to zero at the high value.

$$
\mathsf{sawTooth}(V, \mathsf{low}, \mathsf{desired}, \mathsf{high}) = \begin{cases} 0 & V < \mathsf{low} \\ (V - \mathsf{low})/(\mathsf{desired} - \mathsf{low}) & \mathsf{low} \leq V \leq \mathsf{desired} \\ (\mathsf{high} - V)/(\mathsf{high} - \mathsf{desired}) & \mathsf{desired} \leq V \leq \mathsf{high} \\ 0 & V > \mathsf{high} \end{cases}
$$

To calculate the sensitivity of $\overline{EU}$ to a particular action refinement, we need to know how much the upper bound on the saw-tooth functions can change for a given reduction in the range of an attribute value. Note that in this case, we cannot simply take the derivative of $\overline{EU}$ with respect to a particular attribute and multiply by the range. One problem is that there is no derivative defined at the low, desired and high values of the attribute. Secondly, even if we could determine a derivative at each point, there is a problem of selecting which value of the derivative to use. Outside of the "saw-tooth" region, the derivative is zero, whereas inside this region there are two distinct values corresponding to the up ramp and the down ramp.

Rather than trying to estimate the change in the saw-tooth function using derivatives and $\triangle A$, we use the saw-tooth function directly. Given the amount by which refining an action can affect an attribute $\triangle A$, we restrict the range of attribute values given to the saw-tooth function to find the possible effect of refining the action.

$$
\begin{aligned}
&\overline{\triangle \mathsf{sawTooth}}(A, \mathsf{low}, \mathsf{desired}, \mathsf{high}, \triangle A) = \\
&\quad \overline{\mathsf{sawTooth}}(A, \mathsf{low}, \mathsf{desired}, \mathsf{high}) - \\
&\quad min(\overline{\mathsf{sawTooth}}([A_l + \triangle A \ldots A_h], \mathsf{low}, \mathsf{desired}, \mathsf{high}), \overline{\mathsf{sawTooth}}([A_l \ldots A_h - \triangle A], \mathsf{low}, \mathsf{desired}, \mathsf{high}))
\end{aligned}
$$

Using this formula, the overall effect on $\overline{EU}$ can be determined by summing the effects of due to each attribute, multiplied by their respective weights.

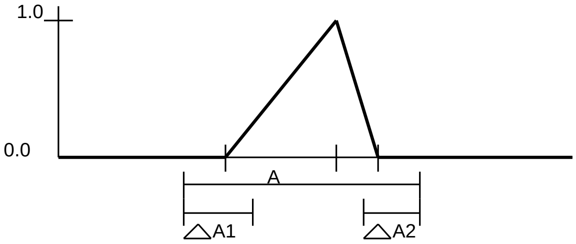

Figure 7.7: The lower bound is raised most when both the upper and lower limits of A are restricted.

$$\triangle\overline{EU} = \sum_{A:Attribute} W_A * \overline{\triangle\mathsf{sawTooth}}(A, \mathsf{low}_{\mathsf{A}}, \mathsf{desired}_{\mathsf{A}}, \mathsf{high}_A, \triangle A)$$

In this example, it is important that all the weights, W_A are positive. The reason is that positive weights mean that it is the upper bound on the saw-tooth function that affects $\overline{EU}$ rather than the lower bound on the saw-tooth function. Calculating the effect that a reduction in the range of an attribute has on the upper bound of the saw-tooth function is much easier than calculating the effect on the lower bound. The maximum affect on the upper bound happens when either the upper bound on the attribute value or the lower bound on the attribute value is restricted by the largest possible amount. In the case of the saw-tooth function, this is true despite the fact that the function is neither monotonically non-decreasing or non-increasing. It works because the function is convex. The single "lump" peaking at the desired value ensures that restricting either the upper bound or the lower bound by the maximum possible amount will produce the largest change in $\overline{EU}$. The same is not true for the lower bound. Consider the diagram in figure 7.7. The largest change in the lower bound occurs when both the upper and lower bound are restricted. The total amount of the restriction, $\triangle A_1 + \triangle A_2$, must add up to $\triangle A$, but the size of each one required to get the maximum change depends on the slopes of the two sides of the saw-tooth and the distance of each bound from the high and low values. Calculating the values for $\triangle A_1$ and $\triangle A_2$ that produce the largest change in the lower bound is not computationally intensive, but working out the formula is tedious and error prone. For other utility functions, there may not be a closed form solution to find $\triangle A_1$ and $\triangle A_2$. In these cases, we can find a bound on $\underline{\triangle\mathsf{sawTooth}}$ by restricting both the upper and lower bound by $\triangle A$. The formula for this weaker bound on the sensitivity is:

$$\underline{\triangle\mathsf{sawTooth}}(A, \mathsf{l}, \mathsf{d}, \mathsf{h}, \triangle A) \leq \begin{cases} \underline{\mathsf{sawTooth}}([A_l + \triangle A \ldots A_h - \triangle A], \mathsf{l}, \mathsf{d}, \mathsf{h}) - \underline{\mathsf{sawTooth}}(A, \mathsf{l}, \mathsf{d}, \mathsf{h}) & A_h - A_l > 2 \triangle A \\ \mathsf{sawTooth}(A, \mathsf{l}, \mathsf{d}, \mathsf{h}) - \underline{\mathsf{sawTooth}}(A, \mathsf{l}, \mathsf{d}, \mathsf{h}) & \mathsf{otherwise} \end{cases}$$

The method of restricting ranges can be applied to any function of a single attribute to find the maximum change for both the upper and lower bounds since it involves calculating the bounds using range of attribute values smaller than can be achieved by refining the action. It can also be extended to multi-attribute formulas, where each attribute range is restricted, both from above and from below by $\triangle A$. Of course, the resulting estimate can overstate the possible effect on $\overline{EU}$, but is very generally applicable.

$$\triangle \mathsf{DSA}(A_1, \triangle A_1, A_2, \triangle A_2, \ldots A_n, \triangle A_n)$$
$$\leq \begin{cases} \overline{\mathsf{DSA}}(A_1, A_2 \ldots A_n) - \min_{\forall A_i A'_i = [A_{il} + \triangle A_i \ldots A_{ih} - A_i]} \overline{\mathsf{DSA}}(A'_1, A'_2, \ldots A'_n) & \forall A_i A_{ih} - A_{il} > 2 * \triangle A_i \\ \overline{\mathsf{DSA}}(A_1, A_2 \ldots A_n) - \underline{\mathsf{DSA}}(A_1, A_2 \ldots A_n) & \text{otherwise} \end{cases}$$

7.5.2 Work Estimates for Deterministic Domains

To select which action to refine, we need not only an estimate of the effect of the refinement on $\overline{EU}$, but also an estimate of the expected work. For the Xavier and Pyrrhus planners, the work needed to perform any particular refinement was approximately constant. For the DRIPS planner, the work needed to refine and evaluate a plan depends on the length of the plan. The reason is that the planner must project the initial state of the world through each action in the plan in order to create the chronicle that is used to evaluate the plan. To estimate the amount of work needed to refine an action, we could determine the number and length of all the sub-plans that would have to be evaluated. The problem with this approach is that, with macro actions, the order in which you expand actions affects the number and length of the plans that need to be evaluated. In this section, we show that automatically expanding macro actions before evaluating a plan tends to reduce the amount of work required to expand an action and eliminates the problem with different action expansion sequences requiring different amounts of work.

Since the DRIPS planner evaluates a plan by projecting an initial state through a sequence of actions to create a chronicle, the plan evaluation cost for a deterministic plan is approximately linear in the length of the plan. The scatter plot in figure 7.8 shows evaluation time versus plan length for plans taken from the brew brewing domain. All empirical results for the DRIPS planner were run on a Sun Sparc 5 computer using Allegro Common Lisp 4.2 under SunOS 4.1. From the graph, we see that the time is approximately linear in the length of the plan, with an initial offset that accounts for the per plan overhead.

The work needed to refine an abstract action can be estimated from the number and length of the sub-plans that need to be evaluated, given the formula for estimating the work needed to evaluate a plan. The problem with this approach is that the number and length of the sub-plans depends on the order in which the sub-actions are refined. If the sub-action hierarchy for an abstract action contains only instance abstractions, then the number of actions in the plan remains constant and the order of expansion does not affect the number of nodes in the search tree or the cost of each evaluation. With macro actions, neither assertion holds. Obviously, macro-actions affect the length of the plan, but they also affect the number of plans that must be evaluated. Consider the action abstraction hierarchy shown on the left of figure 7.9 and the two possible search trees shown to its right. The search tree on the left corresponds to refining the instance action first and requires six plan

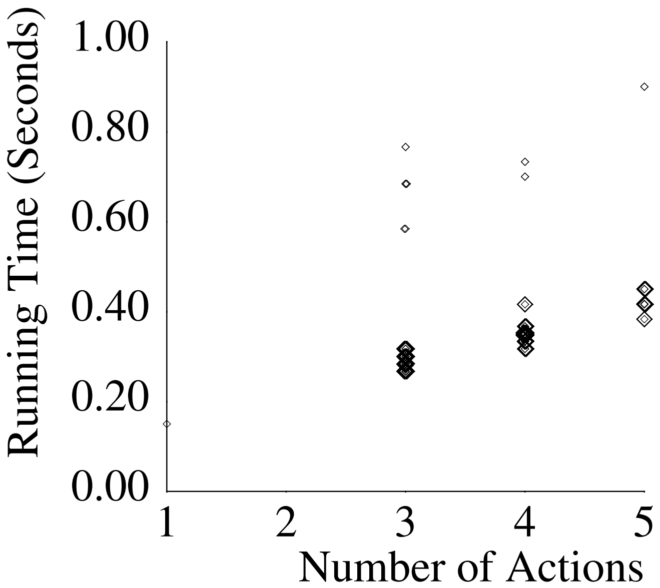

Figure 7.8: Plan evaluation time versus plan length for deterministic domain.

evaluations. Weighting each plan by its length gives 13 as the expected cost of evaluating the tree. The search tree on the right corresponds to refining the macro action first. It requires only five plan evaluations and has an expected evaluation cost of 12. Refining the macro action first, in this case, is better both in terms of the number of plans that must be evaluated and the expected cost of the evaluation.

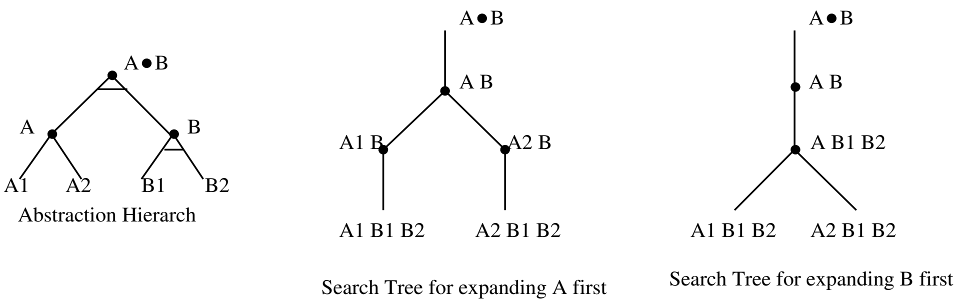

Figure 7.9: Order of refinement affects the size and shape of the search tree.

Generalizing the simple example in figure 7.14 to include instance and macro actions with two or more sub-actions, let I be the number of sub-actions for the instance action and let M be the number of sub-actions for the macro action. If the instance action is expanded first, then the work required is:

$$1 + 2 + 2I + I * (1 + M)$$

Similarly, if the macro action is expanded first, then the work is:

$$1 + 2 + (M + 1) + I * (1 + M)$$

	Number of Plans	Time (sec)
Without Expansion	348.4	3.27
With Expansion	325.2	2.71
Improvement	6.7%	17.1%

Table 7.2: The average time and number of plans evaluated with and without automatic macro-action expansion for 100 random brewing problems.

The only difference is in the third term that correspond to the evaluation costs for the nodes one level up from the leaves. This is to be expected since these are the only nodes that differ between the two trees. The expected work for both trees will be the same when: $2I = M + 1$ This simple analysis shows that the work required to refine the macro actions first grows proportional to the number of actions in the macro-actions, M, whereas the work needed to refine the instance action first grows proportional to twice the number of instance sub-actions, I. Since macro actions tend to be short sequences of two or three actions, selecting macro actions for expansion first will tend to improve performance, even if there are also relatively few sub-actions for every instance action.

In addition, it is not necessary to evaluate all the nodes in the search tree. Suppose we automatically expand each macro action when it is encountered and before the plan with the macro action is evaluated. By doing this, we eliminate some plan evaluations and variation in the topology of the search tree. The cost is that some opportunities for earlier pruning are lost. If the probability of pruning is small, then the benefits outweigh the cost. Let the probability of pruning, after refining the instance action and before expanding the macro action is p, then the expected work for expanding the tree is[2] $1 + 2 + 2I + (1 - p)I * (1 + M)$. If the macro action is always expanded before evaluation, the expected work is $1 + 2 + I * (1 + M)$. In cases where $p \leq \frac{2}{(1+M)}$, then automatically expanding the macro action is better.

In practice, automatically expanding the macro action leads to substantial speed improvements. Typically, the probability of pruning after evaluating a plan with a macro action and before further refinement is relatively low. This is especially true for difficult planning problems where the probability of pruning any plan is low. When the probability of pruning is zero, then the expected savings for automatically expanding macro actions is $\frac{2*I}{2+I*(3+M)}$, using our simple work model. Work savings are most significant for small values of M, corresponding to short macro actions. In practice, most macro actions tend to have two or three sub-actions, and so are relatively short. In our simple example with $M = I = 2$ and $p = 0$, the savings from automatically expanding macro actions is 1/3.

To allow us to demonstrate the effect of automatic expansion of macro-actions empirically, we modified the DRIPS planner so that we could turn automatic macro-action expansion on or off. We then ran the modified planner on a set of 100 random brewing problem, with and without automatic expansion. Table 7.2 summarizes the results, which show a 17% average improvement in run time for using automatic expansion. This is in

[2]We are assuming independence and making use of the fact that the mean of a binomial distribution is n*p when n is the number of trials and p is the probability of success.

Method	Number of Plans	Time (sec)	Time/Plan (msec)
Random	592.41	4.60	7.7
Default (First action)	325.20	2.37	7.3
Sensitivity Analysis	177.01	1.64	9.3
Sensitivity Analysis (no work)	185.19	1.73	9.3

Table 7.3: The average number of plans evaluated and the average running time for each of four action selection methods used on 100 random brewing problems.

line with the predictions from our model, since this domain tends to have more instance actions than macro-actions and the average length of the macro actions is closer to three than two. The probability of pruning is also greater than zero.

We now return to the problem of estimating the work required to expand an abstract action. We base our estimate on the number and length of the sub-plans that have to be evaluated, taking into account the automatic expansion of macro actions. Since macro-actions are automatically expanded, we will need work estimates only for expanding instance actions. The work estimated for each action needs to be only relative to each other since scaling all the estimates by a constant would not affect the choice of which action to refine. The number an length of the sub-plans that result from expanding an abstract action can be determined from the sub-actions. For sub-actions that are instance actions or primitive actions, the cost of evaluating the sub-plan will be the same as the cost of evaluating the original plan since it will be the same length. The relative cost is one. For sub-actions that are macro actions, automatic macro-action expansion will result in a longer plan to be evaluated. Since the cost of evaluation is proportional to length, the relative cost of evaluating the resulting plan is proportional to the relative increase in length. The following formula summarizes the method for estimating work:

$$\begin{aligned}
\mathsf{work(expand(A{:}Action, P{:}Plan))} &= \sum_{\mathsf{a{:}subaction(A)}} \mathsf{work(evaluate(a, P))} \\
\mathsf{work(evaluate(a{:}action, P{:}Plan))} &= \begin{cases} 1 & \mathsf{instanceAction(a)} \\ \frac{\mathsf{(length(a)+length(P)-1)}}{\mathsf{length(P)}} & \mathsf{macroAction(a)} \end{cases}
\end{aligned}$$

7.5.3 Performance Results for Deterministic Domains

In order to evaluate the performance of the sensitivity analysis based refinement selection method, we compared it to the simple heuristic that selects the first abstract action in a plan and random action selection. Table 7.3 shows the average performance of each method on 100 randomly generated beer brewing problems. The problems were created by randomly selecting beer related attributes to include in the utility function and then randomly selecting low, desired and high values used in the saw-tooth functions for each attribute.

As can be seen from the table, the sensitivity analysis method significantly outperforms the default method, that in turn significantly outperforms random selection. The default heuristic simply selects the first abstract action in the plan for refinement. This is particularly effective in the beer brewing domain since the first action is to select which malt to use and the malt has a significant affect on most of the characteristics of the resulting beer. The sensitivity analysis method evaluates 45.6% fewer plans than the default heuristics and takes 30.1% less time. The difference in the time per plan between the default method and the sensitivity analysis is due to the overhead of performing the sensitivity analysis. In this example, the sensitivity analysis takes about 22% of the processing time. Differences in the lengths of the plans evaluated by each method also contribute to the difference. However, in the beer brewing domain, plan length does not vary much, so the difference is almost entirely due to the overhead of sensitivity analysis. It is also interesting to note that when the sensitivity analysis is used without estimating the amount of work needed, the average number of plans and average time increase by 4.6% and 5.5% respectively. This shows the importance of considering the work needed to perform a refinement in addition to how much the refinement can effect $\overline{EU}$.

7.5.4 Sensitivity Analysis for Probabilistic Domains

The sensitivity analysis methods described in section 7.5.1 and the method for estimating work described in section 7.5.2 must be extended in order to deal with probabilistic domains. In this section, we describe a method for calculating sensitivity in probabilistic domains. In the next section, we develop a method for estimating the work needed to expand an abstract action in a probabilistic domain.

Probabilistic domains produce multiple chronicles, one for each possible outcome of a plan. Each branch in a chronicle corresponds to a particular conditional effect in the corresponding action in the plan. Refining an action can affect the utility, both DSA and UR, as well as further constrain the probabilities of each chronicle. It is relatively straightforward to extend the calculations of $\triangle DSA$ and $\triangle CT$. The same calculations are done, only this time using the conditional effects in each action that apply to the chronicle being evaluated. Extending the sensitivity analysis to account for the effects on the probabilities associated with each chronicle is more involved. In this section, we describe the extensions needed to do sensitivity analysis in probabilistic domains. We begin by describing the method DRIPS uses to evaluate plans in probabilistic domains. We then show how the chronicle structure, produced in evaluating the plan, can be used to calculate the sensitivity of $\overline{EU}$ to refinement of a particular action.

Figure 7.10 shows a simple medical domain where a treatment policy consists of an abstract test followed by a treatment if the test is positive. The plan is evaluated by projecting the initial state through the two actions to create the chronicle tree show in figure 7.11. The initial state is first split in two states, one with and one without the disease. This split corresponds to the first branch in the Test1or2 action. Each of the resulting states is again split into two depending on whether the test result is positive or negative. This split corresponds to the second branch in the Test1or2 action. Finally, the treat_if_positive action is applied. Although this action branches on the disease state, there is no split because it is already split on these attributes and none of the intervening actions has changed either

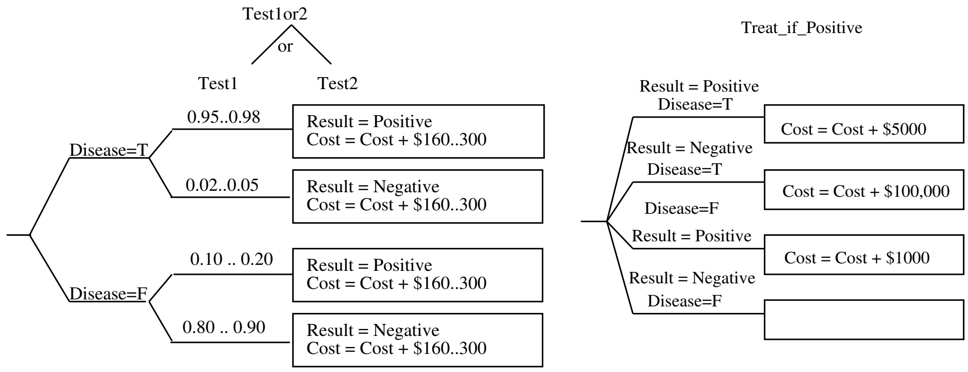

Figure 7.10: Abstract test action representing tests with different costs and different false negative and false positive probabilities and an action that treats if the test result is positive.

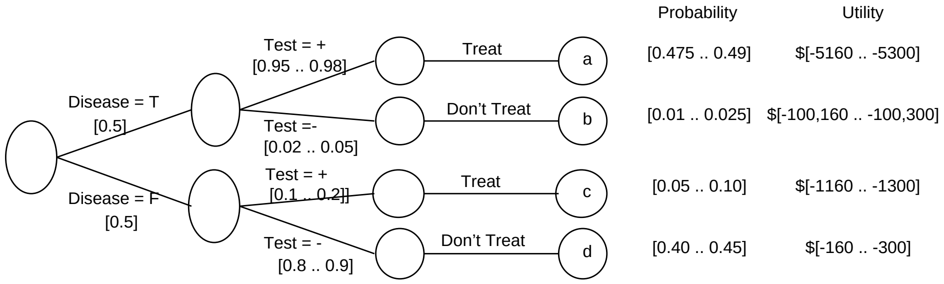

Figure 7.11: Tree showing the chronicles used to evaluate the test and treat plan.

one. Each path from the root of the chronicle tree to a leaf node is a chronicle that gives one possible sequence of states. The number of chronicles is equal to the number of leaf nodes, and we use the labels on the leaf nodes to refer to the corresponding chronicles.

The chronicle tree is labeled with the probabilities for each branch and the final states are labeled with their ranges of probability and utility. The expected utility, $EU(p) = \sum_{c \in chronicles(p)} U(c) \cdot P(c)$, is the sum of the utilities of each chronicle weighted by its probability. When the probabilities of the final states are point-valued, the formula can be applied directly. When the probabilities are ranges, using the formula does not give tight bounds on the expected utility. The reason is that using the probability ranges alone allows the probability mass, represented by the ranges, to shift around in unconstrained ways, possibly violating some of the constraints represented in the chronicle tree. To see how this happens, consider the evaluation of the chronicle tree show in figure 7.11. The probabilities of the chronicles sum to $[0.935 \ldots 1.065]$. This means that 93.5% of the probability mass is allocated to specific chronicles and that 6.5% is partially unconstrained. To calculate $\overline{EU}$, this probability mass is allocated to the high utility (low cost) chronicles. $\underline{EU}$ is calculated by allocating the probability mass to low utility chronicles.

A straightforward application of the expected utility formula to calculating an upper bound on expected utility would allocate excess probability to the chronicles with the highest upper bound on utility. In our example, 5% of unallocated probability mass can be allocated to the chronicle with highest upper bound on utility (chronicle d in figure 7.11) and the remaining 1.5% can be allocated to the chronicle with the second highest bound on utility (chronicle c) without violating the upper bound on probability for each chronicle. The resulting upper bound on expected utility (lower bound on expected cost) is $0.475*(-5160)+0.01*(-100160)+0.065*(-1160)+0.45*(-160) = -3600$. The similarly calculated lower bound on expected utility is $0.49*(-5300)+0.025*(-100300)+0.085*(-1300)+0.40*(-300) = -5335$. The problem with the calculation of the upper bound on expected utility is that the total probability mass allocated to the chronicles where patients do not have the disease, chronicles c and d, is $0.065 + 0.45 = 51.5\%$, which is greater than the fraction of patients who are disease free, 50%. This problem arises because some of the probability mass from chronicles a and b has been shifted to chronicles c and d. A similar problem arises when calculating the lower bound on expected utility, where $0.49 + 0.025 = 51.5\%$ of the probability mass is allocated to chronicles where the patients have the disease.

The solution is to use the chronicle tree and allocate probability mass at each branch point. The utility values at the leaves are propagated up towards the root of the tree. At each branch point, each utility is weighted by the probability of the branch and unconstrained probability mass is allocated to give either the minimum or maximum value of EU. This is done by sorting the branches by increasing or decreasing utility and allocating excess utility in the order of the list. In the example, when calculating the upper bound on EU, the chronicles after each branch would be sorted in decreasing order of utility, which is increasing order of cost. In the branch on test result for people with the disease, the maximum of 98% of the probability mass would be allocated to the branch with the positive result. Similarly, for people without the disease, 90% of the probability mass would be allocated to the negative result. The branch at the root of the tree assigns exactly 50% of the probability to each branch, so there is no excess probability to allocate. The resulting upper bound on expected utility (lower bound on expected cost) is $0.50*0.98*(-5160)+0.50*0.02*(-100160)+0.50*0.10*(-1160)+0.50*0.90*(-160) = -3660$. The corresponding lower bound on expected utility is $0.50*0.95*(-5300)+0.50*0.05*(-100300)+0.50*0.20*(-1300)+0.50*0.80*(-300) = -5275$. The resulting bounds, $[-5275 \ldots -3660]$ are tighter than those calculated without taking the probability constraints into account, $[-5335 \ldots -3600]$. The advantage of tighter bounds is that they increase the probability that sub-optimal plans can be pruned with less work.

The structure of the chronicle tree is also used when calculating the sensitivity of $\overline{EU}$ to refinement of a particular action. The methods outlined for deterministic domains can be used to determine how much $\overline{EU}$ could be lowered for a particular chronicle. The resulting values for $\overline{EU}$ are then propagated up the tree towards the root using the method for calculating the upper bound on EU, except at branches corresponding to the action being refined. At those branches, instead of allocating unconstrained probability mass to maximize EU, it is allocated to minimize EU, since refining the action has the potential to assign this probability mass in the most unfavourable way. The result of this calculate is the minimum $\overline{EU}$ that could result from refining the action. Subtracting this value from $\overline{EU}$ gives the sensitivity of $\overline{EU}$ to refining the action.

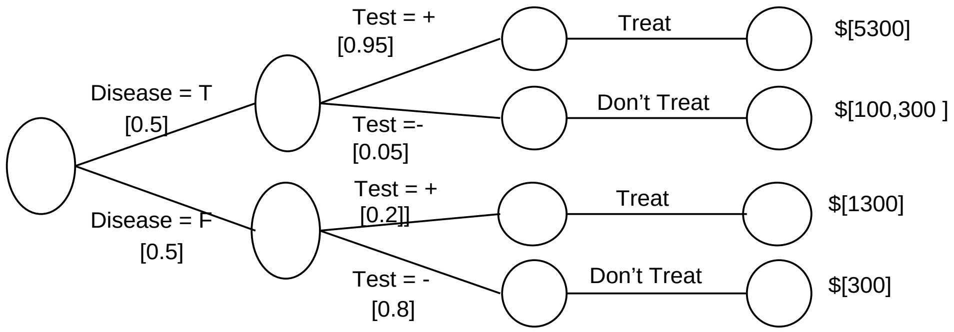

Figure 7.12: Tree showing the values used for the sensitivity analysis.

Returning to our example, figure 7.12 show the values that would be assigned when calculating the sensitivity of the Test1or2 action. Evaluating the upper bound that could result from refining the Test1or2 action gives: $(0.50 * 0.95 * (-5300) + 0.50 * 0.05 * (-100300) + 0.50 * 0.20 * (-1300) + 0.50 * 0.80 * (-300) = -5275$. The sensitivity is the original upper bound on expected utility minus the new bound, $(-3660) - (-5275) = 1615$ This is the same as the range in EU for the plan, since the Test1or2 action is the only abstract action in the plan that could be refined.

To summarize, in this section we have shown how a chronicle tree can be used to calculate bounds on the expected utility for plans in probabilistic domains. Using the chronicle tree to allocate probability mass results in tighter bounds on the expected utility, which promotes pruning of plans with low expected utility from the search space. We have also shown how a chronicle tree can be used in probabilistic domains to evaluate the sensitivity of $\overline{EU}$ to refinement of an action in a plan. In the next section, we develop a method for estimating the work needed to expand an abstract action in a probabilistic domain.

7.5.5 Work Estimates for Probabilistic Domains

In probabilistic domains, the cost of evaluating a plan is generally not linear in the length of the plan, but is approximately exponential in the length. The is because each action in the plan can split the chronicles and the number of chronicles is exponential in the number of splits. The number of chronicles to be evaluated will be approximately $B^{length(plan)}$, where B is the average branching factor per action. As shown in figure 7.13, the time needed to evaluate a plan is approximately linear in the number of chronicles. When doing action refinement, expanding an instance action produces sub-plans with the same length. The work needed is approximately n times the work needed to evaluate the original plan, where n is the number of sub-actions that the instance action expands into. For a macro action, the expected work will be B^{n-1} times the work to evaluated the original plan since the length of the plan is increased by $(n - 1)$, where n is the number of sub-actions in the macro action.

As noted before, macro actions also affect the topology of the search tree. Consider again the action abstraction hierarchy in figure 7.14, and the two search trees that correspond

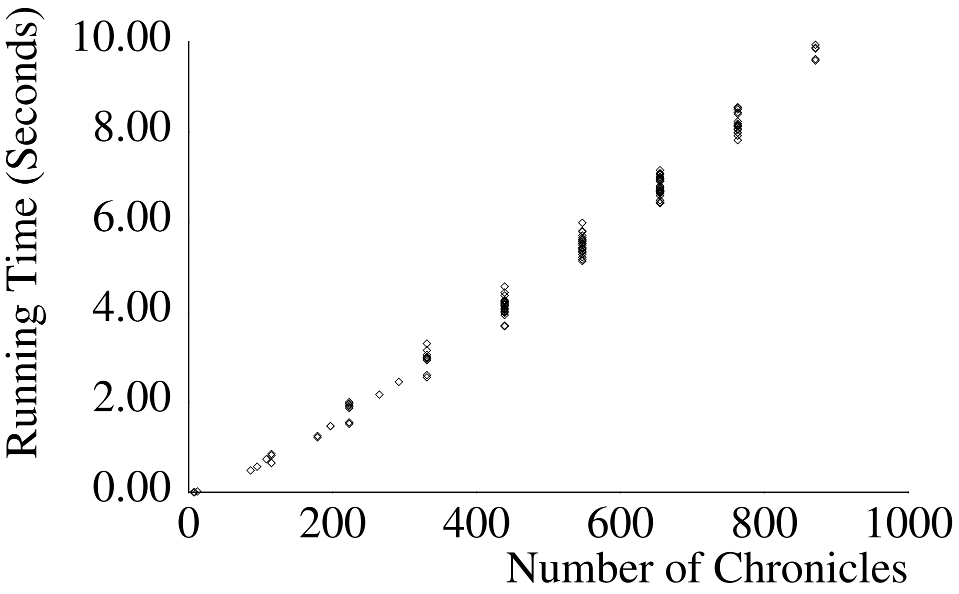

Figure 7.13: Plan evaluation time versus the number of chronicles for a probabilistic domain.

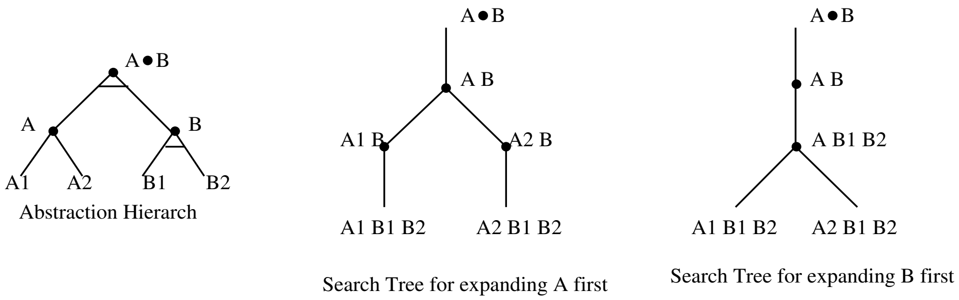

Figure 7.14: Order of refinement affects the size and shape of the search tree.

to different action refinement orderings. In the tree on the left, action A is expanded first and there are six plan evaluations. The expected work is $B^1 + 3B^2 + 2B^3$. The other search tree corresponds to refining action B first. It has 5 nodes and the work needed is $B^1 + B^2 + 3B^3$. The difference is $2B^2 - B^3 = B^2(2 - B)$, equals zero only if B is exactly 2. For a higher B, it is better to refine the instance action first, even though the planner ends up evaluating more plans. For values of B less than 2, expanding the macro action first produces the best result.

Let us generalize the simple example in figure 7.14 to include instance and macro actions with two or more sub-actions. Let I be the number of sub-actions for the instance action and let M be the number of sub-actions for the macro action. If the instance action is expanded first, then the work required is:

$$B + B^2 + IB^2 + IB^{1+M}$$

Similarly, if the macro action is expanded first, then the work is:

$$B + B^2 + B^{M+1} + IB^{1+M}$$

The only difference is in the third term that correspond to the evaluation costs for the nodes one level up from the leaves. This is to be expected since these are the only nodes that differ between the two trees. The expected work for both trees will be the same when:

$$IB^2 = B^{M+1} \Rightarrow I = B^{M-1} \Rightarrow B = I^{1/(M-1)}$$

For $B > I^{1/(M-1)}$, expanding the instance action first is better. This formula has the expected property that increasing I increase the cutoff linearly whereas increasing M decreases it exponentially. Note that this analysis depends on the branching factor, B, being greater than 1. If the branching factor were 1, then there would only be a single chronicle and we would have the deterministic case where the work is equal when $2I = M + 1$

Repeating our analysis of automatic expansion of macro-actions, this time for probabilistic domains, again shows that automatic expansion can save work. Let the probability of pruning, after refining the instance action and before expanding the macro action is p, then the expected work for expanding the tree[3] is $B + B^2 + IB^2 + (1-p)IB^{1+M}$ If the macro action is always expanded before evaluation, the expected work is $B + B^2 + IB^{1+M}$. In cases where $p \leq B^{(1-M)}$, then automatically expanding the macro action is better. For the case where $B = M = 2$, the requirement is only that $p \leq 1/2$, which is generally the case.

In practice, automatically expanding macro actions leads to significant speed improvements. Typically, the probability of pruning after evaluating a plan with a macro action and before further refinement is relatively low. This is especially true for difficult planning problems where the probability of pruning any plan is low. When the probability of pruning is zero, then the expected savings for automatically expanding macro actions is $\frac{IB^2}{B+B^2+IB^2+IB^{M+1}} = 0.27$ when $I = B = M = 2$. Work savings are most significant for small values of M, corresponding to short macro actions. In practice, most macro actions in the DRIPS domains we have available to us tend to have two or three sub-actions, and so are relatively short. Also, the branching factor for many macro actions with tight bounds tends to be closer to B^M rather than B, and the corresponding savings are closer to $\frac{IB^{M+1}}{B+B^2+IB^{(M+1)}+IB^{(M+1)}} = 0.4$ when $I = B = M = 2$. This is because to get tight bounds, the macro action must be as expressive as the sequence of actions it represents. If it is less expressive, then the branching factor is lower, but the bounds on the plan are less tight and the probability of pruning is lower.

Table 7.4 gives the number of plan evaluations and the run time for planning with and without automatic macro action expansion in the DVT domain using the default heuristic for selecting which action to refine. As suggested by the chart, automatic macro action expansion improves performance in this domain by approximately 20%. The actual reduction is not equal to the 27% given in the sample analysis because the mix of instance and macro

[3] We are assuming independence and making use of the fact that the mean of a binomial distribution is n*p where n is the number of trials and p is the probability of success.

Cost of Fatality	With Macro Expansion		Without Macro Expansion		Improvement
	Plans	Time	Plans	Time	
50000	4	1.567	7	1.683	7.4%
100000	16	24.517	20	28.817	17.5%
150000	22	43.633	28	56.583	29.7%
200000	32	93.883	40	116.133	23.7%
300000	64	292.600	80	352.966	20.6%
500000	140	983.833	176	1211.634	23.2%

Table 7.4: Performance with and without automatic expansion of macro-actions

actions is not the same and the number of sub-actions for an instance abstraction tends to be higher than 2.

Automatic macro action expansion was selected by Doan and Haddawy as the default used by the DRIPS planner. The domains supplied with the planner do not specify effects for macro actions. Macro actions are used only for sequencing and never for plan evaluation in the released version of the planner. Our analysis and empirical results suggest that, in general, it was a good choice.

Returning to the problem of estimating work of refining an abstract action in a probabilistic domain, we again look the amount of work that must be done to evaluate the resulting sub-plans. We sum the estimates of the relative amount of work needed to evaluate each resulting sub-plan, this time taking into account the fact that the cost of evaluating a plan is exponential in the length rather than linear, since the number of chronicles that must be evaluated is approximately exponential in the number of actions in the plan. To estimate the branching factor, we use the number of chronicles in the original plan and take the nth root, where n is the number of actions in the original plan. Raising the branching factor to the number of actions in the sub-plan gives an estimate of the number of chronicles for the sub-plan. For instance actions, the number of chronicles and the length of the plan will remain constant. The relative work needed is one. For macro-actions, the length of the plan and the estimated number of chronicles will increase. The expected work is given by:

$$\mathsf{work(expand(A{:}Action, P{:}Plan))} = \sum_{\mathsf{a{:}subaction(A)}} \mathsf{work(evaluate(a, P))}$$

$$\mathsf{work(evaluate(a{:}action, P{:}Plan))}$$
$$= \begin{cases} 1 & \mathsf{instanceAction(a)} \\ \frac{\mathsf{(length(a)+length(P)-1)}}{\mathsf{length(P)}} * |\mathsf{chronicles(P)}|^{\left(\frac{\mathsf{(length(a)+length(P)-1)}}{\mathsf{length(P)}}\right)} & \mathsf{macroAction(a)} \end{cases}$$

7.5.6 Performance Results for Probabilistic Domains

To evaluate the effectiveness of sensitivity analysis based search control in probabilistic domains, we applied our methods to three variations of the deep venous thrombosis (DVT)

Cost of Fatality	S A		Priority		Default	
	Plans	time	Plans	time	Plans	time
50,000	739	1499	852	1014	8413	8813
100,000	723	1465	838	1004	8411	8749
150,000	707	1553	818	952	8415	8890
200,000	713	1553	818	1060	8411	9554
300,000	669	1481	762	972	8409	9541
500,000	669	1498	740	972	8405	9381
650,000	711	1495	1079	1222	10775	13556
850,000	826	1743	1467	2123	19841	22825
1,000,000	860	1800	1447	2157	22629	26046
1,100,000	846	1758	1445	2126	23141	26746
1,300,000	811	1663	1437	2104	23397	27476
1,500,000	686	1367	1672	2537	24339	29533

Table 7.5: Results from the original DVT domain comparing the default heuristic to a hand tuned priority scheme and the sensitivity analysis method. Times are in seconds.

domain and compared its performance to the default heuristic used in DRIPS and a hand tuned priority scheme. All of the tests were performed using the optimal optimistic method for selecting which plan to refine. The first test was performed using the original DVT domain description supplied with the DRIPS planner. The hand tuned priority scheme for this domain was implemented by the same person who implemented the first version of the DRIPS planner and coded the DVT domain description. The second version of the domain introduces recursion, or loops in the action hierarchy. It allows the planner to decide how many tests to perform before giving a treatment rather than having the domain designer specify a preset limit. Finally, we introduce a variation on the domain description that allows waiting periods between tests, to show how certain types of actions can negatively impact the sensitivity analysis action selection method.

The DVT domain is the largest domain that is supplied with the DRIPS planner. The version used for the test results shown in table 7.5 allow up to four tests before a decision is made about whether to treat a patient. From the results illustrated in figures 7.15 and 7.16, it is obvious that the planner using the default heuristic takes an excessive amount of time, on the order of eight CPU hours, to solve some of the harder problems. To improve performance, the creator of the planner implemented a priority based scheme where he could assign priorities to each action. Actions with higher priorities are selected for refinement first. Using this scheme, the priorities were hand tuned to improve performance on this particular problem. The tuning was done iteratively where one set of priorities were selected and tested and then modified and tested again. The result was more than an order of magnitude improvement in the performance of the planner, both in terms of the number of plans evaluated and the total run time.

The sensitivity analysis method, when applied to the same problem, produces comparable results to the hand tuned priority method. In terms of the number of plans evaluated,

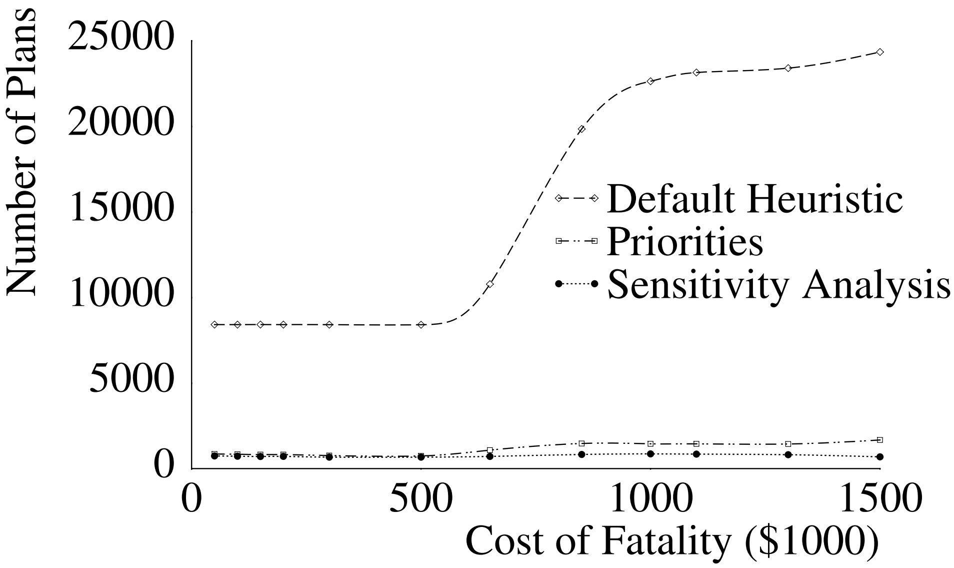

Figure 7.15: Plot of the number of plans evaluated versus the cost of fatality in the original version of the DVT domain.

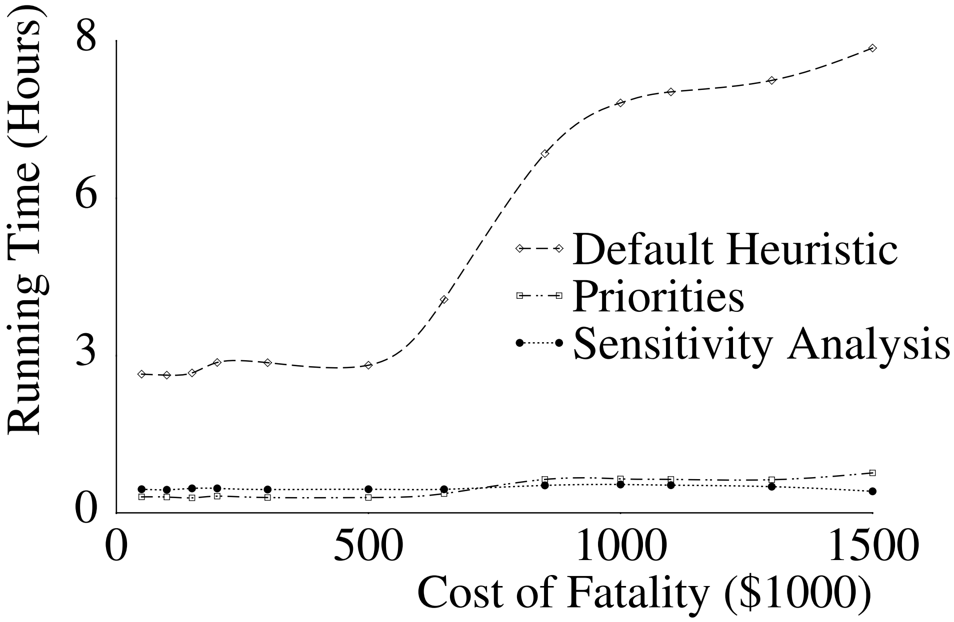

Figure 7.16: Plot of the number of total CPU time versus the cost of fatality in the original version of the DVT domain.

the sensitivity analysis method slightly outperforms the priority scheme at low costs of fatality, but significantly outperforms the priority scheme at higher costs of fatality. Because of the overhead of doing the sensitivity analysis, the priority scheme takes less total computation, even when the sensitivity analysis method evaluates slightly fewer plans. For probabilistic domains, the overhead for sensitivity analysis is approximately 17%. The

Cost of Fatality	S A		Priority		Default	
	Plans	time	Plans	time	Plans	time
5,000	4	2.58	4	1.90	4	1.95
10,000	4	2.33	4	1.88	4	1.90
50,000	4	2.32	4	1.88	4	1.90
100,000	22	49.6	22	33.1	16	28.4
150,000	28	82.6	28	72.1	22	52.1
200,000	32	118.1	40	108.1	32	107.8
300,000	52	259.8	68	236.1	64	332.1
500,000	74	566.3	118	576.1	140	1113.0
650,000	86	813.3	172	1023.8	264	2696.3
850,000	128	1679.2	320	2407.8	868	13022.9
1,000,000	202	3413.6	584	5337.4	1678	29803.9

Table 7.6: Results from the DVT domain with recursive tests comparing the default heuristic to a hand tuned priority scheme and the sensitivity analysis method. Times are in seconds.

sensitivity analysis overhead scales with plan evaluation, which is the dominant planning operation in terms of CPU time. As a result, the sensitivity analysis overhead remains close to a constant 17% of the total processing time. At the highest costs of fatality, where the sensitivity analysis method is evaluating only 40% as many plans as the priority scheme, it is also out-performing the priority scheme by almost two to one in total computation time.

The reason that the priority scheme does relatively worse at higher costs of fatality might be due to the fact that it was tuned using $500,000 as the cost of fatality[4]. The priority scheme was tuned to reduce the number of plans evaluated, rather than the total time, and evaluates its fewest number of plans at a $500,000 cost of fatality. This is not, however, the point at which the priority scheme does best. At $150,000, the priority scheme evaluates more plans, but takes slightly less time. The reason is that some plans are more expensive to evaluate then others. Even at the point where the priority scheme evaluates its fewest plans, the sensitivity analysis method evaluates 10% fewer plans, although it takes almost 50% longer to solve the problem. But, because the sensitivity analysis method can adapt by selecting different actions for refinement depending on the cost of fatality, it is able to solve the planning problem with a relatively constant number of plan evaluations over the full range of problems. Since the priorities are hard coded, the priority based method cannot adapt, and performance degrades as the problem moves away from the one used to tune the priorities.

In addition to adaptability, the sensitivity analysis method has the advantage that it does not require a tuning phase. It produces performance comparable to a hand tuned system, without the effort required on the part of the domain designer to do the hand tuning. Hand tuning requires solving the problem repeatedly to compare priority schemes.

Our second comparison makes use of a DVT domain with recursive test actions [Goodwin, 1996]. One problem with the original version of the DRIPS planner was that it did not

[4]Personal communications with AnHai Doan.

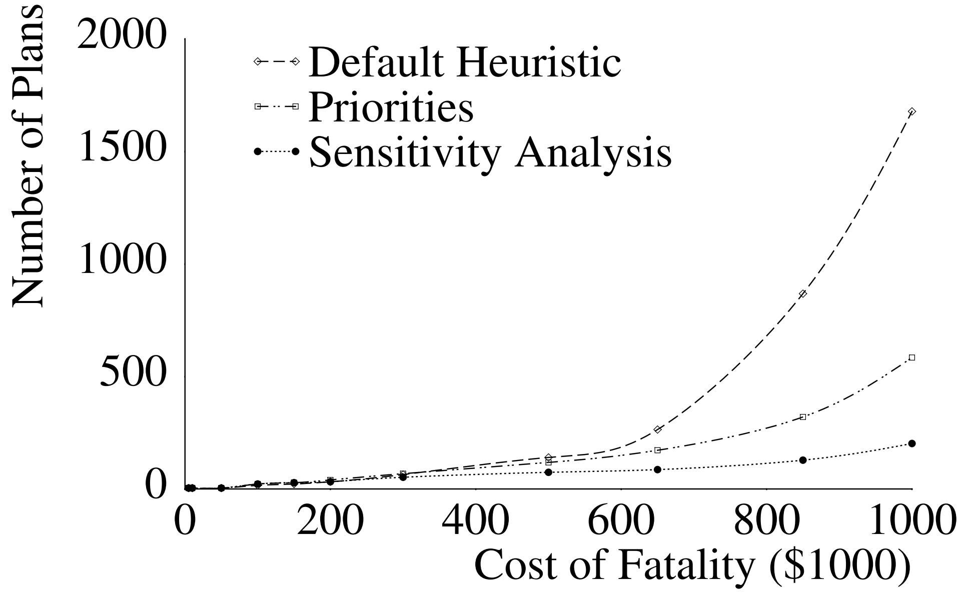

Figure 7.17: Plot of the number of plans evaluated versus the cost of fatality for the DVT domain with recursive tests.

allow actions to be repeated an arbitrary number of times. In the original version of the DVT domain, for instance, the designer had to hand code policies with zero, one, two, three and four tests. This problem was rectified when we extended the planner to allow recursion in the action abstraction hierarchy as described in section 5.5. In the DVT domain, this allows the planner to decide how many tests to include in a treatment policy.

With the addition of recursive test actions, the priority scheme used for the original domain was no longer applicable. We attempted to assign new priorities only to the new actions, but this resulted in poor performance. To improve performance, we re-tuned the priorities for good performance at high costs of fatality. Table 7.6 and figures 7.17 and 7.18 show the comparison between the default heuristic, the revised priority scheme and the sensitivity analysis method for the version of the DVT domain with recursive test actions.

In tuning the priority scheme for high costs of fatality, we created a preference for not unrolling the loop of tests. Instead, the scheme attempts to resolve which tests and treatments to use in parts of the plan not involving recursive test actions. This is a particularly effective control strategy at high costs of fatality where three or more tests are required for the optimal solution. Since every unrolling of the loop increase the length of the plan linearly, but increases the evaluation cost exponentially, unrolling loops in already long plans leads to very expensive evaluations. The default heuristic, on the other hand, generally prefers to unroll the loop because the test action is the first action in the plan. This strategy is effective at low costs of fatality as evident from the performance of the default method relative to the priority scheme at low costs of fatality. At high costs of fatality, it fails to resolve the treatment options early enough to avoid unrolling the loop unnecessarily and as a result, pays a heavy performance penalty.

At very low costs of fatality, the best plan is to simply not test or teat anyone. This

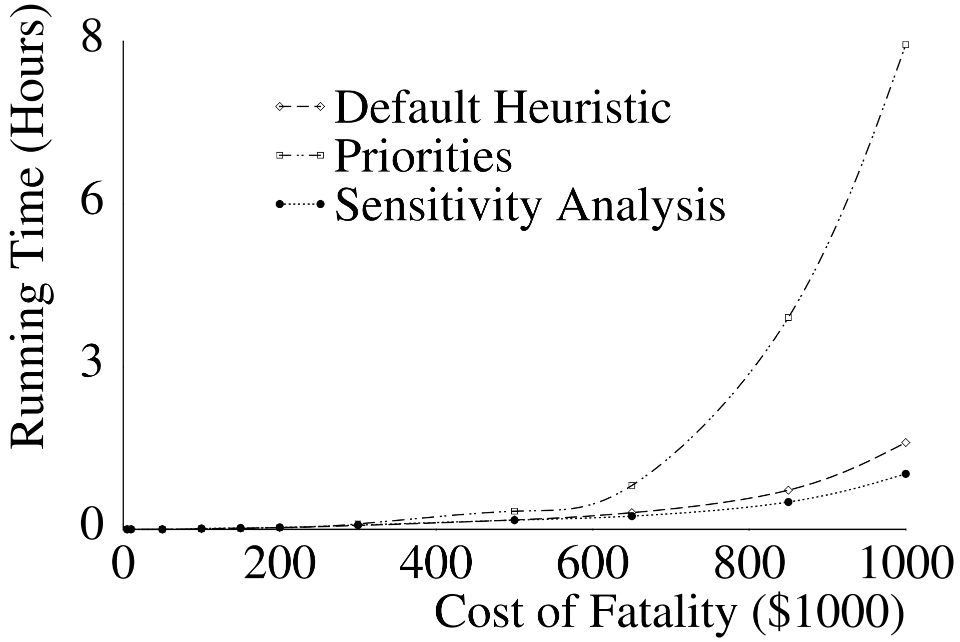

Figure 7.18: Plot of the CPU time versus the cost of fatality for the DVT domain with recursive tests.

result can be determined by evaluating the first node in the search tree and its three children. There are no search control decisions to be made and all methods evaluate the same number of plans. As the cost of fatality increases, the number of choice points increases as does the value of making better decisions. Over the range of problems, the sensitivity analysis method never evaluates more plans than the hand tuned priority scheme. Above a cost of fatality of $300,000, it evaluates significantly fewer plans and takes less total time than either of the other two methods.

When comparing the performance of the sensitivity analysis method and the priority scheme, one might question whether the priority scheme had been sufficiently optimized. Would further tuning significantly improve its performance? If the best priority scheme was found, would it not necessarily be better than any other method? The answer is that the best priority scheme is not necessarily the optimal strategy. The reason is that the priority scheme assigns a fixed ranking to each action and cannot vary its preferences based on the position or relative order of each action. For example, the best decision may be to always refine the second of two particular abstract tests, test_a and test_b in a plan. The priority method can express only a preference for test_a or test_b but not for the second one of the two that appears in the plan. The language used to express preferences could be extended to allow more general rules for encoding preferences. Given a more expressive language, the problem would then be how to generate a good set of control rules. Finding an optimal set of control rules would, in general, be even more difficult than solving the original planning problem. Even if an optimal set of rules could be found, determining which one was applicable at each decision point, the match problem, may add more overhead than the planning time it saves. However, an optimal set of decision point choices for a particular

Cost of Fatality	S A		Priority		Default	
	Plans	time	Plans	time	Plans	time
5,000	5	2.7	5	2.0	5	2.0
10,000	5	2.4	5	2.0	5	2.0
50,000	9	6.8	13	11.2	9	6.4
100,000	51	133.2	37	60.9	33	70.3
150,000	63	202.9	61	131.6	45	104.4
200,000	77	305.2	81	196.1	67	200.0
300,000	121	664.3	167	588.3	157	769.0
500,000	211	2046.7	319	1520.3	441	3537.8
650,000	325	4415.9	507	2877.2	933	9803.3
850,000	615	11888.3	1057	7784.6	3105	46265.7
1,000,000	1309	34164.7	2231	20106.3		
1,100,000	957	23511.4	1699	14130.5		
1,300,000	1167	31950.9				

Table 7.7: Results from the DVT domain with recursive tests and maybe_wait actions comparing the default heuristic to a hand tuned priority scheme and the sensitivity analysis method. Times are in seconds.

problem would be very useful in providing a lower bound against which the other methods could be compared.

As we stated earlier, the advantage of the sensitivity analysis method is not only its run time performance, but the fact that it does not require any hand tuning. In moving to the DVT domain with recursive actions, the sensitivity analysis method did not require any additional work on the part of the domain designer. Since the utility function remained the same, the $\triangle DSA$ and the $\triangle UR$ remained the same. The domain designer can simply change the action descriptions to include the recursive tests and remove the actions that specified specific numbers of tests. Other changes, such as adding another test or treatment could also be done without requiring any changes to the functions needed to calculate sensitivity.

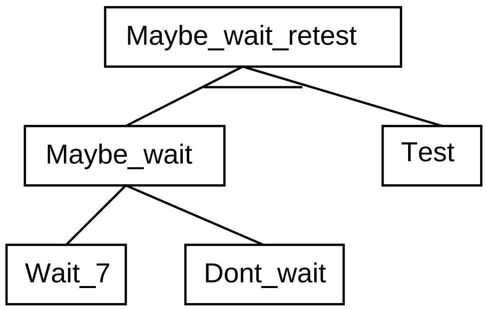

Figure 7.19: Adding an option to wait 7 days between tests.

The final comparison between the action selection methods involves an addition to the

DVT domain with recursive tests. To the set of tests and treatments, we add an option to wait seven days after getting a negative test result before repeating the test. The delay between tests had been proposed by physicians as a method for better detecting the disease. The idea is that the delay does not hurt healthy patients. For patients with the disease, it allows the disease to progress to a stage where it is more easily detectable, but still treatable. However, it turns out that the delay is never worth while. The likelihood that patients with the disease will suffer a complication in those seven days outweighs the benefit of increasing the likelihood of detecting the disease so that it can be treated. We include this example because it shows some of the potential problems with using sensitivity analysis for refinement guiding.

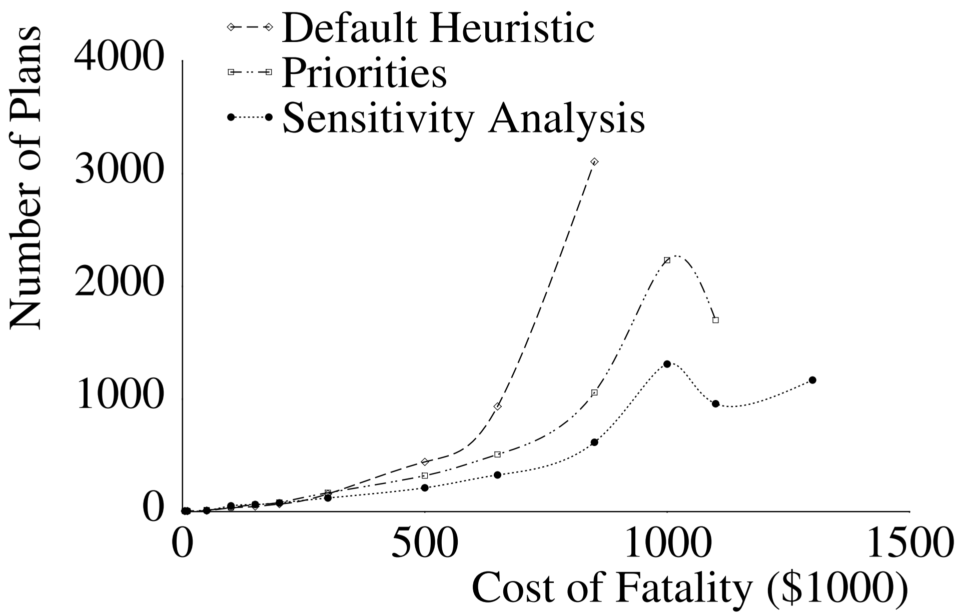

Figure 7.20: Plot of the number of plans evaluated versus the cost of fatality for the DVT domain with recursive tests and maybe_wait actions.

The "maybe_wait" action is added to the abstraction hierarchy as shown in figure 7.19. The "maybe_wait" action can be refined to the null, "dont_wait" action or the "wait_7" action. It is important to note the effect that refining the "maybe_wait" action will on $\overline{EU}$ of the resulting plans. The sub-plan with the "wait_7" action will have its $\overline{EU}$ lowered significantly because delaying testing and possible treatment for seven days increases the likelihood that a person with the disease will die (figure 7.22). The sub-plan with the null action will have an $\overline{EU}$ that is largely the same as the original plan. It will be slightly lower because the upper bound on expected utility for the original plan allowed for an increase in the probability of detecting the disease that a seven day wait would give, without the increased probability of a fatality.

Actions, like the maybe_wait action represent a pathological case for sensitivity analysis based meta-level control as compared to priority based schemes. The reason is that the upper bound on the expected utility of one of the sub-plans does not change even though

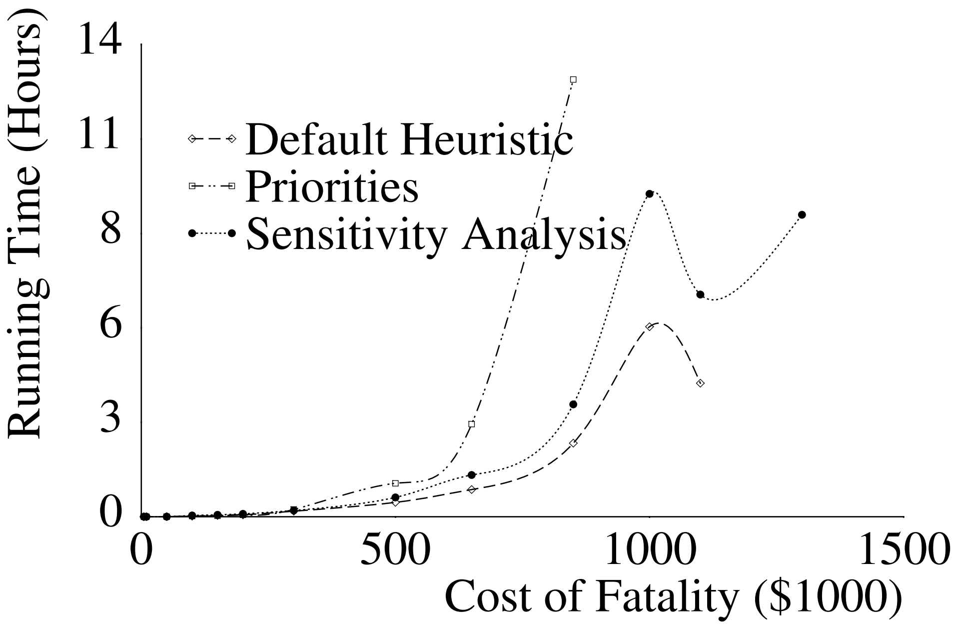

Figure 7.21: Plot of the CPU time versus the cost of fatality for the DVT domain with recursive tests and maybe_wait actions.

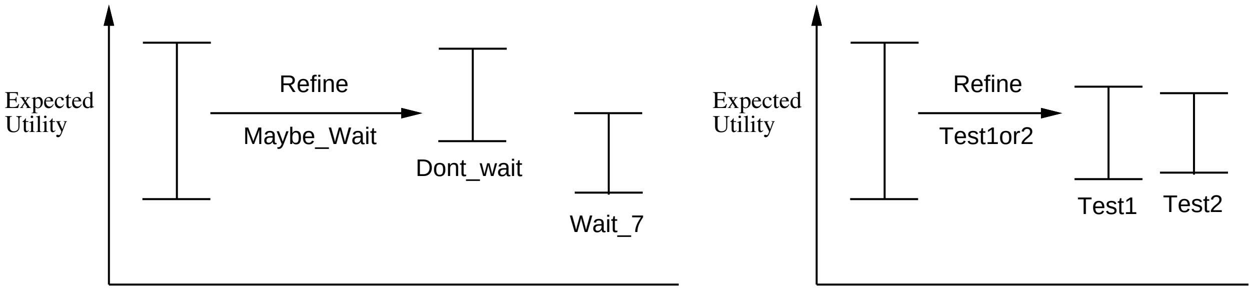

Figure 7.22: Refining a maybe_wait action results in two sub-plans, one where the upper bound on expected utility is largely unaffected.

the upper bound is sensitive to the refinement. To see why this is a problem, consider a plan with only two abstract actions remaining the be refined, a maybe_wait action and a Test1or2 action. The effect on the bounds on expected utility for refining the maybe_wait action and for refining the Test1or2 action are illustrated in figure 7.22. If the maybe_wait action is refined, then one of the sub-plans will have an upper bound on expected utility that is largely unaffected. Since this plan will very likely have the highest upper bound on expected utility, it will be selected for further refinement. The Test1or2 action will have to be refined. On the other hand, if the Test1or2 action is refined first, the upper bound on expected utility for both sub-plans is lowered. The upper bound on expected utility for each of these plans will be very close to the expected utility for primitive plan that includes the selected test and the dont_wait action, since refining the maybe_wait action would again have very little affect on $\overline{EU}$ for the sub-plan with the dont_wait action. The plans that result from refining the Test1or2 action will only be selected for refinement if they are vary nearly optimal. If they are not nearly optimal, they will be pruned when an optimal plan is found. The net result is that refining the maybe_wait action will almost certainly require the planner to refine the Test1or2 in one of the resulting sub-plans, whereas refining the Test1or2 action will only require the planner to refine the maybe_wait action if the plan is the optimal or very nearly optimal plan.

The priority scheme can be modified so that it only refines dont_wait actions after all the other abstract actions have been refined. This allows it to avoid refining the dont_wait actions, except for those in nearly optimal plans. However, if the domain designer recognized this type of situation, it would be easier to remove the maybe_wait actions from the domain description and speed up planning, independent of the meta-level control scheme in use.

The results of the comparison between the three actions selection methods is detailed in table 7.7 and shown graphically in figures 7.20 and 7.21. The addition of the "maybe_wait" action causes the planner to evaluate significantly more plans to find the same optimal plans for each cost of fatality. The performance of the default method on the domain with the "maybe_wait" actions gets significantly worse for high costs of fatality. To find the plan with the highest expected utility for a high cost of fatality, the planner needs to evaluate plans with sequences of "maybe_wait_retest" actions. Since the default method selects the first abstract action in a plan for refinement, it will alternate between selecting a "maybe_wait" action and a Test action in plans with sequences of "maybe_wait_retest" actions. For a $850,000 cost of fatality, the default method evaluates 3.5 times as many plans for the domain with the "maybe_wait" action. The number of plans that the priority scheme evaluates is also adversely affected by the addition of the "maybe_wait" action, but it does slightly better because it expands test actions before "maybe_wait" actions. For a

$850,000 cost of fatality, it evaluates 3.3 times as many plans when the "maybe_wait" action is added. The sensitivity analysis method is most adversely affected by the addition of the "maybe_wait" action. It evaluates 4.8 times as many plans when the cost of fatality is $850,000. The reason for degraded performance is that the sensitivity analysis prefers refining the "maybe_wait" action that is not very useful for finding a plan with the highest expected utility. Even so, the sensitivity analysis still refines fewer plans than the other methods. However, the difference in the number of plans evaluated is not enough to compensate for the overhead of performing the sensitivity analysis.

The test results presented in this section show that the sensitivity analysis method for selecting actions within a plan for refinement provides good performance over a range of problem in a variety of encodings of the DVT domain. It significantly out performs the other methods in term of the number of plans evaluated when solving difficult problems although this difference is not always sufficient to overcome the overhead of doing the sensitivity analysis. One method of reducing the overhead would be to approximate the sensitivity analysis by using only some of the chronicles. A partial sensitivity analysis would trade the quality of the estimates for overhead time.

7.6 Robot-Courier Tour Planner

For the robot-courier, the selection of which refinement to do for a particular tour is fixed, given the two-opt algorithm for tour improvement. Modifying the planner to allow it to dynamically switch between edge-exchange algorithms would allow the planner to avoid some local minima and would affect the rate at which the tour was being improved. Some work has been done by Pedro de Souza in selecting which edge-exchange algorithm to use and when to switch between them [de Souza, 1993].

The fundamental difference between the robot-courier tour planner and the other planners is that only a single child is produced when a tour is refined. Switching pairs of edges could produce $O(n^2)$ children, where n is the number of locations. If all the children were added to the search tree, then on each iteration, an optimistic selection strategy would choose to refine the tour that had the most improvement in the previous iteration. Repeating this process would lead to a sequence of refinements where the child that produced the best improvement would continually be selected for further refinement. The process would continue until swapping pairs of edges did not improve the tour and the planner would be forced to backtrack. By keeping only a single child at each step, the two-opt algorithm eliminates the possibility of backtracking. It also eliminates the need to keep information about backtracking points. The tour planner thus trades completeness for efficiency.

Chapter 8

Commencing Execution

Previous chapters were concerned with efficiently solving the problem of finding a plan with the highest expected utility. In this chapter, we look at efficiently accomplishing the task at hand, which includes the utility of both planning and execution. Achieving this goal may entail beginning execution before the plan with the highest expected utility is found. The question of when to start executing the current best plan is crucial for creating agents that perform tasks efficiently. The more time and resources spent creating and optimizing a plan, the longer the actions of the plan are delayed. The delay is justified if the improvement in the plan more than offsets the costs of delaying execution. This tradeoff is basic to creating resource-bounded rational agents. Since any real agent will have only limited computational resources, an agent demonstrates rationality if it performs optimally given its computational limits.

Deciding when to begin execution is complex. There is no way to know how much a plan will improve with a given amount of planning. The best that can be achieved is to have some type of model of the possible results of further planning, as is done in the anytime planning framework [Dean and Boddy, 1988]. In addition, the decision to begin execution should take into account the fact that it may be possible to execute one part of the plan while planning another. Traditionally, the decision of when to begin execution has been cast as a choice between planning and execution. However, this formulation of the problem ignores the possibility of overlapping planning and execution. By taking the possible overlap into account, we show how performance can be improved in some situations.

In this chapter, we examine a range of approaches to deciding when to begin execution from reactive approaches to deliberative approaches. Of particular interest from a meta-level control point of view are on-line techniques. We describe a proposed idealized algorithm for on-line control and show how it ignores the possible overlapping of planning and execution. We then develop a revised idealized algorithms that removes this limitation. The two idealized algorithms are then compared by using each one as a basis for implementing execution control in a simplified robot domain. Empirical results from this domain show that the revised algorithm can lead to improved performance. We conclude the chapter by looking at commencing execution for the DRIPS and Xavier planners. The Xavier route planner can find the optimal route quickly, using the meta-level control strategies we suggest in the previous two chapters. For this planner, it is best to completely plan the route before beginning execution. The DRIPS planner is a domain independent planner that can be used on-line or off-line. When used off-line, the planner typically produces policies that are used repeatedly. The time cost of planning can then be amortized over the number of times the policy is used. For on-line applications, the time needed for planning needs to weighed against the expected improvement in performance on a single run. For the DRIPS planner, we have implemented an on-line execution control strategy that weights the expected cost of computation against the opportunity cost of forgoing more planning. We show how tacking into account the time cost of computation better reflects the true nature of off-line planning and helps solve practical problems associated with infinite loops and computations that never terminate.

8.1 Approaches

The classical AI planning approach to deciding when to begin execution is to run the planner until it produces a plan that satisfies its goals and then to execute the plan. This approach recognizes that finding the plan with the highest expected utility may take too long and that using the first plan that satisfies all the constraints may produce the best results [Simon and Kadane, 1974]. It also relies on the heuristic that in searching for a plan, a planner would tend to produce simple plans first and simple plans tend to be more efficient and robust than unnecessarily complex plans. Execution is delayed until a complete plan has been generated. Only when there is a complete plan can there be a guarantee that an action will actually be needed and will not lead to a situation where the goals are no longer achievable.

In contrast to the classical AI approach, the reactive approach is to pre-compile all plans and to always perform the action suggested by the rules applicable in the current situation. The answer to the question of when to begin acting is always “now” [Brooks, 1986, Schoppers, 1987]. This approach is applicable when it is possible to pre-compile the plans and in dynamic environments where quick responses are required.

A third approach is to explicitly reason about when to start execution while planning is taking place [Russell and Wefald, 1991]. Using information about the task, the current state of the plan and expected performance of the planner, the decision to execute the current best action is made on-line. This allows execution to begin before a complete plan is created and facilitates overlapping of planning and execution. This approach also allows execution to be delayed in favour of optimizing the plan, if the expected improvement so warrants.

8.2 Idealized Algorithms

In this section, we present Good’s idealized algorithm for deciding when to begin execution and our revised version of the algorithm that accounts for overlapping planning and execution. Both algorithms are idealized because they assume that the decision maker knows the effect and duration of each of the possible computations. Since this information is generally not available, we need to approximate it in order to create an operational system. In the next section we operationalize both algorithms and compare the results empirically.

Making on-line decisions about when to begin execution can be modeled as a control problem where a meta-level controller allocates resources and time to the planner and decides when to send plans to the execution controller for execution. The control problem is to select actions and computations that maximize overall expected utility. Good's idealized algorithm, which has been suggested by a number of researchers, selects the action from the set $\{\alpha, C_1, \ldots, C_k\}$ that has highest expected utility [Good, 1971, Russell and Wefald, 1991]. In this list, α is the current best action and the C_i's represent possible computational actions. The utility of each computational action must take into account the duration of the computation by which any act it recommends would be delayed. Of course, this ideal algorithm is non-operational since the improvement that each computation will produce and the duration of the computation may not be known. A practical implementation will require these values to be estimated.

8.2.1 Revised Idealized Algorithm

Good's idealized algorithm ignores the fact that many agents can act and compute concurrently. A better control question to ask is not whether the agent should act or compute, but whether the agent should act, compute, or do both. The corresponding control problem is to select from the list $\{(\alpha, C_1), \ldots, (\alpha, C_k), (\phi, C_1), \ldots, (\phi, C_k)\}$ where each ordered pair represents an action to perform and a computation to do. ϕ represents the *null* action corresponding to only computing. This is no "act only" pair since acting typically involves (non-planning) computation (e.g. reactive control) that requires some or all the computational resources.

The advantage of considering pairs of actions and computations is that it allows the controller to accept the current action and forgo some expected improvement in favour of using the computational resources to improve the rest of the plan. For example, I may be able to improve the path I am taking to get to my next meeting that saves me more time that the path planning takes. However, I might be better off if I go with the current plan and think about what I am going to say when I get to the meeting instead.

8.3 Robot-Courier Tour Planner

In this section, we show how to operationalize both Good's original idealized algorithm and our revised idealized algorithm for use by the robot courier. The task of the courier robot is to visit a set of locations and return to the initial location as quickly as possible (see section 5.6 for details). Any random ordering of locations is a valid tour, but some tours require more travel than others. The robot can use a simple edge exchange algorithm (two-opt) to improve its tour, but this computation takes time. The decision to commence execution involves a tradeoff between reducing the length of a tour through more computation and delaying the start of execution.

We begin by characterizing the performance of the two-opt algorithm using a performance curve, as is done in anytime planning [Dean and Boddy, 1988]. We fit a curve to the empirical results of running the planner on multiple problems and use the curve to predict the results for new problems. This curve is used to supply the information needed to make each of the idealized algorithms operational.

The expected improvement in the tour as a function of computation time for the two-opt algorithm is accurately described by equation 8.1. In the equation, n is the number of locations and t is the elapsed time spent improving the tour. The parameters A and B are determined empirically and depend on the implementation and the computer used. A is a scaling factor that depends on the units used to measure distance. Measuring distances in meters rather than kilometers would increase A by 1000. B depends on the relative speed of the computer. Doubling processor speed would decrease B by one half.

$$I(t, n) = nA(1 - e^{-Bt}) \tag{8.1}$$

We can differentiate this equation to get the rate of improvement as a function of the elapsed time:

$$\overline{I}(t, n) = nABe^{-Bt} \tag{8.2}$$

The original idealized algorithm suggests either computing or acting, whichever one produces the best expected result at a given point in time. This corresponds to an execution control strategy that commences execution when the rate of tour improvement $\overline{I}(t, n)$ falls below the rate at which the robot can execute the tour (Rexe).

$$\overline{I}(t, n) = nABe^{-Bt} = Rexe \tag{8.3}$$

If the rate of execution (Rexe) and the parameters of the curve (A and B) are known, then the start time is given by:

$$t_{start} = -\frac{\ln\left(\frac{Rexe}{nAB}\right)}{B} \tag{8.4}$$

That is, the robot courier should optimize until t_{start} and then begin execution. We will call this the anytime method for deciding when to begin execution since it is based solely on the performance curve of the two-opt algorithm, an anytime algorithm.

8.3.1 Step-Choice Algorithm

The decision algorithm based on the original idealized algorithm ignores the possible overlap of planning and execution. Taking this overlap into account gives a different decision procedure that can improve performance under some circumstances. When taking the overlap into account, our algorithm will make a choice about whether to begin execution of the next step or delay execution until the planner has had more time to improve the tour. We call our algorithm the step choice algorithm because it make discrete choices at each step rather than calculating a t_{start} before beginning execution. In this section, we give the mathematical basis for the step choice algorithm. We begin by outlining some assumptions that we make in order to simplify the problem. We then give the formula for deciding when to begin execution of the next action. This formula can not be solved in a closed form, but can be approximated. We conclude this section with a discussion of the implications of the step choice algorithm.

In order to simplify the decision problem for the robot courier, we introduce three assumptions. We assume that computation is continuous and interruptible and the expected future behaviour is predicted by the performance curve given in equation 8.1. This assumption is also implicitly used by the anytime method given in the previous section. In addition, we also assume that actions are atomic and that once started, must be completed. The reason for using this assumption is to avoid complications associated with interrupting an action in order to start another. Typically, interrupting an action incurs some communications overhead and some effort to put things in a safe state before abandoning the action to begin another. By treating actions as atomic, we avoid the problem of modeling the cost of interrupting an action. We also avoid considering optimizations that involve interrupting the action. Computing the value of such an optimization may depend on how much of the action has been accomplished, which may introduce more communications overhead between the planner and the executer. For example, the distance from the current location to another location is constantly changing as the robot courier executes part of a tour. As a result, the value of interrupting the current action and heading towards a new location is also constantly changing and the robot would have to know where it was going to be when an action was interrupted in order to evaluate the usefulness of interrupting an action. Finally, we assume that execution does not affect the amount of computation available for tour improvement. This is realistic if tour improvement is carried out on a separate processor from the one used to monitor execution. If the same processor is used for both tour improvement and execution monitoring, then the assumption is almost true if the monitoring overhead is low. If this assumption does not hold, then our algorithm can be easily modified to account for the degradation in planner performance while executing. We make this same assumption when running the anytime algorithm and comparing it to step-choice algorithm in the next section.

Our revised, idealized algorithm suggests that the robot should begin execution when the expected rate of improvement due to tour planning alone is less than the expected rate of improvement due to planning and executing. As with the anytime algorithm, we characterize the expected improvement in the tour using a performance profile. The rate of tour improvement for computing alone is given in equation 8.5. When the robot chooses to act and compute, the robot is committing to the choice for the duration of the action, given our assumptions about atomic actions. The appropriate rate to use when evaluating this option is not the instantaneous rate but the average rate over the duration of the action. Equation 8.6 gives the average expected rate of accomplishment for acting and computing. In this equation, $\triangle t$ is the duration of the first action in the tour. Equating 8.5 and 8.6 and solving for $\triangle t$ gives an expression for the time duration of a move the robot would be willing to execute as a function of time spent computing. Unfortunately, the resulting expression has no closed form solution[1]. If the integral in equation 8.6 is replaced by a linear approximation, an expression for $\triangle t$ can be found (equation 8.7). Using the linear approximation over-estimates the rate of computational improvement of the move and compute option and biases the robot towards moving. On the hand, equation 8.6 only approximates the rate of optimization after taking a step. It under-estimates the effect of reducing the size of the optimization problem. Since two-opt is an $O(n^2)$ algorithm per exchange, reducing the problem size by 1 has more than a linear affect, as equation 8.1 suggests. Since the estimates err in opposite directions, they tend to cancel each other out.

$$\begin{aligned}
Compute\ only\ rate &= \overline{I}(t,n) = nABe^{-Bt} && (8.5)\\
Compute\ and\ Act\ rate &= Rexe + \frac{1}{\triangle t}\int_t^{t+\triangle t}\overline{I}(t,n-1)dt && (8.6)\\
&\approx Rexe + \frac{1}{2}(\overline{I}(t,n-1) + \overline{I}(t+\triangle t, n-1)) \\
\triangle t &= -t - \ln(\frac{(n+1)ABe^{-Bt} - 2Rexe}{(n-1)AB})/B && (8.7)
\end{aligned}$$

Examining the form of equation 8.7, it is clear that $\triangle t$ is infinite if the argument to the $ln()$ function reaches zero. This corresponds to an expected rate of tour improvement about twice the rate of path execution (Equation 8.8). At this point, the marginal rate of only computing is zero and the robot should execute any size step remaining in the plan. This suggests that t_{start} should be chosen so that the anytime strategy waits only until the rate of improvement falls below twice the rate of execution, ($\overline{I}(t,n) = nABe^{-Bt} = 2 * Rexe$). We call this the "Anytime-2" strategy for obvious reasons.

$$\begin{aligned}
\triangle t = \infty \ &\Rightarrow \\
0 &= (n+1)ABe^{-Bt} - 2Rexe = \frac{(n+1)}{n}\overline{I}(t,n) - 2Rexe \\
\Rightarrow \overline{I}(t,n) &= 2Rexe\frac{n+1}{n} \approx 2Rexe && (8.8)
\end{aligned}$$

In the range where $\triangle t$ is defined, increasing the rate the execution (Rexe) increased the size of an acceptable step. Similarly, increasing the rate of computation (B) decreases the size of the acceptable action. In the limit, the acceptable step correctly favours either always computing or always acting, as suggested by the relative speed of computing and execution.

In between the extremes, the robot courier will take an action whenever the time to perform it is less than $\triangle t$ and does not necessarily delay execution. Does it make sense to perform an action even when the planner is currently improving the plan more than twice as fast as the robot can execute it? It does if the action is small enough. The

[1]The result involves solving the omega function.

amount of potential optimization lost by taking an action is limited by the size of the action. At best, an action could be reduced to zero time (or cost) with further planning. The opportunity cost of forgoing more planning by executing an action is thus limited by the size of the action. Performing short actions at the beginning of the tour removes them from the optimization process and has the advantage that the optimizer is faced with a problem with fewer locations, but nearly the same length tour and nearly the same opportunity for improvement.

Using an on-line decision strategy that executes actions that take less than $\triangle t$ allows the robot to be opportunistic. If at any point in the tour improvement process the current next move is smaller that $\triangle t$, then the agent can begin moving immediately. In this way, the agent takes advantage of the planner's performance on the given problem rather than relying only on its expected performance. If the robot gets lucky and the initial random tour contains short actions at the start of the tour, then the robot can immediately begin execution. This suggests that robot might be better off by generating an initial tour using a greedy algorithm and then using two-opt rather than using two-opt on an initial random tour. In our empirical comparison, we ignored this potential optimization to avoid biasing the results in favour of the step choice algorithm.

8.3.2 Empirical Results

To empirically evaluate the utility of the control strategies given in the previous section, we implemented a robot-courier simulator and performed an set of experiments. The five control strategies given in figure 8.1 were each run on a sample of 100 randomly generated problems each with 200 locations to be visited. The experiments were performed on a DecStation 3100 using the Mach 2.6 operating system.

Each exchange or iteration of the two-opt algorithm is $O(n^2)$ where n is the number of locations considered. When optimization and execution are overlapped, the size of the optimization problem shrinks as execution proceeds. Each optimization step cannot be treated as a constant time operation as was done in [Boddy, 1991b]. For these experiments, we measured the actual CPU time used.

The graph in figure 8.2 shows the performance of each algorithm relative to the anytime-1 control strategy for a range of robot speeds. Each line on the graph is produced by subtracting the time the robot needs to complete the task using the anytime-1 control strategy from the time used by each strategy. A point above the horizontal axis indicates that a strategy took more time than the anytime-1 control strategy while a point below the axis indicates that a strategy took less time than the anytime-1 control strategy. Points below the x-axis indicate better performance. The graph covers a range of robot speeds where always acting is best at one extreme and always computing is best at the other.

The performance of the always-act and the always-compute strategies, at either extreme, are as expected. At very fast robot speeds, always acting is better than always computing and at very slow robot speeds, the converse is true. The anytime strategies and the step choice algorithm perform well for the entire range of speeds. At relatively fast robot speeds, these strategies correctly emulate the always-act strategy. Although, the step choice algorithm

1. Always Compute : Complete the entire two-opt algorithm and then begin to act.
2. Always Act : Always perform the next action and run the two-opt algorithm in parallel.
3. Anytime-1 : Compute until $\overline{I}(t, n) = Rexe$ and then act and compute concurrently.
4. Anytime-2 : Compute until $\overline{I}(t, n) = 2 * Rexe$ and then act and compute concurrently.
5. Step Choice : Execute the next action in the tour whenever $\text{time}(\text{action}) \leq \triangle t$.

Figure 8.1: Tour improvement Meta-Level Control Algorithms.

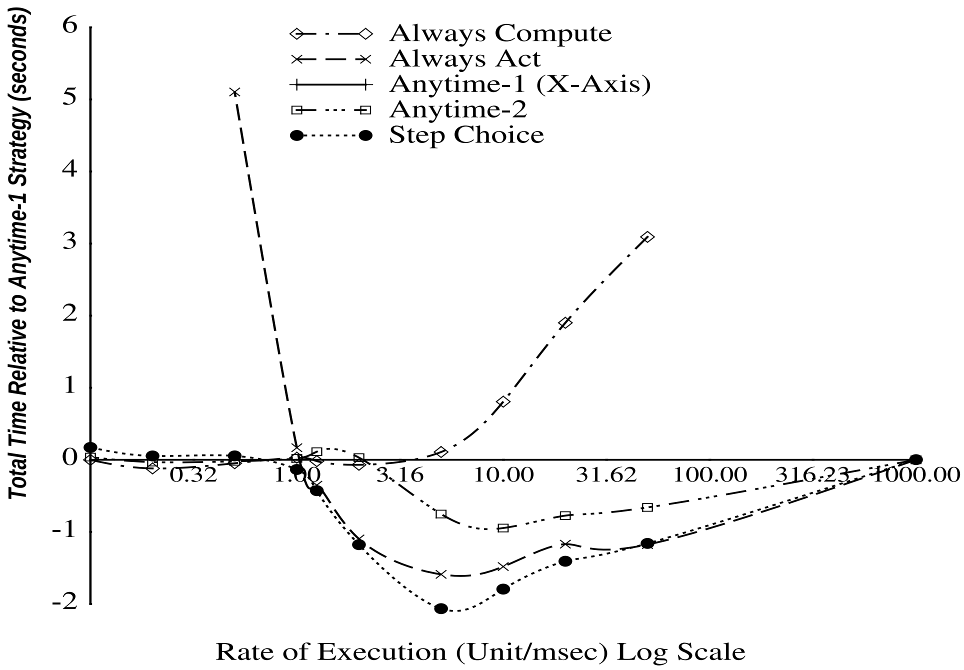

Figure 8.2: Robot-courier performance relative to the Anytime control strategy.

does not perform quite as well because of the overhead of doing on-line control. The our implementation, the overhead is approximately 2.5% of the total processing time.

At relatively slow speeds, the anytime and step choice strategies correctly emulate the always-compute strategy. Between the extremes, the anytime-2 strategy outperforms the anytime-1 strategy by starting execution earlier and better overlapping execution and optimization. The step choice algorithm produces the best results over most of the range of robot speeds. It outperforms the anytime-2 strategy by opportunistically taking advantage of small initial step in the tour plan to begin execution earlier. Only at very fast robot speeds does the anytime-2 strategy do better. This is due to the overhead of doing on-line meta-level control.

It is interesting to note that even at relatively fast robot speeds, the behaviour of the always-act and the step choice strategy are not identical. The step choice algorithm will occasionally pause if the next action to be taken is large. This allows the planner an opportunity to improve the next step before it is taken. The effect of this is to reduce the average length of the tour slightly while maintaining the same performance.

None of the strategies made full use of the computational resources available. For the methods that overlap computation and acting, the computation completed before the robot reached the end of the path. The two-opt algorithm terminates when exchanging any pair of edges does not improve the tour. The resulting tour represents a locally minimal tour, but may not be a globally minimal tour. After completing the two-opt algorithm, the computational resource was then unused for the rest of the run. This extra computation resource could have been used to apply other optimization algorithms to the remaining tour to further improve performance.

8.3.3 Analysis

The empirical results from the robot-courier tour planner show a slight improvement due to taking the overlapping of planning and execution into account when deciding when to begin execution. The question we consider in this section is how much of an improvement is possible. We present an argument that the improvement is limited to doubling performance by reducing the total time by a factor of 2.

Consider a task where the planner is improving a plan at a rate just slightly faster than the rate at which the plan can be executed ($\overline{I}(t,n) = Rexe + \epsilon_1$). Good's original algorithm would continue to optimize the plan, but not execute any part of it. The step-choice algorithm would attempt to do both plan and execute. The rate of task achievement is $\overline{I}(t,n-1) + Rexe = 2 * Rexe + \epsilon_1 - \epsilon_2$, where $\epsilon_2 = \overline{I}(t,n) - \overline{I}(t,n-1)$. The ratio of the rates of task accomplishment approaches 2 as ϵ_1 and ϵ_2 approach zero. If the rate of plan improvement remains relatively constant over the duration of the task, then the revised algorithm performs the task almost twice as quickly as the original algorithm. It is only when the two rates, improvement and execution, are almost equal that considering overlapping has a significant effect. If either rate is much higher than the other, then the relative improvement is not as significant.

8.3.4 Limitations of the Step Choice Algorithm

The step-choice meta-level control algorithm works for the simplified robot courier because the cost of undoing an action is the same as the cost of performing the action. In fact, because of the triangle inequality, it is rare that the action has to be fully undone. The robot courier would only retrace its path if the locations were co-linear and the new location was in the opposite direction, since the shortest route proceeds directly to the next location. At the other end of the spectrum are domains such as chess playing where it is impossible to undo move[2]. In between, there are many domains with a distribution of recovery costs. Consider, as an example, leaving for the office in the morning. Suppose that it is your first day at work, so there is no pre-compiled, optimized plan yet. The first step in your plan is to walk out the door and then to walk to the car. Suppose you do this and while your are walking, your planner discovers that it would be better to bring your brief case. The cost of recovery is relatively cheap. Just walk back in the house and get it. On the other hand, if you forgot your keys, you may now be locked out of the house. In this sort of environment, using only the cost to do an action may not be the best strategy. The decision to begin execution must consider not only the time and resources needed to execute an action but also the time and resources needed to undo an action and probability that the action will need to be undone.

8.4 Xavier

The on-line execution control algorithm developed in the previous section allows an agent to adapt to specific problem instances and is effective for a range of relative computation speeds. If a faster processor or a faster robot are substituted, the algorithm adapts, if the parameters and performance curves are updated appropriately. In many situations, the hardware used for computation and execution is fixed and the relative speeds are significantly different. Such is the case with the Xavier robot that has three relatively powerful computers, but moves at about a walking pace. In this section, we examine some of the practical issues associated with execution control for a real robot.

The Xavier robot moves at about 40 cm/second, which is roughly the speed of a leisurely walk. The time needed to find an optimal route is only 0.13 seconds on average. If the robot could begin moving in the correct direction without any delay, then it would save 0.13 seconds on average and travel 5.2 centimeters in the time needed to generate the plan with the highest expected utility. This is negligible compared to the time needed to execute the average path in Wean Hall, which is a few minutes. The important aspects of execution control for this domain are reducing the computation needed to the point where it is negligible. The methods developed in the previous chapters for selecting a plan for refinement and for selecting the part of the plan to refine lead to the short computation times. The result of planning is a partial plan that is incomplete, but provably optimal. The interesting control question for the Xavier planner is how much of this partial plan should be fully elaborated before execution begins. The time to fully elaborate a plan may be many times the time needed to select the best partial plan since the planner would have to generate contingency plans for all possible contingencies and generating each of these plans can take as long as generating the initial partial plan. Obviously, the first part of

[2] You may be able to move a piece back to its previous location, but only after your opponent has had an opportunity to move, so the board will not be in the original configuration unless your opponent also undoes his move.

the route must be elaborated so that execution can begin, but should the robot attempt to elaborate the entire plan, including all possible contingencies before execution begins or delay planning for some contingencies until they arise?

In determining that a particular plan is optimal, the planner will necessarily elaborate some parts of the plan. When using sensitivity analysis to guide refinement selection, the elaborated parts of the plan will tend to include those that have the largest impact on the expected utility bounds. These parts will generally include the most likely used routes and the largest areas (corridors and rooms) on these routes. The nominal path from the robot to the goal tends to be nearly completely elaborated, with an average of 94.8% of the nodes on this route expanded. To begin execution, only the first link on the route needs to be planned and the rest could be elaborated as the robot moved. This approach could lead to problems coordinating planning and execution. Since the robot has only a probability distribution over its location, it is not clear when the robot finishes one segment of the route and begins another. The elaboration of the route and the seeding of the action policy needs to proceed well ahead of the robot. To simplify the problem, the planner currently elaborates all sections on the nominal route from the robot to the goal. The nominal route is the route the robot tried to follow, but may be blocked from following it if the robot encounters a closed door. Elaborating the links on the nominal route allows a complete action policy to be created.

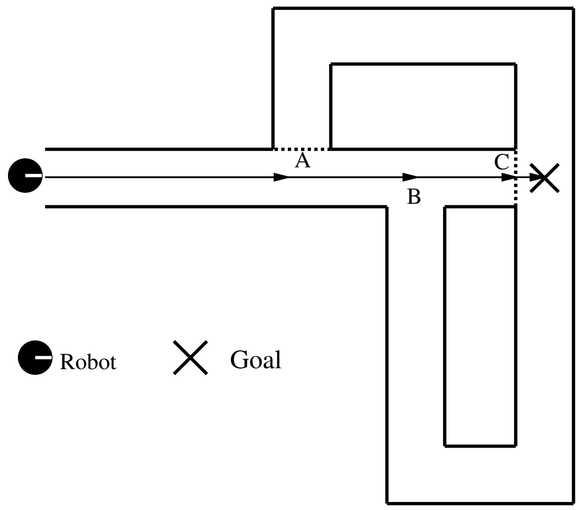

Figure 8.3: The state of the door at A is observed on the way to C, so delaying contingency planning can take advantage of the observation.

The resulting plan passed to the executer may still be only partially complete since it does not necessarily have contingency plans for closed doors the robot may encounter. The robot could plan for these contingencies while following the nominal route, but the utility of planning for these contingencies in advance is unclear. Consider the simple example in figure 8.3 where the nominal route travels straight towards the goal, passing through a door just before it reaches the goal. If the door is closed, the robot must backtrack and take either the upper or lower hallway at A or B respectively. When the robot begins following the nominal route, it does not know the state of the door at A and would have to plan the recovery from C based only on the probability that door A is open. But, by the time the robot has reached C, it will have observed the state of door A and could use a much better estimate of the state of door A to create the contingency plan. There are two obvious ways of dealing with this problem: to delay planning and to create multiple contingency plans, one for each set of possible observations.

The simplest method for dealing with actions whose values depend on future observations is to delay planning until the observations have been made or until that part of the plan is needed for execution. This approach has the advantage that no unnecessary planning is done and contingency plan can take into account all observations to that point. The disadvantages is that the robot does not have a plan for reacting to the contingency. In the Xavier example, pausing for a second in front of a closed door in order to create a new plan does not pose a problem, and the robot does not plan for contingencies until they arise.

The second method for dealing with observations that affect the utility of contingency plans is to create a plan for each possible observation. In the robot example, each door has only two states, open and closed, so this approach doubles the number of contingency plans for each door that can be observed. In general, the number of contingency plans increases by O^n where O is the number of observation outcomes and n is the number of observations. The planner could create contingency plans based on each possible combination of observations and store the results for possible use when needed. Of course, only the single plan that corresponded to the actual observations may eventually be used and the rest are discarded, wasting the computation used to generate them. This is fine if the computation was not going to be used anyway and there was enough time to generate all the contingency plans. Otherwise, the planner needs to select which contingencies to generate, focusing on those that are most likely to be used and those for which having a contingency plan versus replanning on-line makes the biggest difference. For example, in the Xavier domain, the planner could create contingency plans only for the most likely observations, since these are the most likely to be useful and the lack of a contingency plans does not create a crisis. However, if one of the possible observations was a fire in the boiler room, having a contingency plan for this critical, but unlikely situation, may be very useful.

In general, a planner should create contingency plans for likely observations where there is a significant advantage to having a plan on hand versus generating one on-line. For the Xavier domain, there is no significant advantage to having a contingency plan on hand. When a closed door is encountered on a route, the robot stops and replans given its current beliefs, which includes the observations on the way to the closed door.

8.5 DRIPS

Unlike the Xavier planner, the DRIPS planner is typically used for off-line planning where a plan is created well in advance of its use and may be used repeatedly. Such is the case in the DVT domain where the DRIPS planner is used to evaluate treatment policies that could be used for thousands of patients.

It is typical to view off-line planning as not requiring a decision about when to being execution, but this is true only if the best plan can be generated before it is needed and computation is free. Most off-line planning is not truly off-line since there is not an infinite amount of computation or time available to produce a plan. Typically, off-line planning means that the plan must be available before it is needed and that there is no interaction between the planner and the executer. If off-line computation is expensive or limited and the plan is required by some point in time, the planner may not be able to generate the best plan. If, for example, a factory uses a computer each week to plan the production schedule of the next week, the planning process needs to produce a high quality plan with one week of CPU time. This assumes that the computer is only needed for planning production. If the computer is also used for forecasting and management support, then doing more planning takes computation away from these activities. Alternatively, a company may buy CPU-cycles from a time sharing service and pay for the computation it uses. We take the vies that meta-level control for off-line planning should take into account the resources consumed during planning, which might be different from the resources available during execution. Taking the cost of off-line planning into account allows a planner to better use off-line resources and better models reality.

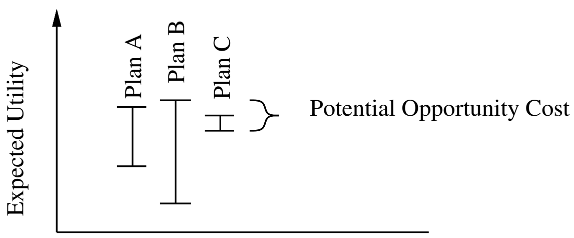

Figure 8.4: The potential opportunity cost is the difference between the highest upper bound and the highest lower bound, assuming that the planner would select the plan with the highest lower bound if it were forced to decide at this point.

In the DRIPS planner, the meta-level controller trades off potential opportunity cost for computation time. The bounds on the set of potentially optimal plans give the bound on he opportunity cost (figure 8.4). The planner uses its model of the time cost and likely affect of each refinement to estimate the value of each refinement and greedily selects the best one (section 7.5). If the value of the best refinement is less than the time cost of computation, the planner halts and returns one of the non-dominated sub-plans of the partial plan with the highest lower bound on expected utility. Generating one operational plan from the set of plans represented by a partial plan is relatively quick in the DRIPS planner.

In on-line planning, the utility function gives the time cost of computation since it determines the cost of delaying action execution for further planning. The utility function could be modified to cover off-line planning resource uses as well as run-time resource use. For example, we could include the cost of buying CPU-cycles from a time sharing company in the utility function. However, doing this might overly complicate the utility function. Keeping on-line and off-line resources and costs separate simplifies the problem. For off-line planning, the DRIPS planner has been modified to accept a plan-time cost-of-

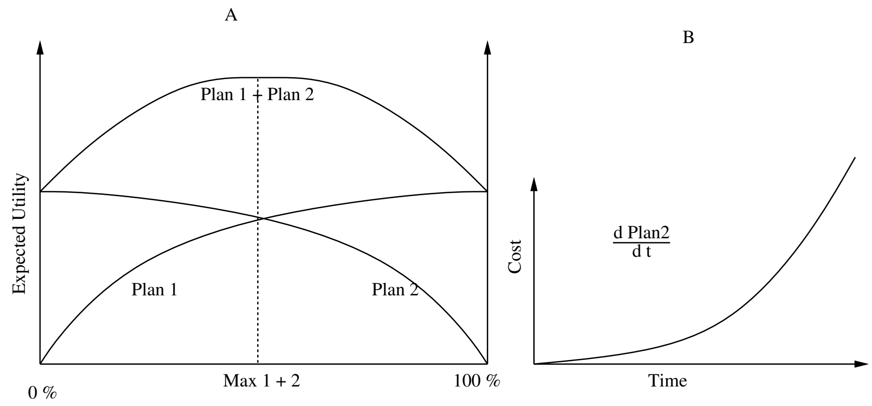

Figure 8.5: Utility versus computer time for two planning problems that share the same computer.

time function. Since off-line planning does not necessarily delay action, but does consume resources, this function reflects the cost of planning. Consider again the example where a manufacturer is buying CPU-cycles for planning. The plan-time cost-of-time function would represent the cost of purchasing more CPU-cycles.

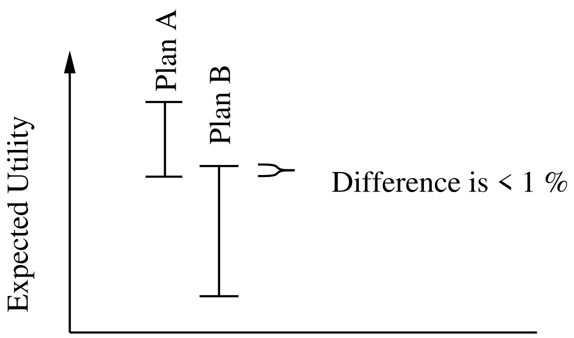

Figure 8.6: Plan A almost dominates plan B.

In addition to the cost of computation, there are practical concerns that should cause the DRIPS planner to halt planning. These include the limited accuracy of the domain model and infinite loops. The domain model and the utility function include parameters with limited accuracy since they are only estimates of any underlying ground truth. If these estimates are accurate only to within one percent, for example, then the planner is not justified in distinguishing between plans that differ by less than one percent in expected utility. If the bounds on the set of potentially optimal plans differ by less than one percent, then the planner can halt and return the entire set of plans. Similarly, if changing the bounds on a plan by one percent would allow it to be pruned (figure 8.6), then the planner can prune the nearly dominated plan. Even if the true value of plan A in figure 8.6 corresponded to its lower bound and that of plan B corresponded to it upper bound, the plans would essentially be equivalent in terms of expected utility. The planner would be equally justified in selecting either plan in this case. Since this is the best case scenario for plan B, the planner is justified in selecting plan A without further refinement. We have modified DRIPS planner to include the relative model accuracy in the domain description and uses this limit on accuracy to prune the search space.

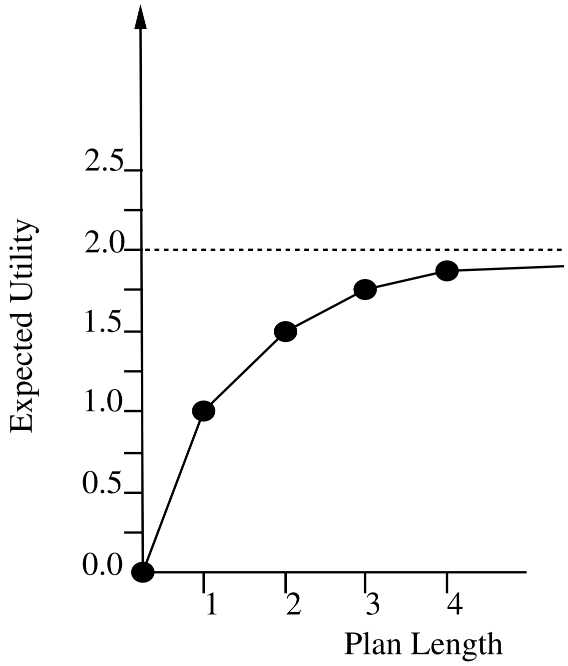

Figure 8.7: The expected utility approaches 2.0 as the number of attempted cup pickups increases.

Finally, the DRIPS planner needs to guard against infinite loops. Consider, for example, a domain where a robot is trying to pick up a cup. Suppose there is a single pick_up_cup action that succeeds with probability 0.5 and that the value of holding the cup is 4 and the cost of trying to pick it up is 1. The expected value of trying to pick up the cup once is 1. If the action fails, we are back in the original state. Assuming independence, the expected utility of two attempts to pick up the cup is 1.5. As the number of pick up attempts is increased, the expected utility asymptotically approaches the value of the infinite plan, 2.0 (figure 8.7). The planner cannot recognize that the plan with the highest expected utility is infinite and will attempt to unroll the loop indefinitely. There are two ways that the planner deals with infinite loops. In the cup example, the difference between the expected utility of the infinite plan and the increasingly long finite plans approaches zero and will eventually fall below the model accuracy for the domain. At that point, the planner will prune the infinite plan since it is not justified in trying to show that the infinite plan is better. Secondly, the time needed to evaluate a plan increases with plan length. Each unrolling of the loop increase the length of the plan and the time cost of evaluating it. Eventually, the time cost of evaluation will be larger than the opportunity cost and the infinite plan can be pruned. This allows the planner to return a nearly optimal plan rather than never returning any plan.

8.5.1 DRIPS Empirical Results

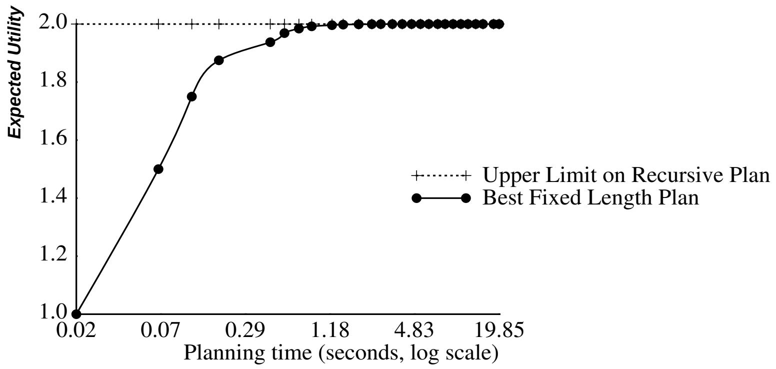

Figure 8.8: The expected utility approaches 2.0 as planning time increases.

To demonstrate our off-line methods for deciding when to terminate planning, we have implement the simple cup-pickup domain described in the previous section. In this domain, the robot has a single action which is to try to pick up a cup. If the action fails, the robot is back in the initial state. The optimal plan is to continue to try to pickup the cup until it is observed in the robot's hand. However, the DRIPS planner can not generate this plan. It can only approximate it will ever longer plans that include more and more attempts to pickup the cup. Without taking the cost of planning into account, the planner would never terminate.

Figure 8.8 shows the performance profile for the cup-pickup domain. If there were no limit on the model accuracy, then the planner would theoretically continue for ever. In practice, when the relative difference between the expected utility of the infinite plan and a finite plan became less than the precision of a double floating point number, the planner would stop. For this example, we use a model accuracy of 10^{08}. Using the model accuracy to decide when numbers are indistinguishable, the planner terminates with a plan that includes 28 pickup actions. The difference in expected utility between the plan with 28 pickup actions and the infinite plan is 7.45e-9.

Table 8.1 gives the results in more detail and shows how the rate of improvement changes over time. Depending on the cost of computation, the planner may terminate planning before the difference between the finite and infinite plans becomes less than the model accuracy. If the cost of computation is one unit per second, then the planner would terminate planning when the rate fell below one unit per second. In this example, the planner would terminate with a plan that had 5 pickup actions. For higher costs of computation, the planner would return shorter plan. Since the rate of plan improvement decays rapidly and since the time needed to evaluate the plan increase with plan length and the increase in expected utility for longer plans decreases geometrically. The planner is sure to eventually terminate for any fixed, positive cost of computation. As this example demonstrates, using model accuracy and the cost of computation ensures that the planner terminates.

Elapsed Time (sec)	Number of Pickups	Expected Utility	Improvement Rate
0.017	1	1.0000	5.88e00
0.067	2	1.5000	1.00e0
0.117	3	1.7500	5.00e0
0.184	4	1.8750	1.86e0
0.434	5	1.9375	2.50e-1
0.550	6	1.96875	2.69e-1
0.700	7	1.984375	1.04e-1
0.867	8	1.9921875	4.67e-2
1.217	9	1.99609375	1.11e-2
1.467	10	1.998046875	7.81e-3
1.900	11	1.9990234375	2.25e-3
2.367	12	1.99951171875	1.04e-3
2.750	13	1.999755859375	6.37e-4
3.334	14	1.9998779296875	2.08e-4
3.934	15	1.99993896484375	1.01e-4
4.600	16	1.999969482421875	4.58e-5
5.334	17	1.9999847412109375	2.07e-5
6.134	18	1.9999923706054687	9.53e-6
7.117	19	1.9999961853027344	3.88e-6
8.050	20	1.9999980926513672	2.17e-6
9.200	21	1.9999990463256836	8.29e-7
10.467	22	1.9999995231628418	3.76e-7
11.800	23	1.9999997615814210	1.78e-7
13.434	24	1.9999998807907104	7.29e-8
15.150	25	1.9999999403953552	3.47e-8
18.084	26	1.9999999701976776	1.01e-8
19.850	27	1.9999999850988388	8.43e-9
19.850	27	1.9999999850988388	8.43e-9
21.867	28	1.9999999925494194	3.69e-9

Table 8.1: Time versus expected utility for cup collection

Chapter 9

Conclusions

The combination of artificial intelligence techniques with decision theory provides a rich planning framework that allows a planner to make the tradeoffs necessary to solve real-world planning problems. Artificial intelligence techniques provide methods for representing actions and algorithms for generating plans. Decision theory, on the other hand, provided methods for evaluating outcomes and accounting for their likelihood. The advantage of combining these approaches has long been recognized. However, the potential benefits are lost if the planner cannot find solutions in a timely fashion. In this dissertation, we have examined the problem of providing effective meta-level control for decision-theoretic planners. The aim, as always, is to improve the performance of the agent making use of the plans. The appropriate strategy is either to find the plan with the highest expected utility using the least amount of computation or to quickly find a satisficing plan with high expected utility, trading planning time for plan quality.

Our approach models decision-theoretic planning using a decision problem framework. The four steps in solving a decision problem, plan generation, parameter estimation, plan evaluation and plan execution are basic to creating plans with high expected utility. Allowing a planner to iteratively approximate the complete solution to each of these steps makes the planner more efficient and able to produce satisficing solutions. The relevant meta-level control questions correspond to controlling the iterative approximation of each step in the decision problem. The three central meta-level control questions concerning plan generation, refinement guiding and commencing execution are the core topics of this dissertation.

The first step in solving a decision problem is to create the set of candidate plans. In approximating this step, a decision-theoretic planner creates plans one at a time and may not completely elaborate each plan. Using abstraction and not planning all possible contingencies can improve efficiency by reducing the amount of work wasted on low utility plans. A consequence of generating partial plans is that the plans have a range of possible expected utility. The meta-level control question is to choose between generating another plan and selecting one of the existing partial plans for refinement. We have shown both analytically and empirically that an optimistic strategy that always selects the plan with the highest upper bound on expected utility uses the least amount of computation to find a plan with the highest expected utility. Modifying the DRIPS planner to use our optimistic plan selection strategy improved its efficiency by an order of magnitude.

Once a partial plan has been selected for refinement, the planner must choose which part of the plan to refine. Ideally, the planner would choose refinements that help distinguish high utility plans from low utility plans with a minimum of computation. To do this, the planner would need to know how long a refinement would take and its effect on the plan's expected-utility bounds. This information is unavailable without actually doing the refinement, but can be estimated. We use a model of the planner's runtime characteristics to predict the computation needed and a sensitivity analysis to predict the effects of a refinement on the plans expected utility bounds. Selecting refinements that maximize the ratio of expected utility bounds sensitivity to expected computation time provides efficient and effective meta-level control. The results are comparable to, and in some cases better than, hand tuned meta-level controllers. The advantage of sensitivity analysis based control over hand tuned and domain specific heuristics is that our methods do not require extra effort on the part of the domain designer.

Any planner will have limited computational resources and cannot instantly generate the plan with the highest expected utility. The decision to begin execution with a possibly sub-optimal plan involves a tradeoff between the quality of the plan and delaying execution. Proposed idealized algorithms for making this tradeoff have ignored the possible overlapping of planning and execution. We created a revised, idealized algorithm that takes into account the possibility of overlapping planning and execution. The algorithm remains idealized because it require knowing the duration and result of each planning calculation before it is carried out. To implement our revised algorithm, we borrow techniques from anytime planning and create performance profiles to predict the duration and result of calculations. Applying our technique to a simplified robot-courier tour planner shows empirically that accounting for the overlap of planning and execution improves performance.

In examining the meta-level control problems for decision-theoretic planners, we have developed theoretically sound methods for making decisions. Where necessary, we approximate these control methods to create practical implementations. Implementing our techniques on a diverse set of four planners has also allowed us to validate our methods empirically.

The remainder of this chapter details the contributions of this dissertation and outlines future directions for research. The contributions include theoretical and empirical results as well as practical implementation of our techniques in four decision-theoretic planners. The directions for future work include extending our techniques to a wider range of planners and easing some of the simplifying restrictions we made when analyzing and implementing our meta-level control.

9.1 Contributions

Taxonomy of meta-level control decisions: Modeling the decision-theoretic planning process as an iterative approximation of a decision problem allows us to identify the relevant meta-level control decisions. These decisions are common to decision-theoretic planners, although only a subset of the decisions may be applicable to a particular planner, depending on its features and capabilities. Using this taxonomy of decisions as a basis for analyzing meta-level control for decision-theoretic planners makes the results generally applicable to all planners in this class.

The meta-level control questions relevant to decision-theoretic planners are:

Plan Generation:	Choose between generating a new plan and refining a partial plan.
Refinement Guiding:	When refining a plan, choose the part of the plan to refine.
Information Gathering:	Choose when to gather more information.
Contingency Planning:	Choose which contingencies to plan for.
Commencing Execution:	Choose when to begin execution of the best plan found so far.

Plan Generation: When a planner iteratively creates partial plans, the planner can calculate only a range of expected utility for each plan. The planner can also, at best, calculate only a limit on the expected utility of the plans yet to be generated. We have proven that an optimistic strategy that focuses effort on the plan with the highest upper bound on expected utility uses the least amount of work to find a plan with the highest expected utility. If the highest bound is for the set of ungenerated plans, then the planner chooses to generate a new plan. If a partial plan has the highest upper bound, then that plan is selected for refinement. If a complete plan has the highest upper bound, then the planner can terminate because it has found a plan with the highest expected utility. Our optimistic strategy has significantly improved the performance of the DRIPS planner and has been adopted as the default strategy by the creators of the DRIPS planner and the Pyrrhus planner.

The optimistic strategy is not necessarily optimal when looking for a satisficing solution. Instead, we have proposed weighing the potential opportunity cost against the cost of computation to address the plan generation problem. We have implemented this strategy in the DRIPS planner to allow the planner to produce satisficing solutions.

Refinement Guiding: When a plan is selected for refinement, the planner must decide which one of the a set of possible refinements to do first. We have shown how a planner could make this choice optimally, if it is given perfect information about the duration and result of each refinement. To perform optimally, the planner chooses refinements so that the sequence of refinements reduces the upper bound on expected utility to, or below, the bound for the best plan, with the least amount of work. The problem with this method, besides requiring unavailable information, is that determining the best sequence of refinements is an NP-hard problem. To make this method tractable, we approximate the algorithm by greedily picking refinements with the highest ratio of change in expected utility bounds to work. We make the algorithm operational by using a sensitivity analysis to estimate the effect of each computation and use a model of the planner to predict the duration. We have shown empirically that our sensitivity analysis based method for refinement guiding is effective in three planners used in four domains. Besides providing good runtime performance, our method also reduces the burden on the domain designer by eliminating the need for hand tuning.

Commencing Execution: Deciding when to begin execution has historically been framed as a choice between computing and acting. It was believed that if you knew the value and duration of each possible computation, you could decide optimally between acting and computing. We have shown that this approach assumes that an agent cannot both compute and act simultaneously. In removing this assumption, we were able to create a revised, idealized algorithm that could theoretically double performance in some situations. Empirically, we have shown that meta-level control based on the revised algorithm can produce measurable improvements.

9.1.1 Implementations

In addition to the theoretical and empirical results, our work has produced working implementations of two decision-theoretic planners: Xavier and the robot courier and has improved the capabilities and performance of a third, DRIPS. These implementation are available to the community for use and future research.

Xavier Route Planner: The Xavier route planner is implemented as part the Xavier robot software and is used to guide the Xavier robot around Wean Hall at CMU. The robot has been in regular use since December 1995 and has traveled over 60 kilometers while serving approximately 1500 requests [Simmons *et al.*, 1997]. Our meta-level control techniques contribute to the robot's success.

DRIPS: We have extended the DRIPS planner to handle recursive actions so that the planner can consider an infinite space of plans. This extension has removed a restriction that forced domains designers to hard code the number of possible action repetitions, such as repeated medical tests. In addition, we have implemented our sensitivity analysis based meta-level control in a domain independent way. Our meta-level control improves the efficiency of the planner and provides an example for implementing similar control in other planners.

Robot-Courier: The robot-courier tour planner provides an example implementation of the step choice algorithm for deciding when to being execution. Our implementation allows the relative speeds of the robot, the tour improvement computation and the meta-level controller to be varied in order to investigate the performance of the algorithm under various conditions. With this implementation, we were able to show the advantage of the step-choice algorithm over competing meta-level control algorithms.

9.2 Future Work

The work presented in this dissertation is but a step on the long road to exploring issues of meta-level control for decision-theoretic planners. Further work is needed to extend our work to other types of planners and to address meta-level questions we have left unanswered.

Stochastic Plan Generation: In the planners we have examined, plans are generated in a deterministic order and in such as way that we can guarantee a bound on the expected utility for the plans remaining to be generated. Another class of plan generators creates plans drawn randomly from a probability distribution over the set of plans. These planners can be analyzed stochastically but cannot provide a limit on the quality of the ungenerated plans. Our analysis and methods need to be extended to handle this class of planners.

Contingency Planning: One of the meta-level control questions that we answer only implicitly is when to make contingency plans. In searching for plans with high expected utility, we sometimes plan for contingencies that could significantly affect the value of a plan. However, this approach addresses only part of the problem. An agent may want to plan for some contingencies ahead of time because there will be no time to plan if the contingency should arise. In other situations, an agent might want to delay contingency planning because the information needed for planning is not yet available. The question of when to plan for contingencies and which contingencies to plan for needs to be addressed directly.

Information Gathering: The other meta-level control question that we cover only implicitly is when to gather information. We cover this question indirectly when considering plans that include sensing. But information gathering raises some of the same issues that contingency planning does about sequencing and delaying planning. An agent could delay planning until after a key piece of information is obtained in order to reduce the search space. An agent may also want to obtain information as early as possible in order to allow more time for planning. The problem with this strategy is that there is often a tradeoff between the time when information is gathered and its quality. Short term weather forecasts are more accurate, but allow less time to plan and react. More work is needed on coordinating information gathering and planning.

Quality of Estimates: In making our meta-level control decisions, we rely on estimates of the planner's future performance, both in terms of the duration of computation and its value. Better estimates should lead to better control decisions, but better estimates require more computation, which increases meta-level control overhead. Some initial work on evaluations this tradeoff suggests that very rough estimates are sufficient for meta-level control since making meta-level decisions correctly are most important when there are clear good and bad options. When choices are almost equally good, making a bad choice has less of an impact. More work needed to be done to quantify this tradeoff.

Commencing Execution: The step choice algorithm works for the robot-courier because the domain is benign and the robot cannot get trapped. In answering the question of when to begin execution for the general case, we need to consider the risks of acting too quickly and of delaying too long. Without sufficient look ahead, an agent may paint itself into a corner.

Conversely, delaying too long may make an agent someone else's dinner. The question of how to coordinate planning and execution in risky environments remains open.

9.3 Summary

Decision-theoretic planners that combine artificial intelligence planning techniques and decision theory have the potential to solve complex real work problems that involve uncertainty and tradeoffs. However, to be effective, these planners have to find solutions efficiently. In this dissertation, we address the meta-level control problem of allocating computation to make decision-theoretic planning efficient and effective. We provide methods to make three key meta-level control decisions related to the planning process: whether to generate more plans or refine an existing partial plan, which part of a partial plan to refine, and when to commence execution. We prove that an optimistic strategy that refines the plan with the highest bounds on expected utility first uses minimal computation when looking for a plan with the highest expected utility. When selecting which part of a plan to refine, our sensitivity analysis methods identify refinements that can quickly distinguish plans with high expected utility and effectively allocate processing time. For deciding when to begin execution, previous methods have ignored the possibility of overlapping planning and execution. By taking this possibility into account, our method improves performance by accomplishing a task more quickly. Combined, our techniques provide efficient and effective meta-level control for decision-theoretic planners.

Bibliography

[Bacchus and Kabanza, 1996] Fahiem Bacchus and Froduald Kabanza. Planning for temporally extended goals. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence (AAAI-96)*, pages 1215–1222, Portland, Oregon, August 1996. AAAI.

[Bäckström and Jonsson, 1995] Christer Bäckström and Peter Jonsson. Planning with abstraction hierarchies can be exponentially less efficient. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence (IJCAI-95)*, pages 1599–1604, 1995.

[Blythe, 1994] Jim Blythe. Planning with external events. In Ramon Lopez de Mantras and David Poole, editors, *Proceedings of the Tenth Conference on Uncertainty in Artificial Intelligence (UAI-94)*, pages 94–101, Seattle, Washington, July 1994.

[Boddy and Dean, 1989] Mark Boddy and Thomas Dean. Solving time-dependent planning problems. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence (IJCAI-89)*. AAAI, August 1989.

[Boddy, 1991a] Mark Boddy. Anytime problem solving using dynamic programming. In *Proceedings of the Ninth National Conference on Artificial Intelligence (AAAI-91)*, pages 738–743. AAAI, July 1991.

[Boddy, 1991b] Mark Boddy. Solving time-dependent problems: A decision-theoretic approach to planning in dynamic environments. Technical Report CS-91-06, Department of Computer Science, Brown University, May 1991.

[Brooks, 1986] Rodney Brooks. A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation*, RA-2:14–23, April 1986.

[Cassandra *et al.*, 1994] A.R. Cassandra, L.P. Kaelbling, and M.L. Littman. Acting optimally in partially observable stochastic domains. In *Proceedings of the Twelfth National Conference on Artificial Intelligence (AAAI-94)*, pages 1023–1028. AAAI, AAAI Press/The MIT Press, July 1994.

[Chapman, 1987] D. Chapman. Planning for conjunctive goals. *Artificial Intelligence*, 32(3):333–377, 1987.

[de Souza, 1993] Pedro Sergio de Souza. Asynchronous organizations for multi-algorithm problems. Technical Report EDRC-02-28-93, Carnegie Mellon University, Pittsburgh PA. USA, April 1993.

[Dean and Boddy, 1988] Thomas Dean and Mark Boddy. An analysis of time dependent planning. In *Proceedings of the Seventh National Conference on Artificial Intelligence (AAAI-88)*, pages 49–54. AAAI, 1988.

[Dean *et al.*, 1993] Thomas Dean, Leslie Pack Kaelbling, Jack Kirman, and Ann Nicholson. Planning with deadlines in stochastic domains. In *Proceedings of the Eleventh National Conference on Artificial Intelligence (AAAI-93)*. AAAI, July 1993.

[Doan and Haddawy, 1995] A. Doan and P. Haddawy. Generating macro operators for decision-theoretic planning. In *Working Notes of the AAAI Spring Symposium on Extending Theories of Action*, Stanford, March 1995.

[Drummond and Tate, 1989] Mark Drummond and Austin Tate. Ai planning: A tutorial review. Technical Report AIAI-TR-30, AI Applications Institute, Edinburgh University, January 1989.

[Etzioni, 1989] Oren Etzioni. Tractable decision-analytic control. In *Proceedings of the First International Conference on Principles of Knowledge Representation and Reasoning (KR-89)*, May 1989.

[Feldman and Sproull, 1977] Jerome A. Feldman and Robert F. Sproull. Decision theory and artificial intelligence ii: The hungry monkey. *Cognitive Science*, 1, 1977.

[Fishburn and Vickson, 1978] Peter C. Fishburn and Raymond G. Vickson. *Theoretical Foundataions of Stochastic Dominance*, pages 37–114. Lexington Books, 1978.

[Garey and Johnson, 1979] Michael R. Garey and David S. Johnson. *Computers and Intractability*. W. H. Freeman and Company, 1979.

[Goldman and Boddy, 1994] Robert P. Goldman and Mark S. Boddy. Epsilon-safe planning. In Ramon Lopez de Mantras and David Poole, editors, *Proceedings of the Tenth Conference on Uncertainty in Artificial Intelligence (UAI-94)*, pages 253–261, Seattle, Washington, July 1994.

[Good, 1971] I. J. Good. Twenty seven principles of rationality. In *Proceedings of the Symposium on the FOUNDATIONS of STATISTICAL INFERENCE*. Rene Descartes Foundation, March 1971.

[Goodwin, 1996] Richard Goodwin. Using loops in decision–theoretic refinement planners. In Brian Drabble, editor, *Proceedings of the Third International Conference on Artificial Intelligence Planning Systems (AIPS-96)*, pages 118–124, Edinburgh, Scotland, May 1996. AAAI Press.

[Haddawy and Doan, 1994] P. Haddawy and A. Doan. Abstracting probabilistic actions. In Ramon Lopez de Mantras and David Poole, editors, *Proceedings of the Tenth Conference on Uncertainty in Artificial Intelligence (UAI-94)*, pages 270–277, Seattle, Washington, July 1994.

[Haddawy and Hanks, 1993] Peter Haddawy and Steve Hanks. Utility models for goal-directed decision-theoretic planners. Technical Report 93-06-04, Department of Computer Science and Engineering, University of Washington, June 1993.

[Haddawy *et al.*, 1995] P. Haddawy, C.E. Kahn, Jr, and A.H. Doan. Decision-theoretic refinement planning in medical decision making: Management of acute deep venous thrombosis. *(To appear in Medical Decision Making)*, 1995. (submitted to *Medical Decision Making*).

[Hanks, 1990] Steve Hanks. Practical temporal projection. In *Proceedings of the Eighth National Conference on Artificial Intelligence (AAAI-90)*. AAAI, July 1990.

[Hart *et al.*, 1968] P. E. Hart, N. J. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on SSC*, 4:100–107, 1968.

[Hart *et al.*, 1972] P. E. Hart, N. J. Nilsson, and B. Raphael. Correction to 'a formal basis for the heuristic determination of minimum cost paths'. *SIGART Newsletter*, 37:28–29, 1972.

[Hillner *et al.*, 1992] B. E. Hillner, J. T. Philbrick, and D. M. Becker. Optimal management of suspected lower-extremity deep vein thrombosis: an evaluation with cost assessment of 24 management strategies. *Arch Intern Med*, 152:165–175, 1992.

[Hills and Johnson, 1996] A. Hills and D. B. Johnson. A wireless data network infrastructure at Carnegie Mellon University. *IEEE Personal Communications*, 3(1):56–63, February 1996.

[Insua and French, 1991] David Rios Insua and Simon French. A framework for sensitivity analysis in discrete multi-objective decision making. *European Journal of Operational Research*, 54(2):176–190, Sept. 1991.

[Insua, 1990] David Rios Insua. *Sensitivity Analysis in Multi-objective Decision Making*. Lecture Notes in Economics and Mathematical Systems. Springer-Verlag, 1990.

[Joslin and Pollack, 1994] David Joslin and Martha E. Pollack. Least-cost flaw repair: A plan refinement strategy for partial-order planning. In *Proceedings of the Twelfth National Conference on Artificial Intelligence (AAAI-94)*, pages 1010–1015. AAAI, AAAI Press/The MIT Press, July 1994.

[Knoblock, 1991] Craig A. Knoblock. Search reduction in hierarchical problem solving. In *Proceedings of the Ninth National Conference on Artificial Intelligence (AAAI-91)*, pages 686–691. AAAI, July 1991.

[Koenig and Simmons, 1994] Sven Koenig and Reid G. Simmons. How to make reactive planners risk-sensitive (without altering anything). In Kristian Hammond, editor, *Proceedings of the Second International Conference on Artificial Intelligence Planning Systems (AIPS-94)*, pages 293–298, Chicago, Illinois, June 1994. AAAI Press.

[Koenig *et al.*, 1996] Sven Koenig, Richard Goodwin, and Reid Simmons. Robot navigation with Markov models: A framework for path planning and learning with limited computational resources. In *Proceedings of the International Workshop on Reasoning with Uncertainty in Robotics, Amsterdam, The Netherlands.*, Dec. 1996.

[Kushmerick *et al.*, 1994] Nicholas Kushmerick, Steve Hanks, and Daniel Weld. An algorithm for probabilistic least-commentment planning. In *Proceedings of the Twelfth National Conference on Artificial Intelligence (AAAI-94)*, pages 1073–1078. AAAI, AAAI Press/The MIT Press, July 1994.

[Lawler *et al.*, 1985] E. L. Lawler, J. K. Lenstra, A. H. G. Rinnooy Kan, and D. B. Shimoys. *The Travelling Salesman Problem*. Wiley, New York, NY, 1985.

[Lin and Kernighan, 1973] S. Lin and B.W. Kernighan. An effective heuristic for the travelling salesman problem. *Operations Research*, 21:498–516, 1973.

[Littman *et al.*, 1995] M.L. Littman, A.R. Cassandra, and L.P. Kaelbling. Learning policies for partially observable environments: Scaling up. In *Proceedings of the Twelfth International Conference on Machine Learning (ML-95)*, pages 362–370, 1995.

[Loomes and Sugden, 1982] Graham Loomes and Robert Sugden. Regret theory: an alternative theory of rational choice under uncertainty. *The Economic Journal*, 92:805–824, December 1982.

[Loomes, 1987] Graham Loomes. Some implications of a more general form of regret theory. *Journal of Economic Theory*, 41:270–287, 1987.

[Lovejoy, 1991] W.S. Lovejoy. A survey of algorithmic methods for partially observed Markov decision processes. *Annals of Operations Research*, 28(1):47–65, 1991.

[McAllester and Rosenblitt, 1991] David McAllester and David Rosenblitt. Systematic nonlinear planning. In *Proceedings of the Ninth National Conference on Artificial Intelligence (AAAI-91)*, pages 634–639. AAAI, July 1991.

[McDermott, 1982] D. McDermott. A temporal logic for reasoning about processes and plans. *Cognitive Science*, 6(2):101–155, 1982.

[Nakakuki and Sadeh, 1994] Yichiro Nakakuki and Norman Sadeh. Increasing the efficiency of simulated annealing search by learning to recognize (un)promising runs. In *Proceedings of the Twelfth National Conference on Artificial Intelligence (AAAI-94)*, pages 1316–1322. AAAI, AAAI Press/The MIT Press, July 1994.

[Newell and Simon, 1961] A. Newell and H. A. Simon. Gps, a program that simulates human thought. In H. Billing, editor, *Lernende Automaten*, pages 109–124. Oldenbourg, Munich, 1961. (Reprinted in Feigenbaum, E. and Feldman, J. [eds.], 'Computers and Thought', McGraw-Hill, 1963).

[Newell, 1982] Allan Newell. The knowledge level. *Artificial Intelligence*, 18(1):87–127, 1982.

[Ngo and Haddawy, 1995] Liem Ngo and Peter Haddawy. Representing iterative loops for decision-theoretic planning (preliminary report). In *Working Notes of the AAAI Spring Symposium on Extending Theories of Action*, Stanford, March 1995.

[Nourbakhsh, 1996] Illah Reza Nourbakhsh. *Interleaving Planning and Execution*. PhD thesis, Stanford University, August 1996. Sven has a copy of the thesis.

[Parr and Russell, 1995] R. Parr and S. Russell. Approximating optimal policies for partially observable stochastic domains. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence (IJCAI-95)*, pages 1088–1094, 1995.

[Pemberthy and Weld, 1992] J. S. Pemberthy and D. Weld. Ucpop: A sound, complete, partial order planner for adl. In *Proceedings of the Third International Conference on Principles of Knowledge Representation and Reasoning (KR-92)*, pages 103–114, October 1992.

[Pérez, 1995] Maria Alicia Pérez. *Learning Search Control Knowledge to Improve Plan Quality*. PhD thesis, Carnegie Mellon University, July 1995.

[Raiffa, 1968] Howard Raiffa. *Decision Analysis: Introductory Lectures on Choices under Uncertainty*. Addison-Wesley, Reading Mass., 1968.

[Raiman, 1991] Olivier Raiman. Order of magnitude reasoning. *Artificial Intelligence*, 51:11–38, 1991.

[Russell and Wefald, 1991] Stuart Russell and Eric Wefald. *Do the Right Thing*. MIT Press, 1991.

[Russell *et al.*, 1993] Stuart J. Russell, Devika Subramanian, and Ronald Parr. Provably bounded optimal agents. In *Proceedings of the Thirteenth International Joint Conference on Artificial Intelligence (IJCAI-93)*. IJCAI, 1993.

[Sacerdoti, 1975] E.D. Sacerdoti. The non-linear nature of plans. In *IJCAI-75*, pages 206–214, Tbilisi, Georgia, USSR, 1975. IJCAI. Also Tech. Note 101, SRI.

[Sacerdoti, 1977] Earl Sacerdoti. *A Structure for Plans and Behavior*. American Elsevier, New York, NY, 1977.

[Schoppers, 1987] Marcel Schoppers. Universal plans for reactive robots in unpredictable environments. In *Proceedings of the Tenth International Joint Conference on Artificial Intelligence (IJCAI-87)*, Milan, Italy, August 1987. IJCAI.

[Simmons and Koenig, 1995] R.G. Simmons and S. Koenig. Probabilistic robot navigation in partially observable environments. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence (IJCAI-95)*, pages 1080–1087, 1995.

[Simmons *et al.*, 1997] Reid Simmons, Richard Goodwin, Karen Zita Haigh, Sven Koenig, and Joseph O'Sullivan. A modular architecture for office delivery robots. In *Submitted to the First International Conference on Autonomous Agents*, Feburary 1997.

[Simmons, 1992] Reid Simmons. The roles of associational and causal reasoning in problem solving. *Artificial Intelligence*, 53(2–3):159–208, February 1992.

[Simmons, 1994] R. Simmons. Structured control for autonomous robots. *IEEE Transactions on Robotics and Automation*, 10(1):34–43, 2 1994.

[Simmons, 1996] R. Simmons. The curvature-velocity method for local obstacle avoidance. In *Proceedings of the International Conference on Robotics and Automation (AA-96)*, 1996.

[Simon and Kadane, 1974] Herbert Simon and Joseph Kadane. Optimal problem-solving search: All-or-nothing solutions. Technical Report CMU-CS-74-41, Carnegie Mellon University, Pittsburgh PA. USA, December 1974.

[Simon, 1956] Herbert A. Simon. Rational choice and the structure of the environment. *Psychological Review*, 63(2):129–138, 1956.

[Simon, 1976] Herbert A. Simon. *From Substantive to Procedural Rationality*, pages 129–148. Cambridge University Press, 1976.

[Simon, 1982] Herbert A. Simon. *Theories of Decision-Making in Economics and Behavioral Science*, pages 287–317. The MIT Press, 1982.

[Simon, 1988] Herbert Simon. Problem formulation and alternative generation in the decision making process. Technical Report AIP-43, CMU psychology, 1988.

[Stefik, 1981] Mark Stefik. Planning and meta-planning (molgen: Part2). *Artificial Intelligence*, 16:141–170, 1981.

[Stone *et al.*, 1994] Peter Stone, Manuela Veloso, and Jim Blythe. The need for different domain-independent heuristics. In Kristian Hammond, editor, *Proceedings of the Second International Conference on Artificial Intelligence Planning Systems (AIPS-94)*, pages 164–169, Chicago, Illinois, June 1994. AAAI Press.

[Tate, 1977] A. Tate. Generating project networks. In *IJCAI-77*, pages 888–893, MIT, Cambridge, MA, 1977. IJCAI.

[Weld, 1987] Daniel S. Weld. Comparative analysis. Technical Report AI Memo 951, MIT, November 1987.

[Weld, 1991] Daniel S. Weld. Reasoning about model accuracy. Technical Report 91-05-02, University of Washington, May 1991.

[Wellman and Doyle, 1991] Michael P. Wellman and Jon Doyle. Preferential semantics for goals. In *Proceedings of the Ninth National Conference on Artificial Intelligence (AAAI-91)*, pages 698–703. AAAI, July 1991.

[Wellman *et al.*, 1995] Michael P. Wellman, Matthew Ford, and Kenneth Larson. Path planning under time-dependent uncertainty. In Philippe Besnard and Steve Hanks, editors, *Proceedings of the Eleventh Conference on Uncertainty in Artificial Intelligence (UAI-95)*, Montreal Canada, August 1995.

[Wellman, 1988] Michael P. Wellman. Formulation of tradeoffs in planning under uncertainty. Technical Report MIT/LCS/TR-427, MIT, August 1988.

[Whitmore and Findlay, 1978] G. A. Whitmore and M. C. Findlay. *Stochastic Dominance; An Approach to Decision Making Under Risk.* Lexington Books, D.C. Heath and Co. Lexington Massachusetts, 1978.

[Williamson and Hanks, 1994] Mike Williamson and Steve Hanks. Optimal planning with a goal-directed utility model. In Kristian Hammond, editor, *Proceedings of the Second International Conference on Artificial Intelligence Planning Systems (AIPS-94)*, pages 176–181, Chicago, Illinois, June 1994. AAAI Press.

[Williamson and Hanks, 1996] Mike Williamson and Steve Hanks. Flaw selection strategies for value directed planning. In Brian Drabble, editor, *Proceedings of the Third International Conference on Artificial Intelligence Planning Systems (AIPS-96)*, pages 237–244, Edinburgh, Scotland, May 1996. AAAI Press.

[Zilberstein and Russell, 1992] Zilberstein and Russell. Efficient resource-bounded reasoning in at–ralph. In James Hendler, editor, *Proceedings of the First International Conference on Artificial Intelligence Planning Systems (AIPS-92)*, College Park, Maryland, June 1992. Morgan Kaufmann Publishers, Inc.