Learning Bounded Optimal Behavior Using Markov Decision Processes

by

Hon Fai Vuong

Submitted to the Department of Aeronautics and Astronautics
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy in Aeronautics and Astronautics

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2007

[illegible]

Author .... [illegible]
Department of Aeronautics and Astronautics
August 24, 2007

Certified by .... [illegible]
Nicholas Roy, Ph.D
Assistant Professor of Aeronautics and Astronautics
Thesis Supervisor

Certified by .... [illegible]
Milton B. Adams, Ph.D
Director of Strategic Technology Planning
The Charles Stark Draper Laboratory, Inc.
Thesis Supervisor

Certified by .... [illegible]
Mary L. Cummings, Ph.D
Assistant Professor of Aeronautics and Astronautics
Committee Member

Accepted by .... [illegible]
David L. Darmofal
Chairman, Department Committee on Graduate Students

Learning Bounded Optimal Behavior Using Markov Decision Processes

by

Hon Fai Vuong

Submitted to the Department of Aeronautics and Astronautics
on August 24, 2007, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy in Aeronautics and Astronautics

Abstract

Creating agents that behave rationally in the real-world is one goal of Artificial Intelligence. A rational agent is one that takes, at each point in time, the optimal action such that its expected utility is maximized. However, to determine the optimal action the agent may need to engage in lengthy deliberations or computations. The effect of computation is generally not explicitly considered when performing deliberations. In reality, spending too much time in deliberation may yield high quality plans that do not satisfy the natural timing constraints of a problem, making them effectively useless. Enforcing shortened deliberation times may yield timely plans, but these may be of diminished utility. These two cases suggest the possibility of optimizing an agent's deliberation process. This thesis proposes a framework for generating metalevel controllers that select computational actions to perform by optimally trading off their benefit against their costs.

The metalevel optimization problem is posed within a Markov Decision Process framework and is solved off-line to determine a policy for carrying out computations. Once the optimal policy is determined, it serves efficiently as an online metalevel controller that selects computational actions conditioned upon the current state of computation. Solving for the exact policy of the metalevel optimization problem becomes computationally intractable with problem size. A learning approach that takes advantage of the problem structure is proposed to generate approximate policies that are shown to perform relatively well in comparison to optimal policies.

Metalevel policies are generated for two types of problem scenarios, distinguished by the representation of the cost of computation. In the first case, the cost of computation is explicitly defined as part of the problem description. In the second case, it is implicit in the timing constraints of problem. Results are presented to validate the beneficial effects of metalevel planning over traditional methods when the cost of computation has a significant effect on the utility of a plan.

Thesis Supervisor: Nicholas Roy, Ph.D
Title: Assistant Professor of Aeronautics and Astronautics

Thesis Supervisor: Milton B. Adams, Ph.D
Title: Director of Strategic Technology Planning, Draper Laboratory

Acknowledgments

"No man is an Island,...".

-John Donne

The thesis would not have been completed without the help of a multitude of people to whom I am deeply indebted.

I would like thank my thesis advisor, Nick Roy, whose endless enthusiasm and thoughtful hands-on approach to research was a unique experience to me even at an institution as unique as MIT. Nick was great source of inspiration and was always able to see a silver lining, even at times when I thought things were not going well. I truly enjoyed working with him.

I would also like to thank Milt Adams, my Draper advisor. Milt had the wisdom and foresight to ask many tough questions who true significance were only apparent with time. These questions helped me clarify, in my own mind, my research approach, and, more importantly, how to communicate it to others. He was also invaluable for his efforts in editing many buggy thesis drafts. My success is due, in no small part, to both Nick and Milt.

My appreciation goes out to the rest of my thesis committee: Missy Cummings, Emilio Frazzoli, and John Deyst. Both Emilio and John spent much time and effort reading over my thesis draft and made many useful suggestions. Missy especially offered many excellent comments, under time pressure, which helped me present the material in my thesis in a much clearer fashion. Also, I would like to thank Missy and Sylvain Bruni for suggesting the vehicle routing problem as a real-world example for metalevel control.

I am grateful the Robust Robotic's Group (unofficially Nicks-group): Emma Brunskill, Finale Doshi, Jared Glover, Valerie Gordeski, RJ He, Tom Kollar, Pete Lommel, Bhaskara Marthi, Sooho Park, Sam Prentice and Matt Walter for listening to me ramble or practice rambling about metaplanning and anytime algorithms over the years. I would especially like to thank Emma and Finale for being brave enough to sit through two of my practice thesis defense talks, providing confidence and offering valuable suggestions that led to a successful defense.

In addition, I would like to thank Eric Feron for his help early on in this thesis and for introducing me to Nick. The department administrators, Marie Stuppard and Barbara

Lechner both deserve my appreciation for all the help they have provided over the years. The only thing I feel bad about is that the only time I would ever see them was when I got into trouble.

I would like to sincerely thank Draper Laboratory for the opportunity and resources provided to me for both my education and research. I have interacted many wonderful people at Draper over the years. I would first like to thank Mark Abramson, Francis Carr, Sameera Ponda, Paul Rosendall, Chris Sanders, Benjamin Werner and Len Wholey for providing an opportunity for me to practice my defense talk. Len was very helpful in catching some last minute errors which I would not have found on my own. I appreciate the Draper staff: Jeff Cipolloni, Bill Hall, Chris Dever, Alex Kahn, Steve Kolitz, John Turkovich and Lee Yang for having supported me in variety of ways. I've been here for such a long time that many people that were around at the beginning of this journey have moved on to bigger and better things. There are far too many to mention, but, most notably, I would like to thank Darryl Ahner, Mathias Craig, Ramsey Key, Spencer Firestone and Eric Zarybnisky, for letting me bounce around my initial research ideas and letting me know how crazy I sounded.

I am indebted to Loren Werner for his emergency editing of my thesis while on vacation and also for helping me prepare for my defense. I appreciate my roommates Anna and Mandy Ing for volunteering to proofread my thesis and listening to an early version of my practice defense. I am thankful for the help Jillian Redfern provided during the formative years of this thesis from proofreading papers to listening to practice talks over the phone. I would also like to thank the people who had no direct contact with my thesis but helped to keep my morale up during many trying times: Guy Barth, Jack Lloyd, Laura Gurevich, Emerson Quan, and Seaver Wang. To all my other friends I say thanks as well.

Finally, I would like to thank my family for believing in me and for providing me with the love and support I needed to get the job done. Sophia, thanks putting up with a grumpy brother and for making an effort to read this. Mom and Dad, thanks for always being there for me. This one's for you!

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Agents interacting in the real world are subject to limited resources. This thesis focuses on the problem of limited *computational* resources which agents must use to generate a plan and the impact of those limited computational resources on the time it takes to generate a plan. Ideally, with unlimited computational resources, an agent would be able to instantaneously generate the best plan at each point in time based on the most up to date state information available. The main effect of limited computation is that it can make generating an *optimal* plan a suboptimal course of action. In particular, consider, for example, a planning agent that must generate and execute a plan under time-critical situations. Spending the required time to compute the absolute best plan may ultimately result in reduced performance when it comes time to execute the plan. One reason for reduced performance may be that opportunities that were available at the start of the planning process have expired by the time the optimal plan is available for execution. To deal with this "cost of time" problem during plan computation, one might consider limiting the amount of computational effort to expend prior to execution. This *ad hoc* approach enables plans to be executed more rapidly, but with a lack of understanding of how the quality of such plans may suffer as a consequence. Inherent in these two extremes, of expending much effort in computing the best plan or expending little effort to compute an inferior plan, is the problem of optimizing the computations taken by the planning agent to generate the best plan possible within the computational capabilities of the agent. These plans are not optimal in the sense of being equivalent to the absolute best plans that would be generated under unlimited computational resources, but instead are optimal because they are the best that can be achieved considering the costs involved in computing them.

The goal of this thesis is to develop a general framework for creating a mechanism to optimally control the course of computation under limited rationality [43] or limited computational resources. This mechanism, embodied as what is referred to here as the *metalevel controller*, is used to select computational actions to perform. Although the optimal metalevel controller for any particular problem domain is problem-specific, the framework for building them is generic. The metalevel planning problem of generating a metalevel controller is first formulated as a sequential decision making problem and modeled as a Markov decision process (MDP). But because of the *curse of dimensionality*, the model grows large very quickly making the generation of exact solutions computationally intractable. A study of the optimal solutions generated exactly for small problems by the MDP approach leads to the development of a heuristic learning algorithm, which takes advantage of problem structure to reduce the search space of much larger problems and serves as the main framework for generating metalevel controllers. The result of applying this framework for a specific planning domain yields an optimal metalevel controller that allows the agent to achieve *bounded optimality* [40]. Simply stated, a bounded optimal agent performs as well as possible in a given problem domain with its computational resources.

1.1 The Basic Problem

This section describes the basic metalevel planning problem that is addressed in this thesis.

1.1.1 Example Problem

A representative example that motivates the need for an optimal metalevel controller is time-critical targeting, wherein missions must be generated for aircraft to strike possibly mobile targets distributed over some geographic region. Each struck target yields some amount of value toward achieving battlefield objectives. Assume that each target is known to have a window of opportunity specifying the time interval over which it can be visited. Aircraft arriving prior to the time window must wait for the target to appear, and arriving past the time window results in zero value. The objective of the problem is to determine missions for a set of aircraft resources to execute such that the total target value collected is maximized. Under these circumstances it is often undesirable to compute the absolute

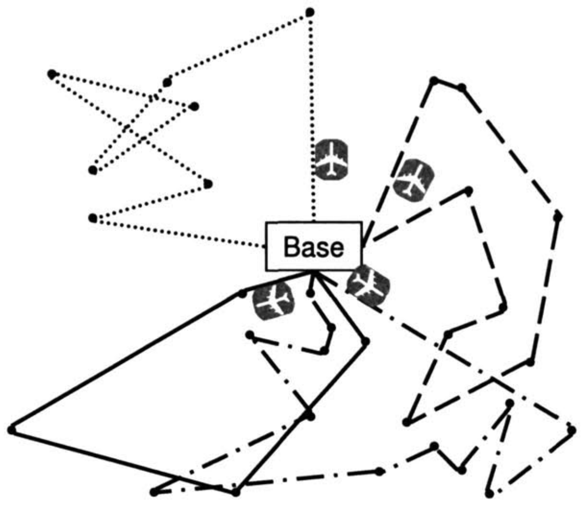

Figure 1-1: A time-critical targeting problem where missions must be generated for aircraft to prosecute targets that appear within a limited time window. In such a situation, it is imperative to account for how the time taken to compute a plan will affect the value of the final mission. Each of the four missions is distinguished by a different line style, where the targets are shown as dots.

best solution. Due to the sheer amount of time required to do so, the resulting set of missions may not be successfully executed because targets appearing in these missions may have expired. On the other hand, using rules of thumb to generate missions may yield fast response times, but may not make full use of the planning agent's computational potential. Because the agent is limited in its computation power, computations require time to be completed. It is therefore important to make optimal use of the computational actions the agent takes when there is a cost, either implicit of explicit, of time. Intuitively, this is accomplished by weighing the utility gained from computational actions against the cost of taking them.

Figure 1-1 depicts the time-critical targeting problem for four aircraft stationed at a single base location. In the figure, a set of missions has been determined, where each aircraft is shown to be carrying out its assigned mission, indicated by the line segments. Each mission starts at the base and includes the set of targets to be visited by each aircraft and the order in which to visit them, indicated in the figure by dots. A mission concludes when the aircraft returns to the base. Line segments for individual missions in the figure are shown to crisscross to notionally indicate that windows of opportunity (not shown in figure) for targets may induce additional complexity in the problem.

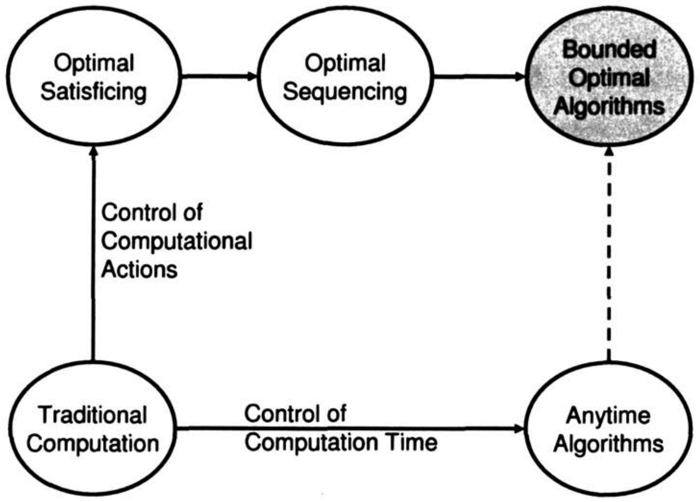
Optimal Satisficing
Optimal Sequencing
Bounded Optimal Algorithms
Control of Computational Actions
Traditional Computation
Control of Computation Time
Anytime Algorithms

Figure 1-2: A variety of approaches for addressing the problem of limited rationality.

The time-critical targeting problem shares many properties with the problem domains considered in this thesis and will be used in this chapter to illustrate some salient points of the problems addressed and the approaches taken to address them.

1.2 General Approaches to Metalevel Planning

This section serves to establish how the work in this thesis fits into the field of established research in limited rationality or limited computational resources, to describe a representative set of metalevel problem formulations, and to discuss how they relate to one another and to the work in found in this thesis.

Figure 1-2 presents various algorithmic approaches that have been developed to deal with systems under limited rationality. There are two main axes of division which distinguish these approaches. These distinctions are based upon the type of control actions available at the metalevel and are labeled "Control of Computation Time" and "Control of Computational Actions" in Figure 1-2.

The baseline case of "Traditional Computation" is shown at the bottom left of the figure. The Traditional Computation category does not represent an approach to limited rationality, but is included to capture the traditional notion of rational agents. Agents falling in this category do eventually calculate the optimal solution to a problem. In a timeless world this would be ideal, however in the real world the optimal solution may turn out to have no operational significance during execution. Since they do not explicitly consider the effects of deliberation (e.g., the time and associated costs of performing calculations), Traditional Computation is inappropriate for acting in time-critical situations. For example, solving the time-critical targeting problem to optimality is combinatorially difficult and unlikely to yield useful real-world plans as discussed earlier.

Moving along the x-axis of Figure 1-2 leads to the category labeled "Anytime Algorithms" ([24], [51]).

> **DEFINITION:** anytime algorithm: an algorithm that generates a series of successive solutions of increasing quality as a function of planning time, so that a reasonable decision is ready whenever it is interrupted.

The sole job of the metalevel controller for anytime algorithms is to decide how long to run the anytime algorithm. There is typically an optimal stopping time where more planning leads to a worse overall performance because the cost of additional planning time outweighs it benefits. The use of anytime algorithms for metalevel planning assumes that the plan performance is monotonically non-decreasing with planning time (i.e. the solutions produced by the anytime algorithm do not get worse with time). This property can be satisfied in most cases by storing the current best plan, only replacing it when a better one is found. The problem in combining generic anytime algorithms with metaplanning problems with hard temporal constraints is that the guarantee of monotonicity may be difficult to maintain. When temporal constraints are violated by the planning process itself, the quality of the current best plan may decrease.

Along the y-axis of Figure 1-2, the control parameter is "what to compute" instead of "how long to compute". This offers an additional degree of control in comparison to anytime algorithms. Rather than treating the algorithm as a black box, the metalevel controller is allowed to dictate which computations to make. The oval labeled "Optimal Satisficing" is the most basic category for the control of computational actions.

> **DEFINITION:** optimal satisficing problem: a problem whose objective is to minimize the cost of generating a feasible solution.

Optimal satisficing may sound like an oxymoron, but it is not since the optimization is solely concerned with minimizing the computational effort involved in finding a satisficing solution [44]. For the time-critical targeting problem, optimal satisficing may find a feasible solution quickly, but its quality might be very low.

Moving along the x-axis leads to approaches that attempt to generate individual computational actions to "optimize" rather than satisfice ([17], [18], [42]). For optimal satisficing, all feasible solutions are assumed to have equal utility, and the only concern is to minimize the effort of finding one of them. Recognizing that not all solutions are of the same quality or utility, the approaches labeled "Optimal Sequencing" balances the utility of computation against its cost. The objective function for these approaches accounts for both the utility of the base-level solution and the cost it took to compute it.

There are two sub-categories that fit into this group. The first sub-category ([17], [18]) consists of approaches, that seek to optimally sequence a fixed set of *methods*, or complete decision procedures, each with a probability of success for solving the problem, an expected utility and an expected running time. For the time-critical targeting problem, this approach might correspond to having various methods for solving the problem. For instance, a heuristic search method may be the fastest algorithm but may seldom yield solutions of high quality. On the other hand, exhaustive enumeration is guaranteed to yield high quality solutions, but may take a long time to terminate. The metalevel planning problem for this case is to optimally sequence the order of running each of these methods that is appropriate for the degree of time-criticality.

DEFINITION: complete decision procedure: a sequence of computations that completely solves the base-level planning problem.

The second sub-category consists of heuristic search control approaches [42]. The metalevel controller for these approaches is to control the search effort. This is accomplished by using heuristics both to determine the planning time and the computational actions to take. Since they are, by definition heuristics, these methods are unable to guarantee the optimal use of computational resources. They also rely on some amount of metalevel computation during run-time in order to estimate the utility of additional computation, continuing when it is non-negative and stopping otherwise.

The work in this thesis falls to the right of these approaches in the category of "Bounded Optimal Algorithms" and focuses on generating "algorithms" or programs to optimally govern the computational behavior of an agent. Like the optimal sequencing approaches, these algorithms also control discrete computational actions. Rather than having to perform a heuristic search for each problem instance encountered, as above, a closed-loop controller is trained to determine the computational actions to take over all problem instances, given feedback from prior computations.

The work of this thesis aims to generate metalevel controllers that allow the agent to behave optimally given its computational resource constraints. Optimal agent behavior consists of achieving the best average-case performance among all possible agent designs that have the same computational resource constraints acting over the same set of problem distributions. That is, there may be other metalevel control designs that perform better on particular problem instances, but as a whole perform worse on average than the optimal design.

For the time-critical targeting problem, the optimal metalevel controller would select the highest utility sub-problem (accounting for the expected utility of future computations) to solve, solve it, and, based on the outcome of the solution, appropriately select the next sub-problem to solve. At some point, the utility of solving sub-problems will be outweighed by its cost, in which case, the optimal metalevel controller will signal that the current best plan be executed.

1.3 Thesis Statement

The general problem of solving for exact bounded optimal controllers is, in general, computationally intractable due to the combinatorial explosion from the exponential number of possible computation sequences that exist. This thesis proposes that many problem domains exhibit some form of problem structure that can be exploited, so that a metalevel policy can be learned rather than solved exactly. It will be shown that exploiting problem structure to learn the metalevel policy, rather than solving for it exactly, can prune a vast amount the search space. The result is that, by solving a significantly smaller optimization problem, performance gains similar to that generated by the exact metalevel policy can be achieved.

1.3.1 Thesis Contributions

The first contribution of this thesis is a specific model of the metalevel planning problem as an MDP for generating bounded optimal controllers. The resulting metalevel controllers use feedback from prior computations to select among *atomic computational actions.* This form of metalevel control is a generalization of previous work, which assumes that there is an *a priori* set of complete decision procedures from which to choose. Using the framework to create a metalevel controller is equivalent to constructing the best possible complete decision procedure. This obviates the need to have a predefined, and potentially limiting, set of decision procedures. One limitation of this framework is that it relies heavily on the assumption of the availability of a good *a priori* problem decomposition. That is, the policies that are generated are the best that can be constructed using the given set of computational actions (sub-problems). The issue of problem decomposition is an important one, but is not specifically addressed in this thesis and assumed to be provided externally. Another drawback of using this exact MDP framework is that it does not scale well to larger problems.

The specific MDP formulation of the metalevel planning problem, lends itself to a learning approach for generating metalevel control policies. The second contribution is a heuristic learning algorithm called Decision Tree Metaplanning (DTMP) for generating metalevel controllers based on decision tree learning and approximate MDP approaches. DTMP reduces the size of the relevant metalevel state space by learning to identify the important aspects of the problem. This thereby reduces the size of the optimization problem that must be solved to generate metalevel controllers for approximating bounded optimality. This heuristic algorithm addresses the scaling problem and allows for metalevel controllers to be generated for problems that are much larger than an exact approach can realistically handle. It sacrifices optimality, but still yields good performance.

Experimental results are generated for both the exact and heuristic approaches to demonstrate that the metalevel controllers generated are better than that of simple open-loop complete decision procedures as well as that of metalevel planning using anytime algorithms. In particular, it is also shown, that DTMP generated metalevel controllers perform comparably with the exact MDP policies.

Although a majority of this thesis is devoted to generating metalevel controllers for the case where the costs of computation and value of sub-problems are explicitly known and stationary. The final result of this thesis is to show that the DTMP algorithm, with some modifications, also yields good results in the more difficult case where the costs and sub-problem rewards are non-stationary. It will be shown that the DTMP algorithm can be adapted to handle non-stationary costs due to hard temporal constraints with little additional computational overhead.

1.4 Thesis Approach

The approach taken in this thesis, for generating optimal metalevel controllers, is to solve a metalevel planning problem, a large-scale optimization problem, off-line. This metalevel planning problem is modeled as a sequential decision making problem [32], where the decision being made at each point in time is which sub-problem to solve and whether to execute the current best plan. The sequential decision making problem can be formulated as a Markov Decision Process (MDP) [32] and solved, to produce a *policy.* The policy serves directly as the agent's metalevel controller. A policy is also known as a *universal plan* [32], which conditionally dictates the next action to take based on the current state. In the metalevel planning problem, the state represents the current condition of computation. The policy (metalevel controller) dictates the next sub-problem to solve given the current state of computation in the form of a learned lookup table, allowing for an efficient online implementation of the metalevel controller.

The advantage of this approach is that a metalevel planning problem does not need to be solved during run-time when the agent is acting in the environment. Computing the optimal metalevel controller is a difficult optimization problem that, in general, surpasses the difficulty of solving the base-level planning problem. By solving for a policy off-line, the agent need not perform additional computations online with the exception of negligible table lookups to determine the next sub-problem to solve. Another advantage of this approach is the availability of well-established and efficient methods for generating optimal policies. Although the planning problems considered in this thesis have a finite number of metalevel policies, brute-force enumeration is computationally intractable. Last of all, this approach allows the generation of provably optimal metalevel policies that guarantee that the agent is making the best use of its computational resources.

The MDP approach taken in this thesis for generating bounded optimal algorithms achieves what heuristic search control approaches attempt to achieve, which is to optimally select computational actions. This approach can also be modified to accommodate optimal satisficing as well. The advantage that it has over selecting from a fixed set of complete decision procedures is that it actually constructs the optimal complete decision procedure from atomic computational actions, rather than having to rely on the designer to provide a good set of starting decision procedures. This is similar to the advantage of having raw materials (atomic computational actions) to construct a building rather than prefabricated parts (complete decision procedures). Using raw materials allows for additional flexibility since they can be shaped and combined in a variety of ways to appropriately accommodate a wide range of designs that may not otherwise be crafted using prefabricated parts.

The same can be said when this approach is thought of in the context of anytime algorithms. While there may be many anytime algorithms appropriate for a given problem, the metalevel planning problem to be solved assumes that the anytime algorithm is given *a priori*. The metalevel controllers generated in this thesis do not possess the anytime property, but can be thought of as selecting the best possible anytime algorithm for the job along with an appropriate optimal stopping time. Although not explored in this thesis, there is the possibility to modify the framework to produce anytime metalevel controllers.

One of the main limitations faced by this approach is that MDPs suffer from the *curse of dimensionality* [5], which is the problem of exponential state space explosion. This prevents the exact approach, as described above, from being scaled to larger more realistic problems. Careful examination of the optimal policies generated by the exact approach leads to the development of a heuristic learning approach based on decision trees. The central idea behind the heuristic approach is that the exact MDP model can be compactly represented by inducing a conditional ordering constraint on solving sub-problems. By doing so, the metalevel planning problem becomes much simpler to solve for two reasons. The first reason is that the state space of the metalevel MDP is reduced by eliminating many of the possible orderings over sequences of sub-problems. The second reason is that the action space of the MDP, which originally consisted of selecting among all sub-problems, is reduced to two actions, either to plan an additional sub-problem in the order suggested by the induced ordering or to stop and execute the current best plan. It will be shown that this conditional ordering can be represented as a tree and that the trees are in fact isomorphic to

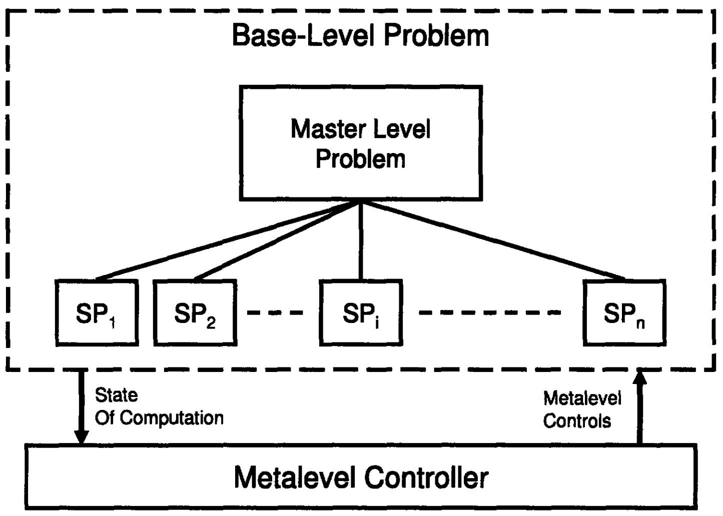

Figure 1-3: Closed-loop metalevel control of a decomposed base-level planning problem.

metalevel policies. In many instances, the performance of the metalevel policies generated heuristically are comparable to solving the original metalevel MDP. By generating these policy trees directly, metalevel planning policies can be generated efficiently to approximate bounded optimality.

1.4.1 Solution Architecture

This thesis aims to solve the problem of generating metalevel controllers to optimally control the computational actions of an agent. In order to formally specify the solution, it is necessary to identify: (1) what constitutes a computational action, and (2) the measures by which to determine they are being controlled optimally. The former question is answered in this subsection, while the latter is addressed superficially here, but in more depth in Chapters 2 and 3.

An agent's computational capability is measured in this thesis by the speed at which it can carry out its computational actions. An important factor to consider in generating metalevel controllers is how computational actions are defined, since they constitute the set of possible control actions. In this thesis, the computational actions considered for control do not extend down to the level of detail of individual cpu instructions, but rather at a higher level of abstraction. In fact, the metalevel controllers are designed here with a specific class of planning problems in mind. That is the class of problems that can be hierarchically decomposed into a two level hierarchy. This class of problems naturally yields a set of computational actions for metalevel control, because solving sub-problems at the lower level can be interpreted as the set of possible computational actions that can be taken. Figure 1-3 illustrates the general properties of the types of planning problems considered in this thesis.

As stated above, foremost of these properties is that the planning problem be hierarchically decomposable into a two level architecture consisting of a master level and a sub-problem level. Hierarchical decomposition is a typical method for addressing problem complexity [33]. The sub-problem level consists of a (possibly large) set of small planning problems representing aspects of the original base-level problem[1]. Solving these smaller sub-problems yields *plan fragments*. These plan fragments are optimally combined by the master level to produce a complete solution to the original base-level planning problem.

Figure 1-3 shows a hierarchical decomposition of a base-level planning problem into master and sub-problem levels. The quality of the final solution produced by the master level depends entirely on the available pool of plan fragments generated by the sub-problem level. Ideally, when every sub-problem is solved optimally, the master level has access to the full pool of plan fragments and can combine them to produce the same optimal solution had the problem not been decomposed.

For instance, the base-level planning problem in the case of the time-critical targeting problem of Figure 1-1 is to generate a plan to maximize the total target value achieved over all executed missions. This problem might be hierarchically decomposed into a master and sub-problem level by defining a sub-problem to be an individual aircraft mission. An individual mission might be defined as determining a plan for sending a particular aircraft after a particular subset of targets. The set of all missions is the set of all possible combinations that can be generated with a given set of targets and aircraft resources. After all possible missions have been generated, the master level performs the role of a mission selector and selects, among all missions, the best combination to produce the complete final plan for execution. Solving a single sub-problem may take a small amount of computation time. However, assuming that sub-problems are solved sequentially (no parallel computation), solving every single one may result in a significant amount of computation time and

[1]The base-level planning problem is the actual real-world problem to be solved and is distinguished from the metalevel planning problem to be discussed later.

causes delays in plan execution. Indeed, in problems with hard temporal constraints, it becomes quite impractical to solve all sub-problems. That is, it can become infeasible to meet some time constraints if too much time is spent computing a solution. Instead, the agent must be smart about which sub-problems it chooses to solve. The role of selecting which sub-problems to solve is filled by the metalevel controller.

> **DEFINITION:** computational action: a single computational action in this thesis is defined as solving a single sub-problem within a problem decomposition.

At this point it should be clear that problems that are hierarchically decomposable motivates the definition of computational actions as the solution of sub-problems. The control of computation for the class of problems considered is equivalent to the control of which sub-problems to solve. The metalevel controller serves both to direct computational actions (i.e., determine which sub-problems to solve), and eventually to determine when to act (i.e., when to execute the current best plan). The metalevel controller is shown in Figure 1-3 and interacts with base-level problem solving in the form of a feedback loop. The metalevel controller receives, as input, the current state of computation (e.g., the results of sub-problems solved thus far), and in turn produces a metalevel control command that either dictates that another sub-problem be solved (as well as which one), or that the current best plan be executed.

1.4.2 Learning the Metalevel Controller

Metalevel controllers in this thesis are generated as a result of learning. In particular, metalevel controllers learn to optimize their control schemes given information about the expected set of problems instances that must be faced by the agent, as well as the agent's computational ability. Learning occurs through observing the interactions between the agent's master and sub-problem level and the utility of the plans that result from these interactions. What must be learned from these observations is the distribution of run-times and the distribution of outcomes for solving individual sub-problems. In addition, the way the master level selects from the set of sub-problem outcomes to generate plans must also be learned. Combining this knowledge together with observing the utilities of the resulting plans allows the metalevel controller to learn to sequentially select the most appropriate sub-problem to solve. In addition, learning may result in discovering that sub-problem outcomes might be coupled in some fashion, where the solution to one may yield information as to whether another should be computed. The advantage of the closed-loop nature of the metalevel controller allows for it to condition its next computational action on the most recent sub-problem outcomes, yielding a tightly controlled system.

As an example of the learning involved in metalevel control, consider once again the time-critical targeting problem of Figure 1-1. The metalevel controller is trained through observing the solution of many problem instances. It must pay attention to the computation times (i.e., the time needed to generate missions), which vary across problem instances. It must also learn how the outcomes that result from generating individual missions vary depending on problem instance. In addition, the way in which missions can interact with one another must also be learned. For instance, successfully generating a high quality mission that uses many aircraft resources may obviate the need to generate other missions that use the same set of resources but are known to be of lower quality. When this situation is encountered in real-time, a metalevel controller which has learned this concept can dispense with computing those lower quality missions. The learning algorithms in this thesis take into account each of these elements to create high quality metalevel controllers.

The goal of learning is to generate a metalevel controller which optimizes an agent's average performance over the distribution of problem instances it is expected to face within a particular problem domain. An agent is said to be bounded optimal when it acts in a way that achieves the highest expected utility over a set of problem instances when compared against all other agents with the same set of computational restrictions.

The ideal class of applications for the learned metalevel controllers developed in this thesis would be in episodic planning problems, where variations in the problem instances across planning episodes prevent the use of pre-planned sub-problems. Indeed, if there were no variation in the planning problem, the agent need only solve the problem once and use the same solution forever. Instead, it is assumed that the agent must face similar situations on a periodic basis where there is enough variability so that sub-problem solutions cannot be exhaustively stored. In these situations, sub-problems must be solved at run-time. The time-critical targeting problem satisfies these conditions, since missions might need to be planned daily or hourly to prosecute targets that may be randomly dispersed over the same geographical region. Each of these problem variations constitutes a problem instance and will have slight variations that will require missions to be solved and combined in real-time.

1.5 Thesis Outline

This section presents a layout for the remainder of this thesis. Chapter 2 presents related work in the literature of bounded optimality. Chapter 3 introduces an MDP formulation of the metalevel planning problem. Some of the issues encountered in the exact MDP implementations of metalevel planning problems are also presented, along with a discussion of ways to alleviate them. Chapter 4 describes the Decision Tree Metaplanning (DTMP) approximation scheme which involves the use of decision trees to solve the metaplanning problem in lieu of the exact MDP approach. The theoretical properties of this approach are analyzed. Chapter 5 presents and analyzes the experiments for both the exact MDP formulation and DTMP for a variety of problem domains. Chapter 6 discusses how DTMP might be adapted to deal with time-critical targeting problems with hard temporal constraints, given by time windows on the targets, which differs significantly from the other problems discussed in this thesis. Experimental results are also presented for this problem. Chapter 7 concludes this thesis and provides a discussion for the possible extensions of this work towards a more complete framework for bounded optimality.

Chapter 2

Related Work

The work in this thesis draws from many interrelated branches of research in the Artificial Intelligence (AI) community. This chapter provides the context necessary to understand the significance of the problem formulated in this thesis, as well as to highlight the main differences between the approaches taken here and those of previously established work in the field of metalevel planning. Section 2.1 motivates the need for an alternative way for judging rational behavior under the conditions of limited computational resources and introduces the notion of Bounded Optimality. The exact metalevel controllers developed in this thesis are generated explicitly to satisfy the requirements of Bounded Optimal agent behavior. The key areas of influence on this thesis, the value of information and metareasoning, are introduced in Sections 2.2 and 2.3 to provide necessary background information. Section 2.4 discusses how the metalevel controllers generated by the framework proposed in this thesis compare to the previous work in metalevel planning.

2.1 Categories of Rationality

Perfect rationality as defined by decision theory, developed by von Neumann and Morgenstern [50], has been recognized as a reasonable goal of agent design. Decision theory combines probability theory and the axioms of utility to help people take actions under uncertainty. According to decision theory, perfectly rational behavior is defined as acting in such a way as to maximize one's own expected utility.

DEFINITION: utility function: a mapping of a state to a real number, which represents a measure of usefulness for being in the state.

For instance, taking an action in gambling, such as placing a particular bet, may yield uncertain results. The uncertainties are represented by a probability distribution over possible state outcomes. The possible outcomes in this case are that the bet is won or it is lost. Each of these states will have a corresponding utility value. In this case, the utility may simply be $+X/-X$ dollars for having won/lost. The expected utility for the bet is the average of the utility values over all outcomes. In this case, the expected utility is given by

$$E[U_b] = p_W X - p_L X,$$

where $E[U_b]$ is the expected utility of the bet, b, and p_W and p_L are the probabilities of winning and losing. Supposing that there are several bets to choose among, a perfectly rational agent will always choose the one which maximizes expected utility. The maximum expected utility, MEU, over the set of all bets, $\mathcal{B}$, is given by

$$MEU = \max_{b \in \mathcal{B}} E[U_b].$$

However, as shown in the literature, perfect rationality is an unreasonable benchmark for real-world agent design when the decision maker is subject to limitations on computation ([15], [24], [42], [43], [51]). This is because decisions are ultimately determined through performing computations. Computations take time to complete. Under real-world conditions, time does not stand still, resulting in natural constraints on the agent. These, in turn, may force it to curtail its computations in order to take action in a timely manner, thereby preventing the agent from computing and taking the same action under the conditions of perfect rationality. Even in this simple betting example, it is necessary to spend some amount of time in deliberation to determine which bet has the maximum expected utility.

In the real-world, there may be time pressure for placing bets, preventing the agent from having enough time to evaluate each one before selecting one. Under these circumstances, expecting the agent to achieve perfect rationality (i.e., always selecting the bet with highest expected utility) is unrealistic. If perfect rationality is an inappropriate measure for describing rational decision making under limited computational resources, what alternatives are available? Russell and Subramanian [40] present a categorization of the various forms of rational behavior and conclude, as does this thesis, that Bounded Optimality is the most appealing alternative. Their categories of rationality include:

- **Perfect rationality** refers to the classical notion of rationality in economics and philosophy [50], where the action with maximum expected utility is always taken. The mechanism (computations) by which the best action is determined is not considered. Russell and Subramanian argue that this is not a realistic goal in the design of situated agents since acting optimally requires computation, which takes time, ultimately affecting the utility of the action. Perfect rationality, as defined, requires instantaneous decision making so that the best action is always available to the agent.

- **Calculative rationality** refers to the ability of an agent to "eventually" compute the optimal action to what would have been the perfectly rational choice at the beginning of its deliberation. This type of rationality explicitly relaxes the effect of time on the utility of a decision. Consider the domain of tournament chess where players have a limited time to make a move. A chess program that is able to eventually evaluate and make the optimal move exhibits calculative rationality since it is able determine the best action for the initial situation. However, in order to do so, it may take a very long time. The effect is that the calculated "best action", is no longer best when the move is finally determined due the passage of time (e.g., the per move time limit is violated). A large number of traditional algorithms, aimed at generating optimal solutions regardless of the amount of time it takes, fall in this category. Again, this definition is generally inappropriate for systems with limited computational resources that must act in real-time situations that demand timely responses.

- **Metalevel rationality** is a direct response to the problems of calculative rationality. Since the best decision is a result of performing computation, metalevel rationality refers to the ability of the agent to perform "exactly" the "right" computations to arrive at the best solution, while accounting for the cost of taking those computations. Optimality under metalevel rationality is a function of both the utility of the final decision and the cost of performing the computational actions to generate the final decision. It has been argued that full metalevel rationality is difficult to achieve since metalevel computations needed to determine the best sequence of computations also take time. However, approximations to metalevel rationality can be useful and a large body of work has been devoted to the subject.

- **Bounded optimality** focuses on an agent performing as well as possible given its computational resources on an expected set of problems. Bounded optimality refers to the optimality of "programs" implemented to control the agent's computations and actions rather than on the actions or computations themselves. This notion is different from that of generating optimal actions, as defined by perfect rationality, or optimal computation sequences, as defined by metalevel rationality. Since agent programs create computation sequences, which ultimately determine the actions performed by the agent, they should be the focus of optimization. Bounded optimality refers to the optimality of an agent's expected performance. That is, a bounded optimal agent program performs, on average for a given distribution of problems, better than all other agents with the same computational resources on the same set of problems. For this reason, bounded optimality relies on having prior experiences (real or simulated) with the problem domain. This prior experience allows the agent to judge the utility of computational actions for the problem domain. Similar definitions of *bounded optimality* have been reached by others ([9], [24], [29], [51]). The metalevel controllers developed in this thesis are instances of bounded optimal agent programs.

To summarize, perfect rationality, calculative rationality and metalevel rationality are unrealistic goals for evaluating the quality of real-time agent design. Bounded optimality offers the most reasonable alternative. Perfect rationality for the betting problem implies that the agent can always miraculously place the best bet. Calculative rationality is achieved so long as the agent can guarantee that its computations will eventually lead to the same bet as that placed by the perfectly rational agent, regardless of how long it takes. Metalevel rationality is a statement about optimal computation sequences. While the definition of perfect rationality only defines an agent's selected action, a metalevel rational agent must optimize the sequence of computations it performs in order to select the best action. Due to time pressure, the best bet selected by a metalevel rational agent may not be the same as that of the perfectly rational agent.

For the sake of illustration, suppose that time pressure takes the form of a deadline and the agent is not allowed to place a bet unless the expected utility of the bet has been computed. An agent is said to be metalevel rational only if, among the entire set of bets whose expected utilities can be computed prior to the deadline, it chooses to compute the expected utility of the bet with the highest expected utility. This is clearly an unrealistic criterion for agent design.

Bounded optimality is achievable because it is a statement about the expected performance of an agent over a distribution of problems, which can be written as a constrained optimization problem. Unlike the ill-posed notion of metalevel rationality, which forces the agent to perform the perfect sequence of computations, this definition allows for the agent to make "mistakes". A bounded optimal agent may not perform well when judged on the basis of a single problem instance, but its expected performance, averaged over problem instances, is the best that can be achieved by all agents with the same set of computational resources, all else being equal.

This is a realistic optimization problem that can be framed and solved. For the betting problem, a bounded optimal agent, might have estimates, based on previous betting experiences, of the expected utility of each bet. Notionally, a bounded optimal agent program might proceed in a round of betting by computing the *actual* expected utility of each bet in the order of highest *estimated* utility (based on past experience) first. Computation would continue until the deadline and the agent would place the bet calculated to have maximal expected utility. The bounded optimality of such a program could be verified by comparing its expected performance to all possible programs with the same set of computational resources[1].

Developing a framework for guaranteed bounded optimal agent design is a goal of this thesis. Although sensible, bounded optimality can be computationally difficult to achieve.

2.2 Value of Information

The concept of *value of information* pervades this work. Value of information reflects the utility of obtaining an additional piece of information, where the utility is ultimately determined by how the information affects the final decision. Of real interest to this thesis

[1]Again, this thesis does not aim to maximize an agent's use of computational resources at the level of cpu commands. Indeed there are many unstated assumptions about computer architectures, programming languages and problem decompositions that are in the metalevel design problem. Instead these are abstracted away, and assumed to be the same for all agents. This leaves the definition of a bounded optimal agent as one which outperforms in expectation all other agents that have access to the same set of computational actions, all else being equal.

is the *value of computation*, which differs in a slight but important manner from the value of information. Namely, the value of information is never negative (colloquially known as "information never hurts"), while the value of computation can be, since resources are expended during computation. Part of the solution to the Markov decision process formulation of the metalevel problem found in Chapter 3 is to determine the value of computation for individual sub-problems. Understanding the approaches taken in determining the value of information can help in developing approaches for determining the value of computation. Before introducing the formal definitions of the value of information, a discussion of how it arises is presented.

Many approaches to metalevel planning can be viewed in terms of applying decision-theoretic approaches at the metalevel ([10], [42], [51]).

> **DEFINITION:** decision theory: a basis for making rational decisions formalized by the combination of probability and utility theory [39] (pgs. 465-466).

The metalevel decision problem is expressed in this thesis as a problem of planning under uncertainty, where probability theory is used to quantify this uncertainty. In addition, utility theory is used to quantify the agent's preferences over planning outcomes. Together they can be used to help the agent make decisions that maximize its expected utility, which is the utility averaged over all of the possible outcomes of the decision. The metalevel controller in this thesis is designed using decision-theory to maximize the agent's expected utility for solving sub-problems by enabling the weighing of the consequences of taking computational actions. An example of the application of decision theory at the metalevel is to consider the utility of running an anytime algorithm for an additional time step versus executing the current plan.

Howard's seminal paper [26] in 1966 on applying decision theory to analyze the value of information for a bidding problem, is summarized here. The value of information can be described in terms of expected utility in the following manner. Suppose that the current best decision is labeled α and ε is the background information, used to encapsulate the current state of knowledge of the agent. The expected utility of performing action α is the average over the outcomes resulting from this action given the background information. This can be expressed as

$$EU(\alpha|\varepsilon) = \sum_{o \in \mathcal{O}_\alpha} p(o|\alpha, \varepsilon)U(o), \tag{2.1}$$

where $\mathcal{O}_\alpha$ is the set of possible outcomes for action α, $U(o)$ is the utility assigned to outcome o, $p(o|\alpha, \varepsilon)$ is the probability of outcome o occurring given that decision α is made with information ε, and $EU(\alpha|\varepsilon)$ is the expected utility of making decision α based on information ε. Suppose that an additional piece of information ε_j is available, such that the expected utility of the decision given this new information is expressed as

$$EU(\alpha_{\varepsilon_j}|\varepsilon, \varepsilon_j) = \sum_{o \in \mathcal{O}_{\alpha_{\varepsilon_j}}} p(o|\alpha_{\varepsilon_j}, \varepsilon, \varepsilon_j)U(o), \tag{2.2}$$

where the set $\mathcal{O}_{\alpha_{\varepsilon_j}}$ represents the outcomes that can result from the new action α_{ε_j}, which is potentially a revised action in light of the new information, and $p(o|\alpha_{\varepsilon_j}, \varepsilon, \varepsilon_j)$ is the probability of outcome o occurring given decision α_{ε_j} and the new information. The expected value of information (EVOI) is the difference in the expected utilities of the decisions and is expressed as

$$EVOI(\varepsilon_j) = EU(\alpha_{\varepsilon_j}|\varepsilon, \varepsilon_j) - EU(\alpha|\varepsilon). \tag{2.3}$$

Note that the expected value of information $EVOI$ is always non-negative. If the information is beneficial, it will result in a higher utility decision, thereby making the $EVOI(\varepsilon_j)$ positive. If the information is not beneficial, then the final decision will remain the same as that without the information, and $EVOI(\varepsilon_j)$ is zero. As described above, the value of information corresponds to the case where the information provided is perfect. However, in the case where the information is provided indirectly, perhaps through some measurement. In this case, the information may contain errors, and the determination of the value of information will need to explicitly account for the probability of false measurements.

Having knowledge of the value of information (VOI) would allow for the trivial determination of the next action to take (either solve another sub-problem or execute the current best plan). Russell and Wefald [41] discuss an ideal metalevel control algorithm for decision making as simply:

1. Keep performing the available computation with highest expected net utility, until none have positive expected net utility.

2. Commit to the action α that is preferred according to the internal state resulting form step 1.

However, the determination of the VOI can become intractable in the face of real-world problems [21]. Many heuristic approaches have been used in the estimation of the VOI, where a typical assumption is that only one additional observation/computation/test will be performed. That is, the agent is given one chance to gather additional information prior to making its final decision. This is referred to as a myopic analysis, in that it may require a sequence of observations/computations/tests rather than a single one to result in a net positive utility. A myopic analysis will not be able to identify such a situation and could lead to sub-optimal decisions.

Heckerman et al. [21] discuss an approximate nonmyopic computation for determining the value of information by exploiting the statistical properties of large samples. The work presented in this thesis also attempts to determine the value of computation in a nonmyopic fashion. Heckerman's analysis of the problem involves determining the probability of a set of mutually exclusive and exhaustive hypotheses based on collected evidence, in order to apply the correct decision procedure. These hypotheses could represent illnesses that a patient might have. This approach relies on the simplifying assumptions that there are binary hypotheses (e.g., illness 1 or illness 2), binary decision procedures (e.g., treatment 1 or treatment 2) and binary and conditionally independent evidence variables (e.g. independent test results). Finally there is a utility value associated with each hypothesis-decision pair. Taking an observation incurs a fixed cost. The example problems explored in this thesis violate each of these assumptions, and thus this approach may not be readily applied. Most notably, the authors claim that it is difficult to generalize the analysis to a multiple-valued hypothesis, which the planning domains considered in this thesis possess.

The next section presents a discussion of various metalevel planning approaches which rely on the concept of the value of computation.

2.3 Metaplanning

Metaplanning can be a catch-all term for evaluating one's own reasoning processes [42]. Research in this area surfaced as a result of realizing that the theory of rationality and rational behavior as classically presented was an inappropriate measure for agents with limited computational resources. To that end, there was a surge of research in the late 1980s through the early 1990s which attempted to tackle many of the fundamental questions of metaplanning. Among the first to question the validity of the role of the classical rationality in decision making was Herbert Simon [43], starting in the 1950s.

In surveying the metaplanning literature, one major distinction that can be made, to distinguish the many different metaplanning problem formulations from one another, is the nature of the metalevel actions available to the agent. On one side, computational actions are modeled as discrete actions which consist of either atomic computations or the computation of complete decision procedures. On the other, the computational actions are considered to be continuous, and the metalevel controller is responsible for scheduling computation time for these continuous procedures. Different solution methods follow from the various modeling approaches. The work in this thesis more closely resembles the former treatment, where the computations are considered as discrete computational steps.

2.3.1 Discrete Control of Computation

The treatment of computation as discrete operations is a natural representation for the types of actions performed by digital computers. However, discrete metalevel control does not refer to manipulating the operations performed at the computer processor level, but at a slightly higher level of abstraction. A metalevel control action manipulates a basic computational step as defined by the scope of the metalevel problem being solved.

2.3.1.1 Optimal Satisficing

One approach to dealing with the problem of limited rationality is to *satisfice* rather than optimize, where the main premise is that satisficing will result in a solution more quickly than optimizing. One of the earliest works in minimizing search effort for satisficing was by Simon and Kadane in 1975. Though not specifically presented in the terminology of metalevel control, their work nevertheless exhibits concepts related to such an application. Simon and Kadane [44] present the notion of *optimal satisficing search*[2] where the aim is to reach any of the set of designated goal states with minimal search effort, as opposed to best-value search (an example of calculative rationality), which searches for the solution with the highest value.

[2]The term *optimal satisficing* may at first appear to be an oxymoron. In this case the modifier "optimal" is in reference to minimizing cost involved in **finding a satisficing solution** as opposed to maximizing the value of the final solution.

The basic problem is presented in the context of hunting for buried treasure, where an unknown number of treasure chests have been buried at n sites. Given are the probabilities of treasure being located at each site, $p_i, i \in 1...n$, the cost of excavating site i, $q(i)$, along with the condition that the hunt is terminated as soon as a treasure is uncovered. This is entirely isomorphic to a goal-directed problem where there are multiple algorithms or decision procedures (e.g., digging) that can be applied to solve the base-level planning problem. Each decision procedure has an *a priori* probability of success, such that each call to an algorithm results in the agent incurring a known cost of computation. Simon and Kadane arrive at the solution through an *interchange argument* [5] (pg. 182) which yields necessary and sufficient conditions for optimality in terms of minimal effort search. The optimal policy consists of selecting excavation sites based on decreasing $\phi(i) = p_i/q_i$. Intuitively this makes sense since this policy dictates selecting sites whose ratio of the probability of success to the cost of excavation is highest. Their analysis is extended to the case where ordering constraints exist at each site, and finally to the case of tree search. The tree search analysis relies on the identification of *maximal indivisible blocs* [*sic*], which are combined sequences of actions or computations whose separation would yield inferior results. Identification of these blocs can be considered analogous to the specification of a good sub-problems decomposition for the metalevel problem formulation. Once these blocs are identified, their p's and q's can be determined, and the optimal policy is again to select indivisible blocs in monotonically decreasing order of ϕ.

Simon and Kadane make several simplifying assumptions. First, their problem deals entirely with the binary outcomes (i.e., failure or success). That is, they want to reach any single goal, after which, the search will terminate. Their action set consists of methods that have a known probability of success for solving the entire planning problem (i.e., complete decision procedures). In addition, the methods are assumed to be independent of one another. That is, the probability of success for one method does not depend on the outcome of another. The problem formulated in this thesis differs from the formulation of Simon and Kadane in that rather than merely minimizing the search effort towards a satisficing solution, the metalevel must trade off a potential improvement in plan value against the cost incurred with further search effort. Rather than selecting among complete solution methods, the computational actions in the metaplanning problem select among individual atomic computations which reveal information about aspects of the problem. The problem in this thesis subsumes the optimal satisficing problem of Simon and Kadane. A metalevel controller can be generated by the formulation in this thesis to optimally satisfice by simply giving all possible goal states the same value.

Smith [45] addresses a similar problem in the context of first-order predicate calculus. He presents a methodology for efficient inference similar in nature to the metaplanning problem formulated in this thesis, where a sequence of inference actions is selected to maximize the total expected utility. Based on the approach in Simon and Kadane [44], Smith presents an algorithm that clusters actions into "indivisible" blocks and uses utility information of each block to determine their order. Once the utility or "worth" information about inference steps has been calculated, he proves that the optimal strategy is to be greedy with respect to worth. Independence of the inference steps is assumed. However, for the problems considered in this thesis, the independence assumption is not always valid because many of the problems considered here contain what Smith refers to as "redundancy" and "caching". Redundancy occurs when a portion of a search space of one inference action appears in another inference action's search space. Caching occurs when the results on one inference action can aid another. Both of these destroy the independence assumption and can make the algorithm produce sub-optimal results. On the other hand, the MDP formulation for metaplanning presented in Chapter 3 handles both of these issues inherently through the problem formulation. In addition, Smith's problem, like Simon and Kadane [44], deals only with finding a satisficing solution with minimal effort, where the utility function is binary (i.e., either success or failure). This thesis trades off the utility of a solution with the cost of finding it allowing for the explicit tradeoff between solution quality and computational cost, an ability that is inherently missing from satisficing methods. It may be possible to rephrase this optimization problem to fit within the context of Smith's approach [45], where the question becomes searching for the minimal effort strategy to find a solution of a specific utility level. Such a strategy may or may not be found, fitting the requirement of binary outcomes. Strategies could be generated for all possible utility levels and the best one, accounting for search effort, computed, though this appears to be computationally intractable.

2.3.1.2 Sequencing of Computations for Optimization

This section continues the discussion of metalevel problems whose action sets are discrete in nature. The difference from the previous subsection is that the objective of the metalevel planning problem is no longer just to minimize search effort, but to also account for solutions of varying utility, since some solutions are better than others. Etzioni [18] attempts to alleviate some of the assumptions of Simon and Kadane [44] by allowing for the possibility of a non-binary utility function over the final states of the world. Etzioni first considers a problem similar to that of Simon and Kadane, where there is a single goal and multiple methods (i.e., complete decision procedures), each with an expected computation cost and probability of satisfying the goal. Rather than having the same utility value for all methods, he allows for differing utility values among methods. Etzioni arrives at a conclusion similar to Simon and Kadane under this more generalized formulation. The optimal ordering is to sort the methods by greatest expected gain[3] over probability of success, $\frac{E[G(m)]}{P(m)}$, where $G(m)$ and $P(m)$ are the gain and probability of success of method m, respectively.

Etzioni formulates a related problem consisting of a single deadline, multiple goal states, and a single method for which there is guaranteed success of achieving each goal state. In addition, the time cost and utility for calling the method are given. The objective of Etzioni's problem is to select a sequence of methods that maximize the expected utility of the agent's actions while respecting the deadline. This is also very similar to the work of Einav and Fehling [17] where complete decision procedures are applied in a non-myopic manner in order to minimize the total cost of deliberation prior to taking action. Again, these complete decision procedures are each guaranteed to solve the problem, but with different resulting solution utilities. The formulation of the metaplanning problem in this thesis is similarly defined to the problems above, in that the objective is to find the best solution that minimizes the expected total cost due to both planning and execution.

As opposed to the approach in this thesis, each computational action in [17] and [18] is again modeled to be statistically independent complete decision procedures. This assumption is difficult to uphold, since many of the computational actions that are taken within each complete decision procedure may coincide. Because of this, partial computations of one decision procedure may yield information about how another might perform, making

[3]The gain of a method is its expected utility subtracted by its expected computation cost.

them *interdependent.*

This thesis assumes that individual computational actions do not constitute complete decision procedures, but are atomic computational parts that can be combined to form a complete decision procedure. Doing so allows for the metalevel controller to learn the best complete decision procedure from atomic parts rather than relying on an *a priori* set of computational procedures to be provided. In addition, the final metalevel control mechanism in [18] cannot be dynamically modified during run-time. Once an optimal sequence of complete decision procedures is determined, each is computed to completion in the specified order. If a decision procedure performs poorly, there is no mechanism for dynamically swapping the next decision procedure to compute. This differs from the metalevel controllers in this thesis, which, like Einav and Fehling [17], use feedback from previous computations to determine the next computational action to take.

Russell and Wefald ([41], [42]) give a discrete metalevel problem formulation similar to the work in this thesis. They present a general approach for the decision-theoretic control of reasoning in which computation is treated as discrete "chunks" that must be allocated as separate decisions. They develop a heuristic search algorithm called decision-theoretic A* (DTA*), which uses online metalevel planning to estimate the value of computation to prune the search space. The search continues for an additional step when the value of computation is found to be non-negative and halts, with the agent taking the best action computed thus far, otherwise.

Russell and Wefald make simplifying assumptions in the development of DTA*. The first is the *meta-greedy* assumption, which selects the computational action with highest immediate benefit. If the estimated benefit is accurate this assumption yields the optimal computational action. However, since the immediate benefit is estimated, the effect of this assumption is that it potentially ignores how these estimates might change with addition computation. They also make the *single-step* assumption, where the agent is assumed to be able to take only one additional computational action before executing. This assumption allows for the evaluation of the value of a single computational action as equivalent to a complete decision computation allowing for the metalevel controller to forgo having to consider the possibility of being able to take additional computational actions in the future.

When these two assumptions hold, they allow the agent to easily evaluate the expected value of computation, EVOC, exactly. That is, the meta-greedy assumption selects computations based on the immediate estimated benefit of a computational action, and the single-step assumption states that the immediate estimate of the benefit of a computational action is exact. In general, however, this is a myopic evaluation. Finally Russell and Wefald assume that a computational action will change the expected utility for exactly one base-level action, known as the *subtree-independence* assumption. This assumption does not hold for the problem domains considered in this thesis, since many of them are in the form of a graph. Under these circumstances, performing a computation can possibly affect the utility of many base-level actions.

The DTA* algorithm can be seen as an approximation to metalevel rationality, but does not guarantee bounded optimality. Russell and Wefald do present a clear theoretical discussion on the value of computation and how it differs from the value of information presented in Howard [26]. The main difference being that computation can negatively affect the utility of an action.

In their discussion, Russell and Wefald assume that the agent has a set of base-level actions A that can be executed in the environment and that there is a default action $\alpha \in A$ corresponding to the action which is currently perceived to be best. At each step, the agent also has a set of computational actions $\{S_i\}$ which can cause the agent to revise its current best action. The agent selects among the set of available options $\{\alpha, S_i\}$. The *net* value of computation for a complete computational action (i.e., a complete decision procedure) can be represented as the difference between the utility of the state resulting from an inference step and the utility of executing the current best action α, or

$$V(S_i) = U([S_i]) - U(\alpha) \tag{2.4}$$

where V is the net value of computation, U is the utility of a state, and $[S_i]$ represents the new state that results from having applied computational action S_i in the previous state. At the end of a complete computation, the agent immediately performs the best computed action. If the computational action results in a revised assessment of the current best action, then $U([S_i]) = U([\alpha_i, S_i])$ where $[\alpha_i, S_i]$ is the state outcome achieved by executing the revised action α_i after having performed computation S_i.

In the case of partial computations, where S_i does not provide an immediate revision of the best action, but subsequent computations might, the utility of S_i can be represented as:

$$U(S_i) = \sum_T P(T)U([\alpha_T, [S_i.T]]) \tag{2.5}$$

where T represents the sequence of subsequent computations following S_i, $S_i.T$ is the computation S_i immediately followed by T, and $P(T)$ is the probability of executing the sequence T.

In general, the agent does not know the exact utilities and probabilities, which must be estimated at the cost of computation. Let $\hat{Q}^S$ represent the estimated value of a quantity Q following a computation S. In this case, the estimated utility of $S.S_i$ is

$$\hat{U}^{S.S_i}([S_i]) = \sum_T \hat{P}^{S.S_i}(T)\hat{U}^{S.S_i}([\alpha_T, [S_i.T]]), \tag{2.6}$$

where the estimated utility of computing T and taking the revised best action, α_T, after T is

$$\hat{U}^{S.S_i}([\alpha_T, [S_i.T]]) = \max_j \hat{U}^{S.S_i}([A_j, [S_i.T]]), \tag{2.7}$$

and A_j is the set of all base level actions in A. Superscripting on utility estimates indicates that those values are the results of performing the indicated computations, rather than the true utility values. This, in effect, captures the essence of computation under computational resource limitations, where subsequent computations must rely on the estimates computed through partial computations of previous steps.

The estimates of utility are a result of computation, and the agent must choose the next best action based on these estimates. Based on this, the net value of computation is

$$\hat{V}^{S.S_i}(S_i) = \hat{U}^{S.S_i}([S_i]) - \hat{U}^{S.S_i}([\alpha]). \tag{2.8}$$

Since the exact results of the computation of S_i is not known before it is performed, $\hat{V}$ is a random variable whose expectation is used to determine the best action

$$E[\hat{V}^{S.S_i}(S_i)] = E[\hat{U}^{S.S_i}([S_i])] - E[\hat{U}^{S.S_i}([\alpha])]. \tag{2.9}$$

Recall Equation 2.3 from Howard's Information Value Theory [26]. The expected value of information of knowing ε_j is the difference in the expected utility of knowing and not knowing ε_j. Russell and Wefald show that this calculation is invalidated in the case of limited rationality when utility estimates are a function of the computation performed thus far. Translating Howard's (EVOI) equation into the present terminology yields

$$E[\hat{V}(S_i)] = E[\hat{U}^{S.S_i}([\alpha_{S_i}, [S_i]])] - E[\hat{U}^S([\alpha])], \tag{2.10}$$

where the expected value of the information gained from performing computation, S_i, is the difference between the expected utility of taking the revised best action, $\alpha_{S.S_i}$, and the previous best action, α. The equations 2.9, and 2.10 will only be equal when in the new state,

$$E[\hat{U}^{S.S_j}([\alpha])] = E[\hat{U}^S([\alpha])], \tag{2.11}$$

which Russell and Wefald refer to as the *coherence condition* [41]. That is, the expected utility of the old action, α, is unchanged after computation S_i has been performed. For a perfectly rational agent, this will be true since the utility of α in state S will have been properly reflected by $\hat{U}^S$. For an agent with limited computational resources, the utility of α can possibly be revised with S_i, which may result from a variety of reasons (e.g., additional computation yields information about α which reduces its utility, or the opportunity to execute α has passed). Russell and Wefald mention an additional difference in their notational representation of the value of computation that differs from the value of information presented by Howard. Consider the case where the result of a computation yields $\alpha_{S_i} = \alpha$ (i.e., the current best action has not changed). In this case $V(S_i)$ may actually be negative since all it did was to induce a time delay for taking the same base-level action, whereas in the calculation of the VOI, information gathering always results in non-negative value.

Thus far, the utility function has been employed to inherently capture the complexity of the combined effects of computational and base-level actions. In order to simplify this, Russell and Wefald make several assumptions. First, they assume the existence of *intrinsic utility*, which is the utility of a base-level action independent of time. That is, taking a physical action always has the same value. In addition, the *time cost* of computational actions is assumed to be independent of time, such that the utility of a state is represented as the difference in intrinsic utility and the time cost for the computation such that

$$\hat{U}([A_j, S_i]) = \hat{U}_I([A_j]) - TC(|S_i|), \tag{2.12}$$

where $\hat{U}_I([A_j])$ is the intrinsic utility of action A_j, TC is the time cost for computation, S_i, and $\hat{U}([A_j, S_i])$ is the estimated utility of taking action A_j after performing the computation S_i. Since not of the terms depend on the actual time, this property is referred to as *time-separability.* This implies that the expected value of computation can be simplified, and determined as the difference between the estimated benefit of computation $\Delta(S_i)$ and the cost of performing the computation:

$$\hat{V}^{S.S_i}(S_i) = \Delta(S_i) - TC(|S_i|). \tag{2.13}$$

The estimated benefit of computation, $\Delta(S_i$, is the difference in utility of taking the revised action, α_{S_i}, and the default action, α:

$$\Delta(S_i) = \hat{U}_I^{S.S_i}([\alpha_{S_i}, [S_i]]) - \hat{U}_I^{S.S_i}([\alpha]). \tag{2.14}$$

One of the difficulties with this assumption is that depending on the problem, the appropriate time cost function can be difficult to determine [10]. The modeling of hard temporal constraints, such as deadlines, is particularly challenging.

The formulation for developing metalevel controllers in this thesis are for problem domains where time cost separability holds, but it can be extended to the case where the utility of actions are time dependent. In addition, the solution of the metalevel MDP in this thesis allows for the utility of partial computations, as given in Equation 2.5, to be determined exactly, making the assumptions used by Russell and Wefald unnecessary.

2.3.2 Anytime Algorithms

This subsection presents an alternative method of metalevel control, where the focus is on allocating computation time to complete decision procedures possessing the anytime property. The terms "anytime" or "flexible" algorithm, attributed to Dean and Boddy [15] and Horvitz [23], respectively, describe algorithms that have the property of iteratively improving the planner's current solution as a function of the time allocated to computation. Although the work in this thesis does not employ anytime algorithms, they offer an alternative and not completely separate view of achieving bounded optimality. The main benefit of working with anytime algorithms under time pressure is that, unlike traditional

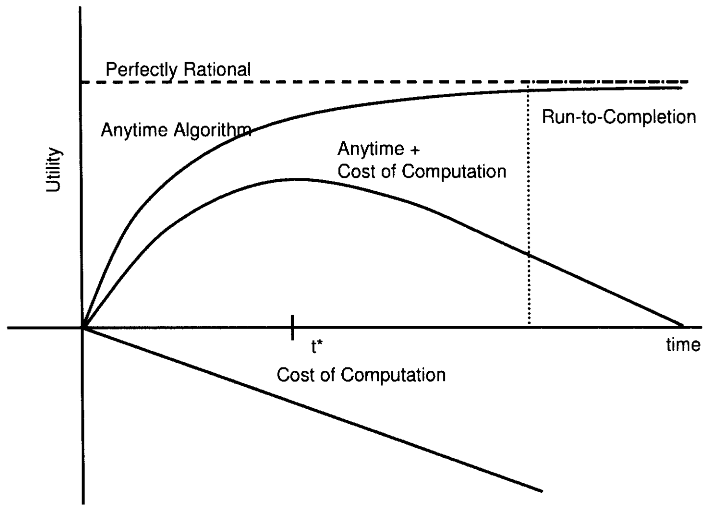

Figure 2-1: Effect of the cost of computation on an anytime algorithm.

run-to-completion algorithms, they are able to supply solutions of intermediate quality so that a feasible solution is available for execution at any point in time[4].

Figure 2-1 shows various curves, known as *performance profiles.* A performance profile summarizes the expected utility of executing the best action or plan recommended by an algorithm as a function of time. As in Russell and Wefald, time cost separability is typically assumed. In the figure, the y-axis measures the performance of each algorithm in terms of expected utility. This allows for the utility of an algorithm to be compared against the utility loss due to the cost of computation.

The performance profile of an anytime algorithm plots what Dean [14] refers to as *object-level* utility[5] as a function of time. Object-level utility corresponds to the inherent utility for taking physical actions (i.e. the utility of the plan independent of time). This is similar to Equation 2.12 when the time cost is zero. A notional plot of the expected utility of the plans generated by an anytime algorithm is given by the curve labeled "Anytime Algorithm" in Figure 2-1. Since an anytime algorithm iteratively improves over time, the plot shows that additional time spent on computation always results in a non-negative effect on object-level utility.

The maximum expected utility is achieved by running the anytime algorithm to completion. The introduction of a non-zero cost of computation function can result in a negative

[4]In general, when a solution must be generated from scratch there will be an initial period of time where no solution will be available.

[5]This is equivalent to Russell and Wefald's notion of intrinsic utility

effect on the utility of computation. This, in turn, affects the utility of running the anytime algorithm for differing amounts of time. Assuming that the cost of computation is additive, the agent's net or *comprehensive utility* function (equivalent to the first term in Equation 2.12) is labeled "Anytime + Cost of Computation". The comprehensive utility is the objective function to be optimized by metalevel planning, while traditionally the optimization is over object-level utility. As shown in Figure 2-1, the cost of computation can have a significant effect on the utility of the generated plans. Accounting for the cost of time, the maximum expected utility of the anytime algorithm is no longer achieved by running to completion. In Figure 2-1, the maximum expected utility accounting for the cost of computation is achieved at time t^*. Stopping prior to or planning for additional time steps beyond this point results in diminished comprehensive utility.

Also shown in Figure 2-1 are the performance profiles for a perfectly rational agent and a calculative rational agent. The generation of the best plan of the perfectly rational agent is achieved instantaneously at time $t = 0$. The calculative rational agent's performance is given by the curve labeled "Run-to-Completion".

DEFINITION: run-to-completion algorithm: an algorithm which generates the optimal base-level solution upon termination with no intermediate results.

The difference between anytime algorithms and run-to-completion algorithms is that run-to-completion algorithms never produce intermediate results. In this case, the advantage that the anytime algorithm has over the run-to-completion algorithm is that the comprehensive utility of the run-to-completion algorithm (not shown) is actually negative over the range of computation times. At no point during computation will the run-to-completion algorithm yield any positive net utility. However, when there is no cost of computation an anytime algorithm does not yield any appreciable benefit over the run-to-completion algorithm.

As opposed to the allocation of discrete computational actions, metaplanning problems using anytime algorithms focus on the allocation of computational effort, in the form of computation time, to a single algorithm or a system of anytime algorithms. While the allocation of discrete computation focuses on the details of what to compute, the anytime representation of computational actions abstracts away these details. That is, the specific computational actions taken by an anytime algorithm are not directly under the control of the metalevel controller, who use it as a blackbox. Anytime algorithms are also considered complete decision procedures since they solve the entire base-level problem. Allowing for anytime algorithms to replace non-anytime decision procedures in some of the previous approaches mentioned for the discrete allocation of computation [17], [18] and [44], may offer additional flexibility for making the tradeoff between computation effort and solution quality.

Horvitz ([23], [25]) concentrates on the theoretical problems involved in determining the amount of deliberation for a single task, such as identifying t^* in Figure 2-1 (also known as the stopping problem). He explores the effects of different representations of the cost of time on the optimal choice of stopping time for flexible decision procedures. Although he focuses on the allocation of time, he mentions that the resource allocation problem defined in metaplanning is not restricted to computation time, but can be expanded to include memory, design time, investments in knowledge acquisition, and hardware capabilities of the system. Horvitz lists two classes of desiderata that are useful for reasoning under scarce resources, *flexibility* and *bounded optimality* (similar in concept to the discussion in Section 2.1). Horvitz describes flexibility in terms of several attributes:

- *Value continuity* refers to the property that the object-level utility of a problem solving strategy is a continuous function of the amount of resource allocated.

- *Value monotonicity* refers to the property such that object-level utility of a problem solving strategy is a monotonically increasing function of the amount of resource allocated.

- *Convergence* refers to the property that a strategy demonstrate convergence to the optimal object-level value at some level of resource expenditure

- *Value dominance* refers to strategies that contain value dominant intervals, ranges of resource fraction where the gain in comprehensive value of computation is a monotonically increasing function of resource allocation.

These desiderata act to limit the nature of the algorithms over which bounded optimality with anytime algorithms is to be achieved. They are interesting from the perspective of metalevel problem formulation in this thesis and serve to highlight some differences between approaches. The first attribute, value continuity, is irrelevant to this discussion since computational actions are modeled as discrete actions in this thesis. Value dominance, however, is certainly desirable since it is hoped that the expenditure of computational resources results in a net gain in comprehensive value. Value monotonicity is also desirable, but determining the overall utility of the plan given a set of solved sub-problems is a tricky value assignment problem. Selecting the order of sub-problems to solve such that sub-problem value monotonicity is maintained is an objective for metaplanning in this thesis.

The attribute of convergence is the most in contention. Since convergence of the plan to the optimal object-level utility may not always be an appropriate condition. Convergence is a reasonable goal in cases where the cost of computation or inference-related costs are zero. In cases where the cost of computation is high, however the best strategy may be one that does not ever converge to the optimal object-level value. Even if the algorithm were able to converge to the optimal solution, the comprehensive value may no longer be positive at the convergence point, rendering the computations useless.

Boddy and Dean [10] also present an algorithm for determining the allocation of computation time to anytime decision procedures. As with Horvitz, they base their approach upon decision-theoretic principles, and refer to metalevel planning as *deliberation scheduling*, since they deal with scheduling computation times. Deliberation scheduling "involves the explicit allocation of computation resource based on the expected effect of those allocations on the systems's behavior." They describe, similar to Russell and Wefald [42] and Horvitz [24], the distinction between *inferential actions* and *physical actions*. Inferential actions (i.e., computational actions involved in running decision procedures) have no effect on the world external to the agent and only affect the agent's internal state. Physical actions actually change the state of the external world. Inferential actions are of interest since they consume computational resources that might be spent otherwise and revise the agent's estimations of how to act. They argue that an agent that is capable of reasoning about its "computational capabilities must have expectations about the potential value of its computations and how long those computations are likely to take." As with other adopters of anytime algorithms, this information is summarized by performance profiles. Boddy and Dean consider the deliberation scheduling problem for several classes of performance profiles, starting with those that are suitably approximated by piecewise linear monotonically increasing functions.

In order to guarantee the optimality of their scheduling algorithm, called deliberation scheduler (DS), they impose the additional constraint that the slopes of the consecutive

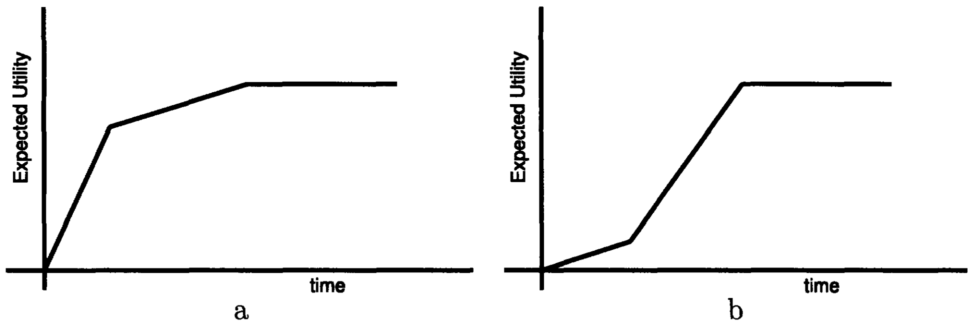

Figure 2-2: Performance profiles plotting the expected utility of the decision as a function of computation time for two algorithms. Algorithm 1 shown in a) exhibits diminishing returns while Algorithm 2 in b) does not.

line segments be decreasing, such that they exhibit *diminishing returns*. Figure 2-2 shows an example of two performance profiles, one of which exhibits diminishing returns. The useful property of diminishing returns is shared by many iterative algorithms, but not by all. Boddy and Dean [10] give the example of an algorithm which determines the 10-digit combination of a safe, where the probability of correctly determining the remaining digits is a function of the known digits. In this example, the expected utility gain in the first step is $9e^{-9}u$ and the expected utility gain in the last step is $0.9u$, where u is the expected utility for opening the safe. Here the gain of the last step far exceeds that of the first.

Another metalevel planning problem involving anytime algorithms is the allocation of a fixed amount of computation time across a set of anytime algorithms that are connected to one another. Zilberstein [51] presents a technique for the efficient scheduling of run times for a system consisting of individual anytime components. He discusses several ways of composing anytime components, which under some restricted conditions result in an optimal allocation. A variety of programming architectures are examined. Sub-problems in this thesis can be thought of as equivalent to the components of a system. The work of Zilberstein in the compilation of anytime algorithms differs from the work in this thesis in that the configuration of components that are activated during execution is not fixed, but conditioned on the results of the computations that have occurred during run time. In terms of the problem formulation of this thesis, Zilberstein assumes that the set of sub-problems over which computation time must be allocated is part of the problem input, whereas the metalevel controllers of this thesis use run-time feedback from computation to select the next sub-problem.

2.3.3 Another Categorization of Metaplanning Problems

Although the division between metaplanning approaches presented above was based mostly on the nature of metalevel actions, it is certainly possible to make the division in another fashion. Conitzer and Sandholm [13] choose to categorize metalevel planning problems into 3 categories and relate their corresponding complexity results.

1. The first problem in their categorization is that of allocating computation time across anytime algorithms with nondecreasing performance profiles given N deliberation steps and a target value of K. It is shown to be NP-complete by polynomial reduction to the KNAPSACK problem. If the performance profile of the algorithm is piecewise linear and concave, the metareasoning problem is solvable in polynomial time according to the DS algorithm by Boddy and Dean [10].

2. The second problem described in Conitzer and Sandholm [13] is related to discrete allocation of computation similar to the problem formulated by Russell and Wefald [42]. The agent must decide on an observation action at each stage, and eventually act. This is shown to be NP-hard by reducing an arbitrary KNAPSACK problem to it.

3. The last problem discussed in Conitzer and Sandholm [13] is the state-disambiguation problem, similar to the problems found in this thesis, where the agent is initially unsure of the state of the world, and must make queries to disambiguate its true state. In general a query may have multiple answers that are consistent with the true state of the world. In these instances, it is assumed that the answers are uniformly and independently chosen from the possible answers. For the case where queries have only one consistent answer, they show that the problem is NP-hard by reducing SET-COVER to it. In the general case, the state-disambiguation problem is shown to be PSPACE-hard by reducing the STOCHASTIC-SAT(SSAT) to it, where SSAT is PSPACE-complete.

Conitzer and Sandholm state that these results are not an argument against metareasoning, but direct the focus of research towards other interesting avenues. These include:

"1) investigating the complexity of metareasoning when deliberation (and information gathering) is costly rather than limited, 2) developing optimal metareasoning algorithms that usually run fast, 3) developing fast optimal metareasoning algorithms for special cases, 4) developing approximately optimal metareasoning algorithms that are always fast, and 5) developing meta-metareasoning algorithms to control the meta-reasoning [process]...".

2.4 Metaplanning Discussion

In light of this prior work in the field of metalevel planning, the research focus of this thesis is most theoretically aligned with the general value of information based approach of Russell and Wefald. However, Russell and Wefald's objective was to create an algorithm for achieving metalevel rationality. As described in Section 2.1, this is not realistically attainable. Their DTA* algorithm approximates metalevel rational behavior while relying on simplifying assumptions. Rather than a fully generic heuristic algorithm like DTA*, which can be applied to arbitrary problems, the metalevel controllers in this thesis are created to achieve or approximate bounded optimality. For this reason, the metalevel controllers are specific for, and limited to, the distribution of problems for which they are generated. This maps to point number three in Conitzer and Sandholm's discussion. The goal of this thesis, in their terms, is to generate metareasoning algorithms for restricted classes of problems.

Many of the approaches to metalevel planning, both discrete and anytime, presented above ([10], [24], [17], [18], [44], [51]) relied on using complete decision procedures. The metalevel controllers developed in this thesis use atomic computational actions, which must be combined together to produce a complete decision procedure. This gives an extra degree of freedom for directing computational effort. In fact, the ability to select from individual atomic computations makes supplying a variety of complete decision procedures unnecessary, since the best one can be determined and synthesized directly from atomic computations.

When given a set of fixed complete decision procedures, the agent is at the mercy of the computations performed by the particular algorithm being used. For instance, a fixed anytime algorithm has a specific performance profile which the metalevel controller cannot affect. However, the flexibility afforded by combining individual computation actions can

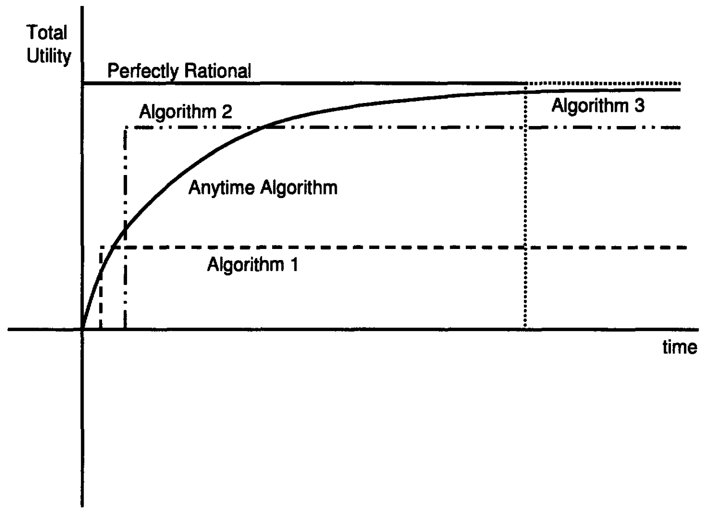

Figure 2-3: Examples of performance profiles for a variety of algorithms. Algorithm 1 is highly sub-optimal but responsive. Algorithm 2 is less responsive, achieves a better but non-optimal level of performance. Algorithm 3 is slow but generates the optimal solution.

result in the generation of metalevel controllers of varying forms. Consider Figure 2-3, which shows the performance of three algorithms or metalevel controllers labeled Algorithm 1,2 and 3 as a function of computation time. The goal of metaplanning in this thesis is to design a metalevel controller to generate behaviors that are optimized for the situations that are expected to be faced by the agent. For example, Algorithm 1 achieves a low utility solution quickly but does not reach optimality and may be useful in situations that warrant a quick response. Algorithm 3 solves the problem to optimality but takes a long time, and Algorithm 2 is somewhere in between. Being able to select individual computational actions allows the flexibility of generating computational behavior that can range over all possible algorithms that can be generated using the atomic computational actions available, an ability that is not present when using a set of fixed complete decision procedures or anytime algorithms. This is indeed a useful property considering the fact that bounded optimality refers to determining the best program, or algorithm, for a class of problems.

2.5 Chapter Summary

Perfect rationality as prescribed by classical decision theory is an unrealistic goal for the design of agents with limited computational resources. An alternative and more realistic goal of bounded optimality was proposed. Bounded optimality is a specification over agent programs acting over a distribution of problem environments, rather than over individual computational sequences as in metalevel rationality. Though metalevel rational systems are difficult to realize, the concept brought to the surface various approaches to account for the cost of computation involved in determining the best action. These approaches to metalevel planning can be categorized by the nature of the metalevel actions available. Some approaches employ the abstraction of anytime algorithms focussing the metalevel decisions on the optimal allocation of computation time. Approaches which employ discrete computations at the metalevel have an additional level of control that anytime algorithms do not have, namely what to compute. The formulation of the metalevel problem in this thesis is an instance of the latter. What makes it different from previous approaches is that the computational actions are atomic in nature as opposed to being complete decision procedures. When combined with information about the problem domain and the cost of time, an optimal policy or algorithm can be determined in order to control the computational behavior of an agent in a bounded optimal manner. The mechanisms involved in making these decisions are discussed next in Chapter 3.

Chapter 3

Metalevel Planning Problem Formulation

This chapter presents the details of formulating the metalevel planning problem as a Markov Decision process (MDP), and will show how the MDP formulation is suitable for metalevel planning problems that have a natural sub-problem decomposition. The result of solving the MDP is a metalevel policy which is then applied online to dictate which sub-problems to solve. This policy can be viewed as a closed-loop controller that conditions the next computational action on the outcomes of previous computations. When the sub-problem outcomes of the metalevel environment are finite and discrete, the MDP formulation developed in this chapter is an exact formulation of the problem and results in an optimal policy. This optimal policy is the metalevel controller that result in bounded optimal agent behavior, the best expected performance that can be achieve by any agent under the same circumstances. When the sub-problem outcomes fall within a bounded but continuous range, a discretization is applied to retain a discrete representation of the problem.

3.1 Hierarchical Decomposition

This thesis assumes the existence of a hierarchical decomposition of the base-level planning problem into a two-level hierarchy composed of a master level and a sub-problem level. Hierarchical problem decomposition is an approach for solving many large-scale planning problems and refers to restructuring the original problem into a hierarchy with smaller, easier to solve sub-problems, whose solutions are combined to retain the constraints and objectives of the original problem [33]. In this thesis, hierarchical decomposition of the base-level problem enables a natural formulation the metalevel planning problem.

> **DEFINITION:** base-level planning problem: the underlying planning problem to be solved (e.g., determining the sequence of physical actions to take).

> **DEFINITION:** metalevel planning problem: the problem of planning the computational actions used in solving the base-level planning problem.

The metalevel controllers in this thesis are designed to output control actions that consist of discrete atomic computational actions. The availability of a hierarchical problem decomposition lends itself to defining a single atomic computational action as generating a solution to an individual sub-problem. Therefore, the control actions of the metalevel controller are defined as solving individual sub-problems. Bounded optimal behavior is achieved by generating the best metalevel controller whose computational actions are selected in such a way that the master level is able to achieve the highest expected utility for a given distribution of problem instances, net the cost of computation.

> **DEFINITION:** master level: the top level of the two-level hierarchy of the base-level problem. It takes individual sub-problem solutions and uses them to generate complete base-level solutions. Assumed to have negligible computational costs.

> **DEFINITION:** sub-problem level: the lower level of the two-level hierarchy of the base-level problem. Sub-problem solutions are building blocks (plan fragments) passed up to the master level to generate a solution to the base-level problem. A sub-problem is defined relative to a specific problem decomposition and can take the form of atomic-level computations or can themselves be planning problems.

> **DEFINITION:** atomic-level computation: the smallest quantity of computation that can be performed by the agent. In this thesis, solving a sub-problem is defined as an atomic-level computation.

The metalevel planning problem with sub-problems as computational actions explicitly assumes that the cost of computation is incurred only through the solution of sub-problems and not though their combination in the master level to generate the final base-level plan. It is not an unreasonable assumption given a "good" problem decomposition, where a good decomposition for metalevel planning will consist of a master level problem that can be solved with ease once it is presented with a set of sub-problem solutions. The issue of generating a problem decomposition is domain dependent and is not addressed in detail in this thesis. Instead, it is assumed that metalevel planning is performed given a specific problem decomposition provided externally by a domain expert. For some problem domains, a problem decomposition may naturally be present.

3.1.1 Roles of the Hierarchy

Functionally, the master level combines the sub-problem solutions to generate a complete solution to the base-level problem. Sub-problem solutions can be thought of as plan fragments which the master level pieces together while respecting the global constraints and objectives of the original planning problem to form a complete base-level plan. Because of this, the quality of the plans generated by the master level depends entirely on the pool of available sub-problems from which it can select. For instance, sub-problems in the time-critical targeting problem from Chapter 1 might consist of generating missions for specific subregions over the map.

The master level problem would be to choose a combination of generated missions in order to maximize the value of the global solution. The master level would also need to consider potential resource conflicts in making its selection of missions when multiple missions utilize overlapping resources. Ideally, if all possible sub-problems are solved and their outcomes provided to the master level, the "optimal" base-level solution can be determined instantaneously[1]. However, under the conditions of limited rationality, generating the solutions to all sub-problems can be very costly, or even impossible, when there are deadlines.

As mentioned above, it is assumed that the main expenditure of computation is devoted to solving sub-problems. The sub-problems are assumed to be defined as part of the base-

[1]Under the assumption that master level computations take a negligible amount of time, the optimal base-level solution is obtained nearly instantaneously when provided the outcomes of all of the sub-problems *a priori.*

level problem decomposition. They can be as simple as node expansion in a search tree, or as complicated as being a hierarchical optimization problems themselves. In this thesis, the abstraction is made such that solving a single sub-problem constitutes a single atomic level computation. The manner in which sub-problems are solved are also assumed to be specified as part of the problem description. There is no restriction on how they are solved. The metalevel planning process needs only to know the expected computation time for solving each sub-problem in order to account for its cost in the optimization problem to be solved.

The problem architecture, as presented in Figure 1-3, is similar to that found in the *composite variable formulations* for integer programming problems [2]. The composite variable approach also depends on a problem decomposition that allows for the generation of a pool of "options" (i.e., sub-problem solutions) that are combined to form the complete solution. An important difference is that computational costs of generating "options" is typically not considered in the composite variable approach, as it is mostly concerned with the problem of maximizing object-level utility. This is equivalent to having the "options" generated *a priori* and stored for later use. For the problem domains considered in this thesis, it is assumed that there are enough variations that occur in sub-problem instances to prevent them from being solved ahead of time and stored. For example, generating a single mission for a subregion in the time-critical targeting problem and expecting to be able to use it in all circumstances is not realistic. From one problem instance to another, targets may have shifted or moved out of the subregion, necessitating that the sub-problem be solved in real-time.

Having established that the metalevel planning problem for hierarchically decomposed problems will consist of selecting sub-problems to solve, and that solving all sub-problems is likely not "the rational thing to do" under limited computational resources, the goal in this thesis is to generate a metalevel controller that will suggest "useful" sub-problems to solve. It is assumed that prior to solving a sub-problem there is uncertainty in regard to its outcome. Solving the metalevel planning problem identifies these "useful" sub-problems, accounting for their outcome uncertainty, and incorporates them into an optimal policy. The usefulness of a sub-problem will vary depending on the problem decomposition, the current state of computation, as well as the cost of time and will be captured by the policy. The metalevel policy is also responsible for determining when to stop computing in order to execute the current best plan.

The metalevel controller cannot be expected to suggest useful sub-problems to solve under arbitrary circumstances. It would be difficult for a metalevel controller to do much better than random guessing when given a problem domain that it has not previously encountered. Instead, the metalevel policies that are generated in this thesis are aimed toward maximizing the expected utility of the agent's behavior over a collection of base-level problem instances over which they are trained. That is, the performance of the metalevel policy is optimized such that it achieves the best average performance over all problem instances the agent might encounter, coinciding with the condition of bounded optimality.

It is assumed that the base-level problem instances are drawn probabilistically from a fixed set of similarly defined problems. The probability distribution over the fixed set of problems instances will give rise to a probability distribution over sub-problem outcomes for a given problem decomposition of the base-level problem. The probabilities will be used in formulating the metalevel planning problem as an MDP to determine the utilities of sub-problems so that a policy for performing computations can be generated. These probabilities will either be given as part of the problem description or learned through off-line simulation.

The metalevel planning environment is the domain over which metalevel planning occurs. It describes the probability distribution over the set of base-level problem instances expected to be faced by the agent. This distribution of problem instances naturally produces a distribution over sub-problem outcomes.

DEFINITION: metalevel environment: the distribution of the set of all possible problem instances to be faced by the agent. This also yields a distribution over the set of sub-problem outcomes for a given problem decomposition.

3.2 A Canonical Metalevel Planning Problem

A canonical example of the general class of metalevel planning problems targeted in this thesis is presented by the graph in Figure 3-1. The graph simply consists of a set of nodes, N, and a set of arcs, A, connecting the nodes, and is referred to as a *graph configuration.* The goal is to find a minimum cost path from S to G by computing the utility (cost) of each arc. For this example, it is assumed that the utility of each arc is unknown but constrained

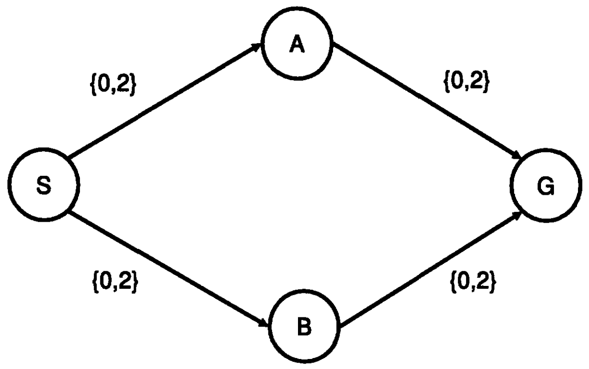

Figure 3-1: The canonical 4 arc metalevel planning problem.

to be within a finite set. This graph corresponds to a base-level problem in the form of a shortest path planning problem and naturally yields itself to a sub-problem decomposition where each arc represents a sub-problem. Notice that each arc in Figure 3-1 is labeled with a set of arc costs, $\{0, 2\}$. These represent the set of possible outcomes that can result as a consequence of solving the sub-problems. Recall that the metalevel controllers in this thesis are designed to act in a bounded optimal manner for a particular metalevel environment, where the metalevel environment is the distribution of the set of problem instantiations expected to be faced by the agent.

For any particular instantiation, the problem is simply a shortest path problem from S to G. Figure 3-1 with the given binary outcomes results in 2^4 possible problem instantiations shown in Figure 3-2. Each graph in the figure is an instantiation of a possible set of arc cost outcomes that can be realized in the canonical example. The metalevel environment is characterized by a probability distribution over these base-level problem instantiations. It is assumed that due to the variability in problem instances, the agent initially has no knowledge of which instantiation over which it must plan[2]. The metalevel must optimize the agent's computational behavior (e.g., computing the utility of arc or equivalently solving sub-problems) over the distribution of problem instances. This graph will be used throughout this chapter to demonstrate a variety of salient points regarding the MDP metaplanning formulation.

When the problem given in Figure 3-1 is viewed as a graph, the computations required to

[2]If the agent did know, the problem instantiation, it could potentially store the optimal solution and look it up. It is assumed in this thesis, that the problem instantiation is eventually revealed through the solution of enough sub-problems.

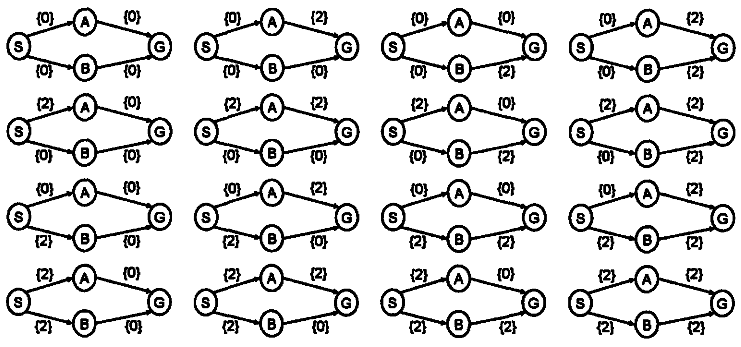
Figure 3-2: Problem instantiations of the 4 arc problem with binary outcomes in $\{0, 2\}$.

"solve" a sub-problem merely consist of something as simple as expanding a node to compute the corresponding arc cost. However, abstractly, the graph can be used to represent a variety of arbitrarily complex planning problems. Consider, for example, a military operation where daily (or hourly) flights must be planned to monitor certain geographical regions for activities of interest. In this case, a sub-problem (arc) might consist of performing the computations necessary to generate a flight plan over a specific region. The optimal flight plan could be arbitrarily complex to compute due to a variety of factors such as the number and types of aircraft to send, routing to avoid threats, air traffic constraints, etc.

In addition, the graph configuration itself may be used to represent precedence constraints of a problem, which in this case corresponds to the legs of a particular flight plan. Suppose that certain geographic regions cannot be reached without flying over others. In terms of Figure 3-1, the sub-problem represented by arc AG cannot be reached without flying through the region represented by arc SA (the same applies to SB and BG).

Without computing the actual utilities of the sub-problems, the preferred path, either SAG or SBG is not known. Since this is a daily operation, the agent can rely on past experience to estimate the relative distributions of problem scenarios and sub-problem utilities. However, since there is variability between problem instances, the true sub-problem utilities for the current scenario will not be known with certainty without performing fresh computations. Solving a sub-problem will inform the agent of the utility of incorporating its solution into the final plan. The metalevel controller for this problem must determine which arc to compute, compute it, and based on its outcome, determine the next sub-problem to solve. All the while, it must also account for the cost of time incurred by computation. Here, the time costs may involve the costs due to having personnel wait for the completion of planning, the reservation of aircraft resources that might be used elsewhere, etc.

3.3 Basic Problem Solving Strategies

The focus of this thesis is to develop bounded optimal strategies for directing the computation for solving base-level problems. Recall that bounded optimality refers to maximizing the expected utility of plans generated over a distribution of problem instances while accounting for the costs of generating the plans. This section presents a variety of strategies, which will be referred to as *fixed-computational strategies*, or open-loop plans whose performances will be used to gauge the benefits of bounded optimal agents. The modifier "fixed" refers to the aspect of the algorithm where base-level computations are not under the closed-loop feedback control of the metalevel controller. These can also be considered instantiations of complete decision procedures as discussed in Chapter 2. Some possibilities are listed below:

- COMMIT: select a single feasibly executable path to the goal.
- RANDOM-DIRECTED: randomly select sub-problems to solve in a depth-first manner until the goal is reached.
- PLAN ALL: solve all sub-problems and report the best plan.
- OMNISCIENT: this is not an example of a viable open-loop plan, but included in this list to show a lower bound on any possible strategy.

In order to compare the strategies, assume that the cost of computation for each sub-problem is fixed and denoted by ϵ, and that the possible utility (cost) values for each sub-problem take binary values in $\{0, 2\}$ with equal probability. The value of epsilon can be thought of in terms of a marginal cost of computation, such that the (disutility) cost of solving a sub-problem is given by ϵ times the expected computation time to solve the sub-problem. In the canonical 4 arc example, it is assumed that each sub-problem is computed in unit time so that they each incur a computation cost of ϵ. Referring to Figure 3-1, let arcs

SA and AG be relabeled as sub-problems 1 and 3, and SB, BG be relabeled as sub-problems 2 and 4.

The simplest strategy and least costly in terms of computation, is to commit to one "path" and solve the corresponding sub-problems. Note that it is assumed that enough sub-problems still need to be solved to generate a complete and feasible plan for execution. That is, solving sub-problems, 1 and 2 is not enough to generate an executable plan to the goal. Here, "solving" sub-problems simply consists of observing an arc and learning its cost. For the flight plan example described earlier in this chapter, it might involve gathering intelligence, allocating resources and generating a flight plan for a specific geographical region. In any case, performing a computation incurs a cost of ϵ, assumed to be additive, and returns the cost of executing the arc/sub-problem plan. The expected cost of the COMMIT strategy on the canonical 4 arc problem is given by

$$\begin{aligned} E[\textsc{Commit}] &= E[SP_1 + SP_3] + 2\epsilon \\ &= 1 + 1 + 2\epsilon \\ &= 2 + 2\epsilon. \end{aligned}$$

Assume that COMMIT consists of the arcs SA and AG. Then SP_1 and SP_3 are the random variables representing the outcomes of the solutions to sub-problems 1 and 3. The expected cost of the strategy is a linear function of the cost of computation since the cost of computation is additive. Thus the expected cost can be described by two separate components, that of the base-level problem, and that of the computation involved in solving the base-level problem.

A more generic version of COMMIT is called RANDOM-DIRECTED, which solves at random one of the outgoing arcs (sub-problems) of the current node and continues doing so in a depth-first manner until the goal state is reached. Since the arcs are assumed to be directed, it is guaranteed that the goal state will be reached, and a feasibly executable path generated in the process.

$$\begin{aligned} E[\textsc{Random-Directed}] &= pE[top] + (1-p)E[bottom] + 2\epsilon \\ &= 2p + 2(1-p) + 2\epsilon \\ &= 2 + 2\epsilon, \end{aligned}$$

where p is the probability of selecting the top path. In this case, since both the top and the bottom paths are symmetric, the expected cost for RANDOM-DIRECTED is the same as for COMMIT. Neither the RANDOM-DIRECTED nor the COMMIT strategies guarantee an optimal base-level or object-level solution, but they are strategies that guarantee the least amount of computational cost to generate a feasibly executable path. This can be considered equivalent to optimal satisficing.

The manner in which the COMMIT strategy chooses a complete plan (and the corresponding sub-problems) can vary. In the above case the upper path SAG was selected at random. Another possibility is to invoke the *principle of certainty equivalence* [5], in which the expected values of each sub-problem are used. The problem then degenerates into an ordinary deterministic shortest path problem. In this case, each solution (top or bottom) has an expected cost equal to 2 so neither path appears to be better than the other.

The PLAN ALL strategy is analyzed next. This strategy involves solving all sub-problems in order to obtain the calculative rational solution, as described in Chapter 2. The expected cost of this strategy can be determined by considering the 16 possible problem instantiations that can occur, computing the optimal solution for each of them and weighting them by the probability of occurrence, while also accounting for the fact that all four sub-problems are solved.

$$\begin{aligned} E[\text{PLAN ALL}] &= E[\min(SP_1 + SP_3, SP_2 + SP_4)] + 4\epsilon \\ &= \frac{1}{16}(2+2+2+2+2+2+2+2+4) + 4\epsilon \\ &= 1.25 + 4\epsilon. \end{aligned}$$

This strategy is guaranteed to generate the optimal object-level cost, but in comparison to the COMMIT strategy incurs twice the amount of computation cost, with 4ϵ instead of 2ϵ.

Last of all, is the OMNISCIENT strategy, which requires advanced knowledge of sub-problems to solve in order to obtain the best feasibly executable plan including the computation costs. For this problem, the OMNISCIENT strategy looks exactly like the COMMIT strategy, where the commitment for every single problem instance is different, but always

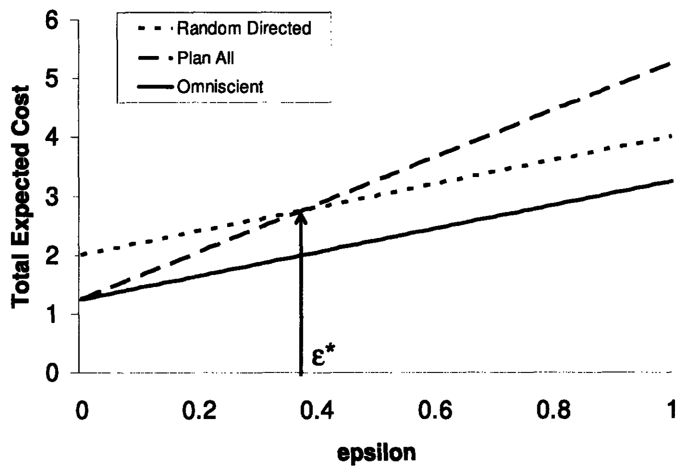

Figure 3-3: Performance of strategies as a function of ϵ, the cost of computation. ϵ^* is the optimal switching point for the switch strategy.

corresponds to the optimal path.

$$\begin{aligned} E[\textsc{Omniscient}] &= E[\min(SP_1 + SP_3, SP_2 + SP_4)] + 2\epsilon \\ &= \frac{1}{16}(2+2+2+2+2+2+2+2+4) + 2\epsilon \\ &= 1.25 + 2\epsilon. \end{aligned}$$

In this case, OMNISCIENT, which can be interpreted as obtaining the benefits of the PLAN ALL strategy, with an expected object-level cost of 1.25, at the computational cost of the COMMIT strategy, with a computational cost of 2ϵ. The OMNISCIENT strategy serves as a lower bound on the expected cost for any metalevel strategy, including the closed-loop policy presented in the next section. In terms of the categories of rationality explained in Chapter 2, OMNISCIENT can be thought of as the embodiment of metalevel rationality, since it always performs the "right" sequence of computations. Figure 3-3 compares the performance of the individual strategies as a function of the cost of computation, ϵ.

Each of these strategies, except for OMNISCIENT, can be considered a particular open-loop metalevel algorithm, in the sense that PLAN ALL dictates that all sub-problems be solved before generating the final plan, and RANDOM-DIRECTED limits the number of sub-problems to solve. Each of these algorithms has its usefulness for extreme values of ϵ. For instance, when the cost of computation is zero, it is easy to see that PLAN ALL is rational and achieves the minimum expected cost. When the cost of computation is high, RANDOM-DIRECTED is rational since it is the policy that achieves the minimum expected cost. In Figure 3-3, there is a transition point labeled ϵ^*, which indicates the change in dominance of the PLAN ALL strategy (lower cost is better) to that of the RANDOM-DIRECTED strategy. For this particular case, the switching point, the intersection of PLAN ALL and RANDOM-DIRECTED, occurs at $\epsilon^* = 0.375$. This suggests that one possible way of selecting strategies is to select PLAN ALL when computation costs are below ϵ^* and to select RANDOM-DIRECTED otherwise.

The following section discusses the formulation of the metalevel planning problem as an MDP and establishes the existence of metalevel policies that perform better than the ϵ^* switching policy suggested here. When the MDP is solved exactly, it automatically generates the best policy or algorithm at selecting computations such that bounded optimal behavior is achieved for each value of ϵ.

3.4 Metalevel Planning as a Sequential Decision Making Problem

The approach taken in this thesis for solving the metalevel planning problem is, in essence, to treat it as a *sequential decision problem* [32] in the space of partial plans and planning time. Sequential decision making refers to the act of making a sequence of decisions to determine the actions to take in the environment. The goal is to determine the "best" course of action, where "best" usually refers to maximizing the utility of taking each action. The decision making is described as *sequential* since actions taken at any step can influence the environment, the agent's internal state, and the possible choices of action in the next step. Figure 3-4 shows an example of the metalevel planning problem for the problem in Figure 3-1 as a sequential decision making problem. At each stage, the metalevel needs to make a decision as to which sub-problem to solve. Solving the corresponding sub-problem yields an outcome, which reveals new information. This process is repeated in the subsequent stages.

In Figure 3-4, s denotes the initial computational state. The diamonds labeled sp_i represent the set of possible actions that can be taken at each state. Suppose that action sp_1 is selected. This results in a stochastic outcome yielding state s' with probability p and state s'' with probability $1-p$. This process continues until, the *Exe* action is chosen,

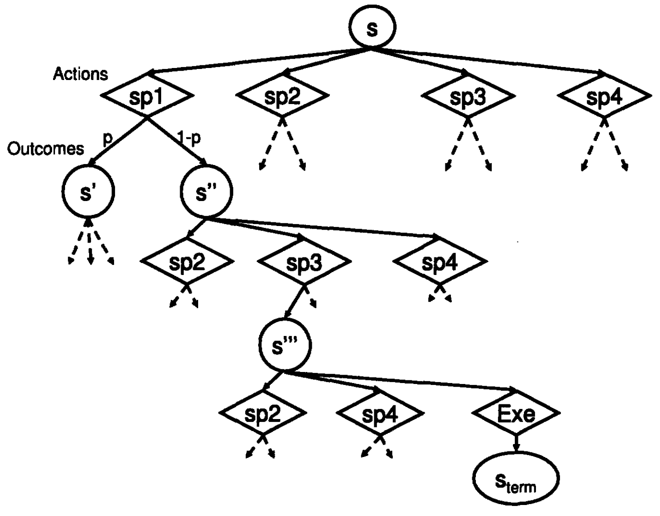

Figure 3-4: The 4 arc problem metaplanning problem as a sequential decision making problem.

allowing for the transition into a terminal state, s_{term}. Though the figure shows only one trajectory through the decision space, the dotted lines represent the other possibilities.

A desirable output for an automated sequential decision making system is a *policy* for making decisions. As discussed in [32], a *plan* is the simplest type of policy, where the agent executes the same actions regardless of information it receives, like the open-loop strategies of the previous section. Ideally a policy would condition the next action upon the current state of the agent. In the context of control systems, this difference is akin to open-loop versus closed-loop control. The sequencing of a list of decision procedures to run, as in Etzioni [18] is an example of an open-loop plan, whereas the metalevel policies generated in this thesis are closed-loop policies. These closed-loop policies can be thought of as algorithms that prescribe the computational actions an agent should take.

Sequential decision making problems may, in general, be placed into two categories, planning and reinforcement learning. Planning occurs when a complete model of the environment is given in advance and the optimal policy is derived from this information. In reinforcement learning, the model of the environment may not be known in advance or difficult to work with directly. Here, the policy is learned by the agent while it is actively exploring the environment. Under the circumstances of limited rationality, generating a policy in real-time while having to learn about the environment may not be viable due to the cost of learning during planning. For this thesis, it is assumed that the metalevel environment is known to the agent as part of the problem specification. In cases where the environment is unknown, additional effort is expended off-line to learn an appropriate model. Models that are to be learned or given should accurately reflect the utilities and probabilities that can result through the computation of sequences of sub-problems. The following section discusses this point in more detail.

3.5 Planning as a Markov Process

This section introduces the notion of how planning can be modeled as a stochastic process and, in particular, as a Markov process. A *stochastic process* $X = \{X(t), t \in T\}$ is a collection of random variables such that for each t in the set T, $X(t)$ is a random variable. If T is a countable set, then the process is called a discrete-time stochastic process [38]. A Markov process or Markov chain is defined as a stochastic process where the conditional distribution of a future state is dependent only on a finite history of past states. Mathematically, an n^{th}-order Markov chain is defined as

$$P(X_{t+1}|X_{0:t}) = P(X_{t+1}|X_{t-(n-1):t}), \forall t, \tag{3.1}$$

where X_{t+1} is the random variable representing the next state and $X_{0:t}$ is the complete history of states up to t. The simplest version is the first-order Markov process, where the conditional distribution of a future state depends only upon the current state,

$$P(X_{t+1}|X_{0:t}) = P(X_{t+1}|X_t), \forall t. \tag{3.2}$$

The outcome of a complete plan as well as the outcome of a single sub-problem can be considered a random variable. Even though for any particular problem instance and any deterministic planning algorithm the outcome is deterministic, by virtue of the fact that the problem instances are drawn probabilistically from the distribution provided by the metalevel environment, the outcomes are random. Therefore, even though the problem is solved by a deterministic algorithm, the outcome is uncertain until the problem is solved. Finding the metalevel policy involves selecting which sub-problems to solve, thereby making it a "decision process". Once a policy is determined, the policy itself is a Markov process, where state transitions are only dependent on the previous state. The Markovian nature of the decision problem is highly dependent upon the manner in which the states of the problem are defined. The states of the metalevel MDP are defined in terms of partial plans, represented as a tuple of the utilities (costs) for solved sub-problems (see Section 3.6).

In this thesis, the first-order Markov assumption is used exclusively, such that the evolution of the partial plan state is only a function of the previous state. The next section presents the theory of Markov decision processes.

3.5.1 Markov Decision Processes

This section presents the basic definitions and concepts underlying Markov decision processes (MDPs). A Markov decision process is a Markov process with the addition of actions and rewards. An MDP consists of a 5-tuple, $\{S, A, T, R, \gamma\}$, where

- S: is the set of states.
- A: is the set of actions that can be taken, which may be state dependent.
- T: is the set of state transition functions describing how the state evolves. The transition function is typically a conditional probability distribution that can be written as $T(s, a, s')$. This is meant to convey the probability of making a state transition into state s' if the current state is s and action a is selected (i.e., $P(s' \mid s, a)$). This is a direct consequence of the Markov assumption.
- R: is a reward function for describing the value of taking actions. The prevailing effect of taking a computational action results in a cost. So, the reward function is replaced by a cost function, C (i.e., a negative reward). In general the cost can be a function of both the state and action, $c(s, a)$.
- γ: is a discount factor, that is often used for infinite horizon problems. Most of the metalevel planning problems discussed in this thesis fall under the finite-horizon variety so that γ is not necessary and is set to 1.

The solution to an MDP is called a **policy**, π, which is simply a mapping from states to actions for all states in the MDP.

$$\pi : S \rightarrow A. \tag{3.3}$$

A policy is a closed-loop control law which dictates, at each state, the action to take to maximize reward (minimize cost). The policy acts as a closed loop controller for stochastic problem domains by correcting for uncertainties in the problem. A common objective function, assuming that costs are additive, is to minimize expected cost over a sequence of decisions. The expected cost of a state, s, is given as $V(s)$. Under the optimal policy, the optimal value function, $V^*(s)$, is given by the Bellman equation [39],

$$V^*(s) = \min_{a \in A}(c(s,a) + \gamma \sum_{s' \in S} T(s,a,s')V^*(s')), \forall s \in S. \tag{3.4}$$

A policy that satisfies Equation 3.4 is called an **optimal policy**, denoted π^*. For the optimal policy, Equation 3.4 must be satisfied for all states, and not just a particular state since the value at state s depends on the value of the states s', which in turn depend on the value of the subsequent states that can be reached. This defines a recursive relationship between stages of the decision making problem. For shortest path MDPs, there is at least one *terminal* state, whose value $V(s_{term})$ is known. Then according to the *principle of optimality* [5], this value can be used to *backup* the values of the previous stages. The principle of optimality simply states that the optimal policy for the remaining stages is independent of that of the previous stages and is a consequence of the Markov assumption, which is a useful realization that allows for the backwards stage-wise solutions of MDPs.

Bellman's equation is not a single equation, but really a system of equations (one for each state). There are a variety of ways to solve for optimal policies, two of which are presented in the following subsection.

3.5.2 MDP Solution Methods

Two iterative approaches commonly used to solve for the optimal policies in MDPs are value iteration and policy iteration [39]. Value iteration consists of solving the Bellman equation locally for each state s for $V(s)$ such that each local solution is called a *Bellman update*, or a *back-up*. This process is repeated until convergence. While the metalevel MDPs in this thesis mainly fall in the category of finite-horizon problems and will converge finitely, in general, the convergence to optimality of value iteration is guaranteed in the limit of Bellman updates being applied infinitely often.

For discounted problems, value iteration has the property [39] such that if the max-norm of the value function of the current iteration and the last iteration is no more than $\frac{\epsilon(1-\gamma)}{\gamma}$, then the difference between the current value and the optimal value is no more than epsilon. In this case, using the current value function to determine the policy will result in a value loss of no more than $\frac{2\epsilon\gamma}{(1-\gamma)}$. The value iteration algorithm is given in Algorithm 1.

Algorithm 1: Value Iteration

```
input  : $\{S, A, R, T, \gamma\}$, the MDP. $\epsilon$, an error tolerance for the value function.
local  : $V$,$V'$ vectors of value functions. $\delta$ the maximum difference in values
         over all states.
output: $\pi$ the resulting policy. $V$ the resulting value function.
begin
    $V \longleftarrow 0$;
    repeat
        $V' \leftarrow V$;
        for $s \in S$ do
            $V(s) \leftarrow \min_a(c(s,a) + \gamma \sum_{s' \in S} T(s,a,s')V(s'))$;
        end
        $\delta \leftarrow \max(\delta, \| V - V' \|_{max})$;
    until $\delta < \frac{\epsilon(1-\gamma)}{\gamma}$ ;
end
```

While value iteration focuses on using the value function to obtain the optimal policy, policy iteration can be seen as a way to work with the policy directly. Policy iteration assumes an initial policy, π from which a value function, V^{π}, based on the policy can be determined. The first stage of policy iteration is called *policy evaluation*, since it evaluates the value function for a given policy. Policy evaluation occurs by solving the system of n linear equations with n unknowns given by the Bellman equation (one for each state). Exact policy evaluation occurs in $O(n^3)$ time. For large state spaces this may be prohibitive and the policy evaluation step can be implemented as a simplified version of value iteration, shown in Algorithm 2. The next stage is called *policy improvement*, whereby the current policy is updated per equation (one for each state) according to the current state of the value function. As opposed to value iteration, policy iteration is guaranteed to terminate finitely since each step guarantees an improvement and there are only a finite number of policies, $|A|^{|S|}$. Each step of the value iteration algorithm can be seen as taking one sweep of policy evaluation followed by policy improvement.

These are by no means the only methods for solving MDPs, as they are the subject of ongoing research. Other approaches involve both exact and approximate methods. Value

Algorithm 2: Policy Iteration

```
input  : ${S, A, R, T, \gamma}$, the MDP. $\epsilon$, an error tolerance for the value function.
local  : $V$,$V'$ vectors of value functions. $\delta$ the maximum difference in values
         over all states.
output: $\pi$ the resulting policy
$V \leftarrow 0$;
$\pi \leftarrow A$;
repeat
    begin Policy Evaluation
        repeat
            $V' \leftarrow V$;
            for $s \in S$ do
                $V(s) \leftarrow \min_a(c(s,a) + \gamma \sum_{s' \in S} T(s,a,s')V(s'))$;
            end
            $\delta \leftarrow \max(\delta, \| V - V' \|_{max})$;
        until $\delta < \frac{\epsilon(1-\gamma)}{\gamma}$ ;
    end
    begin Policy Improvement
        policy-stable $\leftarrow$ true;
        $b \leftarrow \pi$;
        for $s \in S$ do
            $\pi(s) \leftarrow \arg\min_a(c(s,a) + \gamma \sum_{s' \in S} T(s,a,s')V(s'))$;
            if $b \neq \pi(s)$ then policy-stable $\leftarrow$ false;
        end
    end
until *policy-stable* ;
```

iteration is primarily used for solving the MDP formulations found in this thesis. In Chapter 4, a heuristic approach to solving the metalevel MDP will be an approximate form of policy iteration [8]. In Section 3.6, the specific details involving the formulation of the metaplanning problem as an MDP are presented.

3.6 MDP Formulation of the Basic Metalevel Planning Problem

This section presents a description of the metalevel problem in terms of Markov decision processes presented in the previous section. Recall that the objective of formulating the metalevel planning problem as an MDP is to generate a policy to act as a closed-loop feedback controller for computational actions (solving sub-problems). The next computational action selected is conditioned on the current state of computation. As will be shown shortly, the state of computation will be represented as an unordered history of solved sub-problem outcomes. The following assumptions regarding the properties of the metalevel planning problem are made in order to take advantage of the available machinery for solving MDPs:

- First-order Markov property and sub-problem independence: is the property that the sub-problem outcomes are statistically independent of one another. This independence allows for the first-order Markov assumption to hold given the state description of the metalevel MDP. Though there may be problems where the first-order assumption is violated, it can potentially be recovered with the inclusion of additional state variables.

- Time cost separability: is the property that the cost of time can be considered independent of sub-problem outcomes and is additive. This property allows for the reward (cost) function of the MDP to be defined.

- Discrete and static sub-problem outcomes: is the property that solving a sub-problem can possibly result in a finite set of discrete outcomes. Once a sub-problem outcome has been determined, it is assumed that it remains fixed for the remainder of the planning episode. This is an important property that serves as a stepping stone for the heuristic solution method developed in Chapter 4.

- Feasibly Executable: is the property that ensures that a complete plan to the goal can be generated from the set of solved sub-problems. Prior to the availability of a complete plan, plan execution cannot take place. That is, partial plans that do not reach the goal state cannot be executed by the agent. Planning must be completed prior to plan execution, and no additional planning takes place during execution. In Figure 3-1, having solved the sub-problems for arcs SA and AG makes the resulting plan feasibly executable. Though the agent could conceivably execute the partial plan as a result of having computed just SA, this is not allowed.

The following subsections detail all of the required elements (states, actions, costs, and transitions) for specifying the metalevel decision problem as an MDP.

3.6.1 Information States

The state, s, of the metalevel MDP consists of a vector representing all that is known about the current problem, effectively summarizing the current state of computation. The state will be represented by an information state vector, which encodes the state of computation as an $|SP|$-tuple, where $|SP|$ is the number of sub-problems in the base-level problem. Each element of the information state vector encodes whether a sub-problem has been solved and, if so, the cost of the outcome, such that

$$S = \{s \mid s \in \Re^{|SP|}\}, \tag{3.5}$$

where s is a tuple of real-valued elements of length equal to the number of sub-problems, S is the set of all such tuples, and s_i is used to indicate the information regarding sub-problem i within a specific tuple. Although sub-problem outcomes (costs) can in general be continuous, as indicated in Equation 3.5, this thesis will deal only with the discrete representation of sub-problem costs. Continuous sub-problem costs can be suitably discretized into a finite number of bins to meet this constraint. Each element of the information state vector holds the outcome values of solved sub-problems. When a sub-problem has not yet been solved, a place holder symbol, ?, is used as an indicator.

For discrete MDPs where sub-problem costs can take on m discrete values, such as in the 4-arc problem, the information state is also given as a tuple, but with discrete elements,

$$S = \{s \mid s \in \{?, b_1, \ldots, b_m\}^{|SP|}\}, \tag{3.6}$$

where the $\{?, b_1, \ldots, b_m\}$ is used to indicate the set of m discrete outcomes of a sub-problem, b_is are the particular outcome values, and the exponent is used to indicate that the state s consists of the cross-product of $|SP|$ sub-problem outcomes. For example, the information state for the 4-arc problem is composed of the cross-product of what is known about the four sub-problems. For each sub-problem there are three possible values of knowledge $\{0, 2, ?\}$. Either the sub-problem cost is known, in which case the sub-problem is represented as a 0 or a 2 accordingly, or the sub-problem has not been solved, and is represented by a ?. The set of all information states for the 4-arc problem is given in this notation by

$$S = \{0, 2, ?\}^4 \text{ representing } \{\{0, 2, ?\} \times \{0, 2, ?\} \times \{0, 2, ?\} \times \{0, 2, ?\}\}. \tag{3.7}$$

The specific information state, s, of having solved sub-problem 1 in Figure 3-1 and obtained an outcome of 0 with no knowledge of the other outcomes is represented as,

$$s = \{0, ?, ?, ?\}, \tag{3.8}$$

where the 0 in the first element indicates that sub-problem 1 currently is known to have value 0, and nothing is known about the remaining sub-problems. While the information states represent states of computations, a special terminal state, s_{term} is added to the set of information states to represent the state of having solved for and executed a plan. A transition to the terminal state is only allowed when a feasibly executable plan can be generated with the set of sub-problems solved thus far. The set of information states where plan execution may occur is referred to as the feasibly executable set FE, where $FE \subset S$. For the problem in Figure 3-1,

$$FE \subseteq \{\{0, 2\} \times \{0, 2, ?\} \times \{0, 2\} \times \{0, 2, ?\}\} \cup \{\{0, 2, ?\} \times \{0, 2\} \times \{0, 2, ?\} \times \{0, 2\}\},$$

where the first set of terms is the set of information states where the top path, SAG, is feasibly executable, while the second set represents states where the bottom path, SBG, is feasibly executable. The feasibly executable set of states is given by the union of these two sets.

3.6.2 Action Set

The set of actions allowed in the metalevel MDP are either to compute a particular unsolved sub-problem, or to execute the current best plan when it is feasibly executable. The set of computational actions is given by A_{comp} and the set of execution actions (there is only one) is given by A_{exe}. Let a_j be the computational action which solves sub-problem j and a_{exe} be the physical action which executes the best path found thus far. Then the set of possible actions at each state s is given by

$$A(s) = \begin{cases} A_{comp} & \forall s \in S \\ a_{exe} & \forall s \in FE \end{cases} \tag{3.9}$$

In general problem domains, the value of sub-problems outcomes may change with time. For example, the time-critical targeting problem of Chapter 1 has missions that can expire with a deadline, forcing the value of the mission to zero once the deadline is exceeded. This effect is generally not considered in this thesis, where the assumption is that *information is static* but unknown *a priori*. The consequence of static information is that once a sub-problem has been solved, the solution and corresponding cost stays fixed for the duration of the problem[3]. This eliminates the need to ever compute a sub-problem more than once, such that

$$A(s) = \begin{cases} a_j & \forall s \in S \text{ and } j \in SP \text{ and } s_j = ? \\ a_{exe} & \forall s \in FE \end{cases} \tag{3.10}$$

indicating that a sub-problem j is to be computed only when its cost is currently unknown.

3.6.3 Cost Function

The overall metalevel MDP cost function consists of two components. The first component is the cost of solving sub-problems, c_a. The second component is the cost of executing the final plan (referred to as the path cost below), determined from the minimum cost plan

[3]However, as will be shown in Chapter 6, the MDP formulation can be extended to include hard temporal constraints.

that can be generated from solved the set of sub-problem outcomes as given by state s. The metalevel MDP cost function is specified as

$$C(s,a) = \begin{cases} c_a & \forall s \in S, a \in A_{comp} \\ PC(s) & \forall s \in FE, a \in A_{exe} \end{cases} \tag{3.11}$$

where c_a is the constant cost of solving a sub-problem, and $PC(s)$ is the minimum path cost that can be generated by the master level given the current information state. For the 4 arc problem, the information state represented by $\{0,2,0,2\}$ yields a minimum path cost of 0. Both the top and bottom paths are feasibly executable, but the master level is able to use the set of sub-problems to select the best one, which in this case happens to be the top path.

For the basic metalevel MDP, where the additive time cost separability assumption holds, the cost of computation for computing sub-problem a, c_a, can be determined for a general time cost function $\mathcal{TC}(t)$ as the difference between the time cost function evaluated at the current time t and at a later time, $t + E[t_c]$. It is assumed that $E[t_c]$ is the expected time for solving sub-problem a.

$$c_a = \mathcal{TC}(t + E[t_c]) - \mathcal{TC}(t). \tag{3.12}$$

Typically the time cost function, $\mathcal{TC}(t)$, used in this thesis is given as a linear function of time with a y-intercept of zero and is therefore defined entirely by its slope, ϵ. This representation allows the cost of time to be interpreted as a marginal cost of computation. The units of ϵ are given by utility (cost) per unit time. In this case, the cost of computation of Equation 3.12 can be written as:

$$\begin{aligned} c_a &= \mathcal{TC}(t + E[t_c]) - \mathcal{TC}(t) && (3.13) \\ &= \epsilon(t + E[t_c]) - \epsilon(t) && (3.14) \\ &= \epsilon E[t_c]. && (3.15) \end{aligned}$$

Although the formal cost of time function is $\epsilon E[t_c]$, ϵ will be referred to informally as the cost of computation when the expected computation time for each sub-problem is the same.

Assuming that the cost of time is of this form, ϵ is an input to the metalevel MDP, and is a measure of the relative urgency of obtaining a solution. When ϵ is 0, computation is free, and the metalevel controller should take as much time as necessary to determine the optimal solution, since there is no utility loss with planning. The higher the value of ϵ, the more cost is associated with the time spent solving sub-problems. For high ϵ, the metalevel controller should restrict it computations to find a solution as quickly as possible.

3.6.4 Transition Function

The transition function represents the probabilistic effects of taking computational actions and describes the probability of the next information state given the selected action. The goal of the metalevel policy is to achieve bounded optimality over a set of problem instances determined by the *metalevel environment.* Recall that the metalevel environment is a probability distribution over the set of possible base-level problem instances. An example of the metalevel environment for the 4 arc example is shown in Figure 3-2, where each problem instance occurs with equal probability. This distribution determines the relative frequencies of the occurrences of sub-problem outcomes. The transition function of the metalevel MDP is used to represent the probability distribution over the outcomes of a solved sub-problem. The transition from any information state in FE to s_{term} is assumed to occur with certainty. That is, the a_{exe} is guaranteed to take the agent to the goal state.

$$P(s_{term} \mid s, a) = \begin{cases} 1 & \forall s \in FE, a \in A_{exe} \\ 0 & \text{otherwise} \end{cases} \tag{3.16}$$

The result of a computational action may in general affect the information of several sub-problems if they are defined in a way that does not make them statistically independent. For instance, if solving a sub-problem yields information that changes the outcome probabilities of another sub-problem, then they are not statistically independent. In general, this does not pose a significant theoretical problem to this formulation, since the transition function is state dependent. So long as the probability of the next information can be described as conditionally dependent only on the current information state (first-order Markov assumption), this model can still be applied.

However, if the sub-problems are assumed to be statistically independent of one another, as they are here, then the effect of a computational action will only affect the information corresponding to the sub-problem being computed, making the transition function significantly simpler. That is,

$$P(s' \mid s, a_j) = P(s_j), \tag{3.17}$$

where $P(s_j)$ is the *a priori* probability distribution over the range of cost outcomes for sub-problem j. This can be determined directly from the model of the metalevel environment. The metalevel environment of the 4 arc problem, shown in Figure 3-2, indicates that for each sub-problem, there is equal probability of obtaining either of the two binary outcomes. That is $P(s_j) = 0.5$ for all j.

3.6.5 Policy

The optimal metalevel policy, π^* (i.e., the metalevel controller), for the discrete problem formulation is determined either through Algorithm 1 or Algorithm 2 and will provide a mapping of the current information state to the optimal action. The resulting policy is largely influenced by the marginal cost of computation ϵ. Intuitively, for an ϵ of zero, the optimal metalevel planning policy should be expected to be the same as the PLAN ALL strategy, where information for all sub-problems is gathered, resulting in a final solution that is optimal at the object-level. When computational costs far exceed individual sub-problem costs, the best policy may be to do nothing. However the constraint of feasible execution forces some amount of computation so that enough sub-problems are solved to generate a feasible plan for execution. It should be apparent that an optimal satisficing approach is bounded optimal under the conditions of high computational costs.

3.6.5.1 Bounded Optimal Policies

This subsection shows that the exact solution of the metalevel planning MDP produces optimal policies that are a function of the metalevel environment and the cost of computation. For a given metalevel environment, the optimal metalevel policies generated by the MDP formulation are functions of the cost of computation, ϵ. The results of metalevel planning for the 4 arc problem is used to illustrate that the metalevel MDP formulation generates bounded optimal policies.

An agent program p is bounded optimal, when it achieves the highest expected performance over the metalevel environment, $\mathcal{E}$, given a particular agent architecture, $\mathcal{A}$:

$$p^* = \arg\max_{p \in \mathcal{P}} U(p, \mathcal{E}, \mathcal{A}), \tag{3.18}$$

where $\mathcal{P}$ is the set of all agent programs that can be generated within a given agent architecture, and p^* is the bounded optimal agent program. In the MDP model, agent programs take the form of policies and agent architectures correspond to the hierarchical problem decompositions. In direct analogy to Equation 3.18, the optimal policy, π^*, is the one which achieves the best expected performance for a given metalevel environment, of all the possible policies in Π, the set of policies that can be generated within a given problem decomposition:

$$\pi^*_\epsilon = \arg\max_{\pi \in \Pi} U(\pi, \mathcal{E}, \mathcal{A}, \epsilon). \tag{3.19}$$

The solution to the metalevel MDP for a specific value of ϵ will result in the generation of the bounded optimal metalevel policy, π^*_ϵ, for that particular value of ϵ as indicated in Equation 3.19. For the problems considered in this thesis, the value of ϵ is assumed to be part of the specification of the metalevel environment.

Plotting the expected cost of the bounded optimal metalevel policy over the continuous range of ϵ, yields a piecewise linear, concave curve composed of the dominant (minimum cost) policies over the range of ϵ. Concavity is ensured since the minimum function applied over a set of piecewise linear curves, representing the performance curves of the set of bounded optimal metalevel policies, is a concave function [7] (pgs. 16-17). This can be seen in the Figure 3-5 which plots the expected performance of four metalevel planning policies over the range of ϵ. Three of them achieve bounded optimal performance over specific ranges of ϵ, while the fourth, labeled INTERMEDIATE 2, is a sub-optimal policy dominated by the others. For this problem, it has no use, but is shown to illustrate that aside from the three dominating policies there exist other inferior policies. The existence of other complete policies also illustrates the danger with the approaches in Chapter 2 that assume the availability of an *a priori* set of decision procedures. The onus is on the designer to explicitly identify and include the best set of decision procedures. If INTERMEDIATE 2 were included but INTERMEDIATE 1 were somehow not identified as potential decision procedure, then the performance of the resulting system suffers. By allowing the computational actions

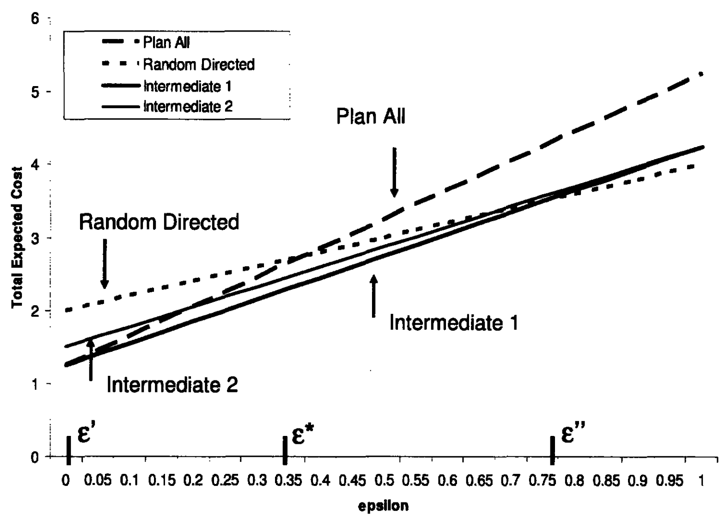

Figure 3-5: The performance of four different policies for the 4 arc planning problem as a function of ϵ. PLAN ALL, the dashed line is the best policy for $\epsilon = 0$ and RANDOM-DIRECTED is best when $\epsilon > 0.75$. The remaining lines represent other policies. The thicker line, indicated by INTERMEDIATE 1 is the performance of the optimal metalevel policy generated by solving the metalevel MDP for $0 < \epsilon < 0.75$. The thinner line INTERMEDIATE 2 is the performance of a sub-optimal policy dominated across the whole range of ϵ by the others.

to consist of individual sub-problems, the solution to the metalevel MDP automatically yields the dominating policies.

The optimal metalevel policy is clearly acting rationally in the extreme cases of zero and high computational costs. As confirmed by intuition, the PLAN ALL is the best policy for zero computational costs, and RANDOM-DIRECTED is the best policy for high computational costs. Recall that in Section 3.3, it was noted that one way of determining the strategy for intermediate values of ϵ was according to a simple switching policy which selects PLAN ALL or RANDOM-DIRECTED depending on which side of ϵ^* the true cost of computation lies. For intermediate values of ϵ, the optimal metalevel policy differs from both PLAN ALL and RANDOM-DIRECTED, and is shown in Figure 3-5 as INTERMEDIATE 1. This policy is generated automatically as a result of solving the metalevel MDP exactly. This made possible the discovery of a new closed-loop feedback controller that is optimal for the range of $0 < \epsilon < 0.75$. This new policy results in a performance improvement over the simple switching policy.

The improvement can be viewed from the perspective of the number of sub-problem computations saved. Let $\Delta = \min\{V_{\text{PLAN ALL}}, V_{random}\} - V^*$ be the difference between the minimum achieved by the PLAN ALL and RANDOM-DIRECTED strategies and the optimal metaplanning policy. When the cost of computing sub-problems is given by ϵ, then average number of sub-problems saved is given by $\frac{\Delta}{\epsilon}$. For example, the average number of sub-problems saved by INTERMEDIATE 1 over the switching policy at ϵ^* is exactly 1.

The three separate linear regions, representing the 3 separate dominating policies in Figure 3-5 are partitioned by ϵ' and ϵ''. For this simple problem, the regions are easy to delineate, since the first and last regions consist of the lines representing PLAN ALL and RANDOM-DIRECTED respectively. The boundaries for the middle region are given by $0 < \epsilon < 0.75$.

The expected cost function of an entire policy is given by the expected cost function of the initial "unknown-information" state, $\{?, ?, ?, ?\}$. By determining these functions for each policy, the boundaries can be found by identifying the intersections of these curves. In this particular case the expected cost function of the optimal policy in the range of $0 < \epsilon < 0.75$ is given by

$$E[\text{INTERMEDIATE 1}] = 3\epsilon + 1.25. \qquad (3.20)$$

The outer boundary value for epsilon is determined by the finding the intersection of lines representing the performance E[INTERMEDIATE 1] and E[RANDOM-DIRECTED], which happens to be 0.75. In fact, INTERMEDIATE 1 is optimal in the case where $\epsilon = 0$ as well.

3.6.5.2 Properties of Bounded Optimal Metalevel Policies

The two main properties of bounded optimal metalevel policies for the MDP formulation are given below.

- Tree-structured: The optimal metalevel policy is in the form of a tree of bounded degree since it has been assumed that there are a bounded number of outcomes that result from each computational action.

- Bounded depth: The optimal metalevel policy tree is of finite depth, bounded by the number of sub-problems.

These properties prove useful in the development of the heuristic approach in Chapter 4. Typically, MDP policies can be stored in the form of a lookup table, where for each state entry, there is a corresponding entry for the best action to be taken. While, in general, MDP policies are not tree-structured (because states can potentially be revisited), the optimal policies of the metalevel MDP are. The main reason for a tree structured policy is that each state can be visited at most once by any policy. Recall Figure 3-4, which notionally shows the entire metalevel MDP, consisting of all states and all actions. Also recall that a policy maps each state of the MDP to a single action. The figure shows four possible computational actions, $\{sp_1, sp_2, sp_3, sp_4\}$, at the root node, s, which represents the initial information state of having no knowledge of the problem instance being solved. Solving the MDP yields a policy that maps the best action to state s. Assume that the best action at s is to solve the first sub-problem, sp_1. Taking action sp_1 can yield two possible outcomes represented by s' and s''. Each of these states represents a distinct state of computation (e.g., the outcome of sp_1 is 0 or 2). Combined with the fact that sub-problem outcomes are assumed to be fixed once they have been solved, neither state s' nor s'' will ever need to be revisited by the remainder of the policy. This continues by induction and is true of every state in the policy. Because each state visited by the policy can only be visited once, the policy is a graph that contains no simple cycles [37] and is, by definition, a tree. The policy tree is of bounded degree since it is assumed that there are a finite number of outcomes for each sub-problem. The *policy tree*, the tree representing the metalevel MDP policy, for the problem represented in Figure 3-4, consists of a tree of degree two, since there are two possible outcomes for solving each sub-problem.

Finally, the policy tree is bounded in depth by the number of sub-problems. This is due the fact that each sub-problem need only be computed once, implying that each branch of the tree can, at most, consist of a computation sequence which solves every sub-problem.

3.6.6 Worst Case Time and Memory Analysis

Although policy iteration has the worst case running time of $O(|S|^3)$, which is polynomial in the size of the state space, as with most MDP problems the *curse of dimensionality* [5] is hard to avoid. The curse of dimensionality states simply that the size of the state space is exponential in the number of state features. For the discrete form of the metalevel planning problem the state features correspond exactly to the number of sub-problems, $|SP|$, such that the number of states is

$$|S| = O(M^{|SP|}), \tag{3.21}$$

where M is the largest number of possible discrete outcomes for the set of sub-problems, including ?, the unsolved sub-problem indicator.

The worst case storage requirements for the transition matrix in terms of a table of transition probabilities, is $O(|S|^2)$, which is also exponential in the size of the number of sub-problems. It is this limit that tends to be more restrictive. Even for sub-problems with binary outcomes where $|SP| = 16$ there are over 43 million states, easily exceeding the available memory on desktop PCs. Chapter 4 introduces a method that takes advantage of the tree-based structure of the metalevel planning polices to allow for the solution of larger problems that would be computationally intractable when solved as exact MDPs.

3.7 Discussion

The approach of framing the metalevel problem as a Markov decision process is applicable to many of the other problem formulations mentioned in Chapter 2. Einav and Fehling [17] presented a model that is very similar in nature to the one here, but differs in that they assume the existence of a set of individual uninterruptible solution procedures for every problem instance which will return a complete solution if one exists. In contrast, this model assumes that a sequence of sub-problem solutions must be combined to form a complete solution. Barnett [3], who also uses complete methods, discusses the importance of determining the conditions under which resources are spent on "method selection", where a method is a complete decision procedure. Considering the effects of method selection and execution can result in appreciable savings in cost. He presents some simple examples involving two independent methods which satisfy the same goal with different probabilities of success and expected costs. If a method is tried and is successful, then the other is not executed. He analyzes the utility of several strategies. One result of this analysis is that the order in which the methods are applied will affect the utility of the outcome just as the selection of sub-problems to solve will affect the utility of the final plan. He examines several cases which mirror those discussed in Subsection 3.3.

1. Expected cost of an ordering. (Equivalent to COMMIT, open-loop)

2. Coin flip. (Equivalent to RANDOM-DIRECTED strategy)

3. Best-Order Criterion (Equivalent to PLAN ALL and is the best order to activate the

methods when all information is known a priori. This is the same result as in [44]). From this calculation, the expected value of knowing the information *a priori* can be calculated and compared against the previous strategies.

4. Situation dependency. (Metalevel control policy) metaplanning)

Barnett's conclusion, just as that of this thesis, is that the "cleverness in selecting the taxonomy" is an important factor for efficiently selecting the control mechanism. That is, having situation-dependent data (e.g., feedback from sub-problem outcomes) gives an advantage over just knowledge of expectations. He states that being able to identify the current situation, just as the metalevel policy does, will aid in the decision as to which method to attempt, if the methods have different probabilities of success under different situations.

The identification of the specific problem instance is accomplished by the metalevel controller through being able to select computations at the sub-problem level and use it to determine the next action, an ability that is not available to complete decision procedure approaches. In order for the complete decision procedure approaches to be equivalent to this approach, they would need to specify *a priori* all of the possible algorithms expressible by the class of policies of the MDP formulation. This is clearly impractical since, the number of policies is exponential in the number of information states. One other infeasible approach would be to exhaustively store the optimal solution to every possible problem instantiation such that the "right" solution is invoked when that problem instance is encountered. One objection to doing so is the shear amount of computational effort required to solve every problem instance *a priori*.

Another objection is based on the informational assumptions for the metaplanning problem, which states that the agent does not know which problem instance it is given. Therefore, even given the set of optimal solutions, the agent still must solve the problem of efficiently identifying the current problem instance in order to apply the correct one.

The advantage of the MDP formulation is its ability to automatically discover and determine the appropriate policy to use for the situation. This was demonstrated by the set policies generated in Figure 3-5 showing that the generated metalevel controller behaves as the PLAN ALL strategy does when computational costs are zero, and converges to the RANDOM-DIRECTED strategy when computation costs are high. At intermediate costs, the generated policy performs better than both of them.

3.8 Chapter Summary

This chapter discussed the formulation of the metalevel planning problem as a sequential decision problem, where the objective is to determine a policy for rational decision making under limited computational resources. The metalevel planner generates a policy that is a function of the explicit outcomes of solving sub-problems, the problem decomposition, and the cost of time. These policies are equivalent to specially developed algorithms for a given distribution over a set of problem instances and cost of time function. The exact solution of the metalevel MDP is hindered by the problem of size of the state space explosion limiting the size of solvable problems. However, the benefit of the MDP framework is that once an optimal policy has been generated it can be repeatedly applied to the distribution of problem instances for which it was generated and will continue to be optimal. In practice, a domain expert will be needed to select the correct policy to execute by accurately identifying the cost of time for the situation at hand. As discussed previously, the optimal metalevel policy for a given distribution of problems is a function of the cost of time, when it can be represented explicitly under the separability assumption. The next chapter shows that the MDP formulation the metalevel planning problem of this chapter lends itself to an approximation algorithm using decision trees. Decision tree learning is used to reduce the state space of the metalevel planning problem, allowing for the solution of larger problems without sacrificing much in terms of metalevel controller performance.

Chapter 4

Decision Trees and Approximate Policy Iteration

In Chapter 3, the basic metalevel problem was presented and formulated as a discrete Markov Decision Process. As explained, the main deterrent to implementing the exact formulation is the size of the state space, which is an exponential function of the number of sub-problems. This exponential explosion is due to representing the state of computation of the metalevel MDP by the information state, which captures the known information about solved sub-problem outcomes. This representation allows for the metalevel controller to tightly control the next computational action given the results of previous computations. In this chapter, the same representation of the state is maintained, but the number of states considered in generating metalevel policies is drastically reduced through heuristic means via what is referred to as decision tree learning. Decision tree learning is embedded within approximate policy iteration to learn approximate metalevel policies that perform well without solving the metalevel MDP exactly. Thus, metalevel problems, whose exact solutions are computationally intractable, can be solved using the approach of approximate policy iteration with decision trees outlined in this chapter. This approach offers a variety of advantages over the exact MDP method, as well as some new challenges.

4.1 Policies as Decision Trees

In Chapter 3, it was shown that metalevel planning policies can be represented as trees that are finite with a depth no greater than the number of sub-problems. The value iteration approach to solving the metalevel MDP described in Chapter 3 amounts to searching through the set of all possible policies representable by the MDP formulation and selecting the one that results in minimum expected cost. In this chapter, rather than searching through the entire space of possible policies (difficult due to the sheer size of the search space) and generating a single optimal policy, a metaplanning policy is generated and improved upon over subsequent iterations. This process is similar in nature to policy iteration and can be thought of as a form of approximate policy iteration (API) [6]. Approximate policy iteration consists of a policy evaluation step, which learns a functional approximation to the true value function based on sampling trajectories for a "representative" set of states, generally determined through a combination of heuristics and simulation. This approximate policy is then improved upon in the policy improvement step. In this chapter, decision trees are the principle structures to be used both to represent the policy as well as to approximate the value function. The next subsection introduces decision trees and shows how they are generated.

4.1.1 Decision Tree Learning

Trees are natural structures that arise in a variety of AI applications. A tree is defined as an undirected graph with no simple cycles [37]. It is considered binary when there are two child nodes for every internal node. A decision tree is a tree structure, used either for classification or regression that is trained to learn how to classify, or predict, outputs given a set of input features. There are a variety of decision tree learning algorithms (see [36] for a survey). The algorithm used here follows closely from Breiman's Classification and Regression Trees (CART) algorithm [11]. In this thesis, decision trees are used for the purposes of regression over the value function, of the metalevel MDP given the set of feature values, which correspond to the costs of individual sub-problems. Decision trees will be used to learn which sub-problems are the best predictors of the total expected cost (including computation costs) of plans for a specific problem domain. Identifying these sub-problems for a particular metalevel environment will help to focus the search to generate good metalevel control policies without having to examine every state as in Chapter 3. The associated decision tree is referred to as a *regression tree.*

The following discussion of decision trees is based on [20] and presents the CART algorithm as used for regression. In regression trees, the feature values and response values are numerical values rather than categorical. Consider the case where one wants to learn to predict the output of a system given a set of input feature values. This is accomplished by training a decision tree on pairs of data consisting of observed features and the corresponding output of the system. Suppose that the training data consists of N entries, where each data entry consists of a pair $(\mathbf{x},y)$. The vector $\mathbf{x}$ consists of a vector of instantiated feature values of length p, and y corresponds to the value of the response of the system given $\mathbf{x}$ as the input. That is, for each $i \in 1...N$ the vector $x_i = (x_{i1}, ..., x_{ij}, ..., x_{ip})$, where each x_{ij} is the value for feature j in instance i.

For example, assume that in the system of interest, there is one feature, $p = 1$, which corresponds to the ambient temperature. Suppose the error in the weight reported by a certain scale is known to be sensitive to temperature and biased towards over-reporting the true weight. Also suppose that many weighings have been performed in the past with the same scale along with an accurate accounting of the error of the reported weight and the ambient temperature. The index i of the input feature x_i corresponds to the data point of the ith weighing, where x_i is the recorded temperature and y_i is the weighing error. Figure 4-1 shows an example of a collection of past data points in the form of input/output (temperature/weighing error) pairs.

Given the data, CART recursively makes binary splits in the feature space to partition the input into regions of similar values. Specifically, the decision tree splits the data into M regions $R_1...R_M$, where the response value in each region is modeled as a constant c_m. Letting $I(x \in R_i)$ be an indicator function that input vector x falls into R_i, the response function f of the decision tree can be expressed as:

$$f(x) = \sum_{m=1}^{M} c_m I(x \in R_m)$$

This yields the value of the response when x belongs to region R_m. The evaluation criterion of the best response is typically to minimize the sum of squares over the resulting split regions, such that $\sum(y_i - f(x_i))^2$ is minimized over the region i. The best choice of the

c_m's for this criterion is the average over all response values corresponding to an individual region such that

$$\hat{c}_m = \frac{1}{N} \sum_{i:x_i \in R_m} y_i.$$

What remains is to determine the partitions, R_i's, themselves.

The optimal binary partition in terms of minimizing the sum of squares is computationally infeasible in general [20], and most algorithms resort to a greedy partitioning scheme. The entire input space is initially assumed to be a single region. The algorithm then iteratively creates new regions by selecting an existing region and splitting it in two along a single dimension, that is, along one specific feature. The selected feature is called the *splitting variable* and the value on which to split is called the *splitting point.* Suppose that the splitting variable is selected to be feature j along with a split point s. As a result the data is split into two regions where:

$$R_1(j, s) = \{X \mid X_j \leq s\} \text{ and } R_2(j, s) = \{X \mid X_j > s\} \tag{4.1}$$

All data entries where the value of feature j is less than or equal to the split point are clustered into region R_1 and similarly for R_2. In order to choose the splitting variable and splitting point, the objective becomes:

$$\min_{j,s}[\min_{c_1} \sum_{x_i \in R_1(j,s)} (y_i - c_1)^2 + \min_{c_2} \sum_{x_i \in R_2(j,s)} (y_i - c_2)^2]. \tag{4.2}$$

The inner minimizations, given a splitting variable and split point, are the average responses over the respective regions. The outer minimization is performed by iterating over all possible splitting variables and split points and selecting the best one. Once the data is split into two parts, this procedure is repeated for each individual region. Figure 4-1 shows an example of a split created over the feature, x, representing the ambient temperature. The output values, y, are clearly distinguished, depending on where each x falls with respect to the splitting point. For this example, the decision tree for predicting y given x consists of a single splitting node at the root with the splitting condition of $x < 0.5$ and two leaf nodes corresponding to the prediction values of 1, when $x < 0.5$, and 3, when $x \geq 0.5$. That is, if the ambient temperature is less than fifty degrees, the expected weighing error is $+1$ pound, and if the temperature is greater than fifty degrees the predicted weighing error is

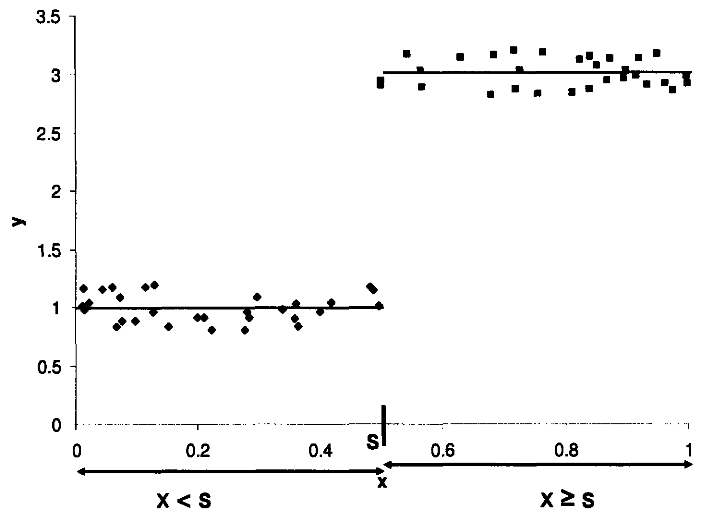

Figure 4-1: Example of a split in a decision tree, where S indicates the splitting point. The training data shows the weighing error for a variety of temperatures. The horizontal bars indicate the predicted weighing error on each side of the splitting point.

off by +3 pounds, assuming that the temperature scale ranges from zero to one hundred degrees. Any other splitting point in this case would lead to a larger sum of squares and poorer predictions. For instance, if the split occurred at $x = 0.25$, assuming that the ambient temperature is uniformly distributed over the interval of $[0, 1]$ for each weighing, the prediction would lose much of its accuracy for $x \geq 0.25$, since it includes a wider range of weighing errors than the best split.

For this simple example, a new data point x' would have its corresponding weighing error predicted by "dropping" it through the decision tree (i.e. evaluating all splitting conditions until a leaf node is reached). If weight of the object was recorded when the temperature was less than fifty degrees, the error is +1 pound and +3 pounds otherwise. Although this example includes only a single split, this same procedure applies in the natural way for larger decision trees. The decision tree learning algorithm, called *treeGen*(), is given in Algorithm 3.

The input to Algorithm 3 is a set of training data as in the temperature/weighing error pairs of the previous example and a parameter, *splitmin*, to indicate the number of training samples that must be at a decision tree node to warrant further splitting. The decision tree, given as T, is initialized as a single node and assigned a node index 1 for identification. This node is set to be the current node, represented by the variable ID. This

Algorithm 3: The decision tree learning algorithm *treeGen*()

input : x, a vector of instantiated feature values. y, a vector of corresponding observed responses. *splitmin*, the minimum number of observations needed for a split.

local : *numNodes* is the number of nodes in the decision tree. *ID* is the index of a node in the decision tree. x_{ID}, y_{ID} corresponds to the data set at a given node index. Initially, this data set is comprised of the full input. *size*() is a function that returns the number of elements in a vector. *findBestSplit*(), finds the splitting value resulting in the minimum sum of squares given a particular feature. *splitVar*, *splitVal*, and *splitErr* represent temporary splitting variables and values along with their resulting sum squared error. A "$*$" indicates the best of these values. *lChild* and *rChild* indicate the left and right child nodes of a split. *createChild*() creates a new node with an new index, and assigns to it an appropriate (according the to the best split) subset of the current data.

output: T, a decision tree consisting of a list of nodes. Each node has a corresponding splitting variable, splitting value, child pointers and a predicted output value.

```
begin
    $x_1 \leftarrow x$;
    $y_1 \leftarrow y$;
    $numNodes \leftarrow 2$;
    $ID \leftarrow 1$;
    while $ID < numNodes$ do
        $x' \leftarrow x_{ID}$;
        $y' \leftarrow y_{ID}$;
        $prediction \leftarrow E[y']$;
        if $size(y') > splitmin$ then
            $splitErr^* \leftarrow \infty$;
            foreach $splitVar$ do
                $[splitVal, splitErr] \leftarrow findBestSplit(splitVar, x', y')$;
                if $splitErr < splitErr^*$ then
                    $splitErr^* \leftarrow splitErr$;
                    $splitVal^* \leftarrow splitVal$;
                    $splitVar^* \leftarrow splitVar$;
                end
            end
            $lChild \leftarrow createChild((x'(i), y'(i)), i : x'(i, bestVar) < splitVal^*)$;
            $rChild \leftarrow createChild((x'(i), y'(i)), i : x'(i, bestVar) \geq splitVal^*)$;
            $T(ID) \leftarrow [splitVal^*, splitVar^*, lChild, rChild, prediction]$;
            $numNodes \leftarrow numNodes + 2$;
        end
        $ID \leftarrow ID + 1$;
    end
end
```

initial node holds the entire set of training data, since splits have not yet been performed. If the number of training samples within the current node exceeds *splitmin*, potential splits are considered. For the current node, each possible splitting variable is considered and the function *findBestSplit*() is called to perform the calculations described in Equation 4.2 to find the best splitting variable and value, *splitVar** and *splitVal** respectively. The data in the current node is split accordingly, by partitioning it based on the splitting variable and value, and passed on to each of its two child nodes. The child nodes, representing the results of the split, are added to the queue of tree nodes to be considered. The current node index is updated to the next unexpanded node, and the process is repeated until all nodes have been considered for splitting.

The art of building decision trees involves, among other things, determining the size of the tree, since a small tree will not capture the structural relationships that may be inherent in the data, but a large tree will not generalize well. The most common procedure is to grow the tree out to a large size. This allows the tree to take advantage of the potential utility of a combination of splits, when a single split may not appear to be useful on its own. This large tree is pruned according to a procedure known as *cost-complexity pruning*, (see [11]). Cost-complexity pruning proceeds by performing a tradeoff in the size of the decision tree against its predictive power. The original decision tree is sequentially pruned according to a parameter α, representing the reduction of prediction error per leaf, to produce a sequence of sub-trees. The performance of each sub-tree is determined on a test set or through cross-validation, and the best sub-tree is selected as the final decision tree. It will be shown that cost-complexity pruning is not used here, rather an MDP is formulated to prune the decision tree.

From the above discussion, it is clear that the decision tree algorithm is greedy in the sense of choosing a single variable and splitting point at each iteration to minimize immediate squared-error of splits. Global tree optimization for classification has been addressed by Bennett [4] but requires the user to specify the initial tree structure. It will be shown that the structure of the decision tree is part of optimization problem that must be solved since the structure determines the expected cost of computation for a metalevel policy.

Thus far, the discussion of decision trees has dealt with regression. Limited rationality has also been recognized in the case of classification problems. The goal of classification is typically to generate decision trees that minimize the cost of predictive error, but does so without considering the costs of performing the tests[1] involved in classification. In contrast, Ling et al. [31] consider the case where the objective function involves minimizing the sum of the test costs and the misclassification costs. A parallel to Ling's work can be drawn to the work in this thesis, namely, the test costs correspond to the costs involved in solving sub-problems for the metaplanning problem.

Turney [47] has developed a genetic algorithm based approach to generate decision trees for cost-sensitive classification. The cost of performing tests is, according to Turney [48], strikingly overlooked in machine learning literature. He concludes that tests should not be performed if their costs exceed the misclassification error costs.

Greiner et al. [19] also consider cost-sensitive "active classifiers". A passive classifier does not consider the possibility of performing additional tests to obtain missing information and active classifiers do. They present a probably-approximately-correct (PAC) algorithm that can make such decisions efficiently when the number of additional tests is limited.

Zubek and Dietterich [52] also consider the case of cost-sensitive learning problems and state that it "is interesting to note that the problem of learning good diagnostic policies is difficult only when the cost of testing is comparable to the cost of misclassification. If the tests are very cheap, compared to misclassification errors, then it pays to measure all of the attributes. If the tests are very expensive, then it pays to classify directly without measuring any attributes." This is echoed in the results for the example problems found in this thesis: when computational costs are low, there is virtually no penalty for performing additional computations, but when they are high, a greedy strategy (equivalent to directly classifying without testing) for planning will dominate. Zubek and Dietterich also employ an MDP approach to determine a testing policy, which is similar to the MDP approach discussed in Chapter 3.

The aspect of the metaplanning problem that does not appear in cost-sensitive classification problems (besides regression) is that of generating a feasible plan. That is, once the classification problem is solved, the decision tree halts. This is not viable in the case where the decision tree is used for planning, due to the constraint that plans must be feasibly executable. For instance, when test costs are expensive in the cost-sensitive classification domain, the best action is to "classify without testing". In metalevel planning, this is not

[1]Test costs refer to the costs involved in determining the outcome of a splitting variable. For instance, if a split involved the outcome of an x-ray scan, the test cost would be the cost of performing the x-ray.

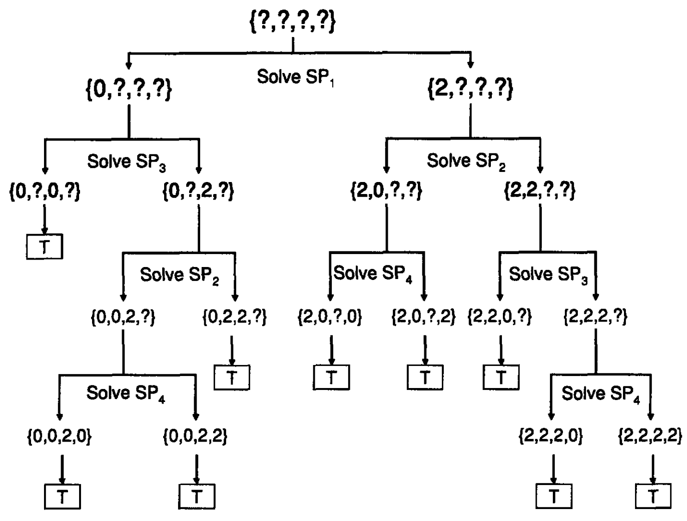

Figure 4-2: An optimal policy tree for the 4 arc problem represented as a policy tree.

an option, since the act of classifying is equivalent to executing a plan. However, without solving any sub-problems, a plan cannot be generated at all. As will be seen, one of the shortcomings of using decision trees for metalevel planning is that they may not always yield feasibly executable plans. A methodology for plan completion is proposed later to handle this issue.

4.2 Decision Trees For Metaplanning

Figure 4-2 shows a tree representation of the optimal policy labeled INTERMEDIATE 1 for the 4 arc planning problem presented in Chapter 3. Recall that a policy is a mapping of states to actions. The bracketed sets (tree nodes) are the states of the metalevel MDP. For instance the root node of the tree consist of a set of all ?'s, representing the initial state where the agent has no knowledge of the problem instance, that is, no knowledge of the sub-problem solution outcomes. The computational action prescribed by the metalevel controller (i.e., which sub-problem to solve) is given under each node. Each computational action results in a transition from the current information state represented by a ? to another information state according to the state transition function. The first action of the policy at the root node is to solve sub-problem 1. Going from the root node to the next level, there is a change in the first element of the bracketed set. The act of solving sub-problem 1 results either in the state where the outcome (cost) is a 0 or a 2 as indicated by

$\{0, ?, ?, ?\}$ and $\{2, ?, ?, ?\}$ respectively. This tree-based policy consists of 18 states including the terminal state indicated by in the figure by T. Recall that the terminal state s_{term} is used to represent the state of having solved for and executed a feasibly executable plan. For the 4 arc planning problem each sub-problem outcome can take on 3 possible values, $\{0, 2, ?\}$, yielding a total of 82 possible states including the terminal state. The methods in Chapter 3 returned an optimal action for *every* state, many of which are not encountered when executing the given policy. In contrast, as shown in Figure 4-2 only 18 of the 82 states are ever visited. The tree based representation of the optimal policy is clearly economical in terms of memory, and will be shown to allow for a much more efficient generation of the approximate metalevel MDP policy.

4.2.1 Learning the Decision Tree Policy

The exact solution methods of Chapter 3 are abandoned here in favor of approximate methods, in particular approximate policy iteration. The decision tree representation allows for the metalevel policy to be generated through decision tree learning, thereby bypassing the exact computation of the optimal action for every state. As seen in Figure 4-2, the optimal metalevel policy does not necessarily visit all states. The decision tree algorithm discussed in the previous section is embedded in approximate policy iteration to learn the important states for a problem domain and serves two purposes. Decision trees are generated and used as approximations to the value function of the metalevel MDP and to serve as a basis for the creation of a metalevel control policy. Approximate policy iteration consists of three main steps:

1. Initial policy generation.
2. Policy evaluation.
3. Policy improvement.

In **initial policy generation**, the initial policy, $\bar{\pi}$, is typically guessed or set randomly. Here, the decision tree generating algorithm, *treeGen*(), is used in the first step of approximate policy iteration to generate a decision tree that is subsequently pruned to serve as the initial metalevel control policy. The *treeGen*() algorithm is used subsequently in the policy improvement step to generate new policies as well. The advantage of using *treeGen*(), rather than guessing or setting the policy randomly, is that the resulting decision tree will help to identify important information states of the metaplanning MDP, thus pruning away large portions of the search space. The decision tree algorithm uses the expected cost of the optimal base-level solution to create an initial decision tree which clusters together information states whose plans yield similar costs when executed.

Using decision trees to generate policies can partly be seen as a form of value function approximation in the terminology used in approximate policy iteration. In approximate policy iteration, a function approximation of the current value function is computed from a set of basis functions, ϕ_i. For instance, in a linear approximation architecture [8], the estimate of the value function, in approximate policy iteration, is computed as a weighted sum of these basis functions,

$$\tilde{V}(s,r) = \sum_{i=0}^{K} r_k \phi_k(s), \tag{4.3}$$

where $\tilde{V}(s,r)$ is the approximation to the value function of the current policy at state s, and r_k is the weighting factor for the kth basis function ϕ_k. The values of r_k can be determined by solving a least-squares minimization problem to fit the weights using sampled data given by:

$$\min_r \sum_{s \in \tilde{S}} \sum_{j=1}^{M(s)} (\tilde{V}(s,r) - v(s,j))^2, \tag{4.4}$$

where $\tilde{S}$ corresponds to the set of sampled states, where each state, s is sampled $M(s)$ times, and the j sample of the value function of state s is given by $v(s,j)$. From Equation 4.4, the optimal r can be determined.

The approximate value function using decision trees can be seen as employing basis functions that consist of constant functions over specific regions of the state space. There is no direct analogy to the weighting parameters, r, in decision tree learning. Instead, the splitting variable and splitting criteria are used to define regions to be approximated by each constant function. The duty of finding the optimal r values of Equation 4.4 for linear value function approximation is replaced by splits determined by Equation 4.2 for decision tree based value function approximation.

The training data for the decision tree version of approximate policy iteration consists of samples of problem instances drawn from the metalevel environment. That is, many problem instances are drawn according to the appropriate probability distribution and solved

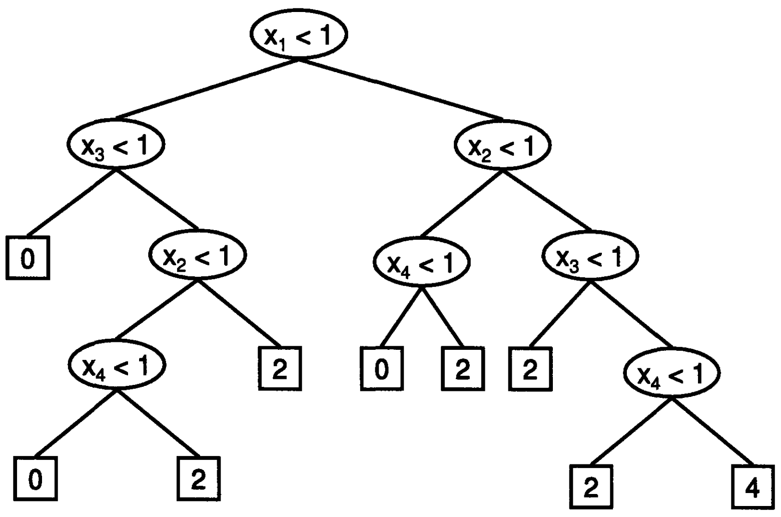

Figure 4-3: Decision tree generated for the 4 arc problem.

optimally assuming that there is no computational cost. For each instance of the training sample, the training features are the tuple of sub-problem outcomes (costs) and the observation is the accompanying optimal base-level plan cost. The optimal cost of each sample is effectively the "label" in the language of supervised learning. From the samples, the decision tree is trained to accurately predict the cost of the optimal solution.

Recall, from Subsection 4.1.1, that in order to determine the prediction for a particular problem instance, the instance must be "dropped" through the decision tree (by performing the tests recommended by the decision tree until a leaf node is reached). In the context of metaplanning, the evaluation of a splitting condition has special meaning and is equivalent to performing a computational action. Essentially, a split in a decision tree is equivalent to the branching in the policy tree in Figure 4-2, where at each level a sub-problem is solved (a computational action) so that its outcome can be used to determine which branch of the policy tree to follow next.

A decision tree can be directly interpreted as a policy tree. A policy tree consists of a set of nodes representing the information state, a computational action for each state, and branches representing the possible set of outcomes that can result from performing the computational action. A decision tree, consists of nodes, corresponding to splits in the data, and branches corresponding to the results of a splitting action. Each split in the decision tree can be interpreted as solving a sub-problem in the policy tree, where the splitting variable corresponds to the sub-problem to solve, and the splitting criterion corresponds to a condition on the outcome of solving a sub-problem.

Figure 4-3 is a decision tree generated from data for the 4 arc planning problem. Each node in the decision tree contains a splitting condition. The variable over which to split is given by x_i, where i corresponds to the sub-problem index. If the splitting criterion is true after solving a sub-problem, then the left branch is taken, otherwise the right branch is taken.

There is slight difference in branching syntax found in decision trees, as opposed to policy trees, which allow for them to deal with continuous data. That is, in the MDP policy, solving a sub-problem results in a probabilistic but discrete state transition (e.g., solving sp_1 results either in an information state where the cost of sp_1 is a 0 or a 2). In the decision tree, a split is expressed as a continuous interval defined by an inequality. However, the decision tree can easily accommodate discrete outcomes. An equivalent split for outcomes of 0 or 2 for sp_1 in Figure 4-2 is represented by $x_1 < 1$ in Figure 4-3. Considering the binary outcomes possible for sp_1 in the 4 arc problem, the split indicates that the left branch corresponds to states where sp_1 has a cost less than 1 (i.e., the information state where sp_1 has an outcome of 0) and the right branch corresponds to sp_1 having a cost greater than 1 (i.e., the information state where sp_1 has an outcome of 2).

The corresponding MDP information state for each node in the decision tree can easily be determined by tracing it up to the root node while keeping track of how each split was evaluated. For instance, in Figure 4-3, the root node corresponds to the initial information state of $\{?, ?, ?, ?\}$, while the leaf node furthest to the right corresponds to the conditions where $(x_1 > 1) \wedge (x_2 > 1) \wedge (x_3 > 1) \wedge (x_4 > 1)$ (for this problem it represents the information state given by $\{2, 2, 2, 2\}$). The leaf nodes of the decision tree give the predicted outputs (in this case the prediction of the cost of the optimal plan) given the evaluation of the splitting conditions. The decision tree in Figure 4-3 is isomorphic to the optimal policy tree for INTERMEDIATE 1 given in Figure 4-2.

Thus far, no mention has been made of the cost of computation in this process. The initial decision tree is generated assuming that computational costs are zero. As in Chapter 3, it is assumed that additive time cost separability holds. This allows the cost of computation can be added into the decision tree as shown in Figure 4-4. The composite decision tree is the sum of the trees representing the expected computation cost for each node, and the expected base-level execution costs. In Figure 4-4, it is assumed that each branch can

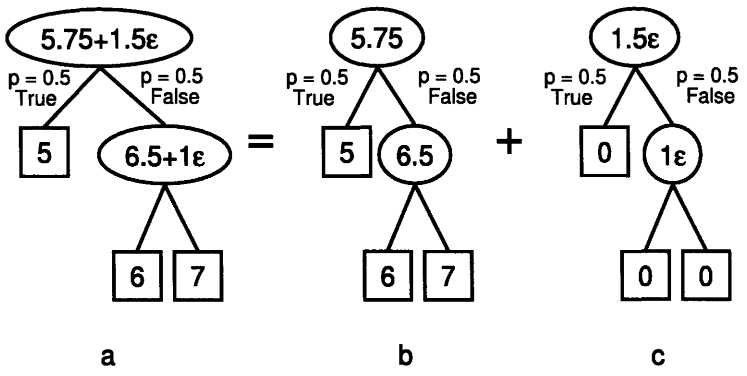

Figure 4-4: Time cost separability allows for the expected cost of decision tree policies to be expressed as the sum of base-level action costs and computational costs. a) Shows the composite tree, which is the result of summing b) the tree representing expected execution costs with c) the tree representing the computational costs involved.

occur with equal probability and that each computational action incurs a cost of ϵ. The squares for Figures 4-4b and 4-4c represent the terminal execution and computation costs respectively. The value shown in each node is the expected cost-to-go of the corresponding information state (i.e., the expected cost of following the policy to completion starting at the node). The value shown at the root node of Figure 4-4a is the expected cost for the entire metalevel control policy represented by the decision tree. The initial decision tree is called the zero computational cost tree since it is trained with data gathered by setting ϵ to zero. In terms of approximate policy iteration, the zero computational cost tree accurately approximates the value function of the optimal metalevel policy when the true cost of computation is zero. When true computation costs are non-zero, they are added to the zero computational cost tree followed potentially by a pruning step to yield a metalevel planning policy that does account for the cost of computation.

As discussed in Chapter 3, the policies for high computational costs versus low computational cost will differ greatly. It is expected that the decision tree for the case of high computational costs will result in the solution of far fewer sub-problems as compared to the case of the tree for low computational costs. As with CART, an initial decision tree is generated to full depth, and pruned to account for the cost of computation (longer tree branches incur higher computational costs). The pruning step proceeds by treating the state space spanned by the decision tree as an MDP. A branch is pruned when the expected benefit of further computation recommended by the branch outweighs its cost. The pruned tree constitutes an approximate policy. For instance, suppose that the full decision tree when $\epsilon = 0$ is given by Figure 4-4a. When the cost of computation rises high enough, the pruning step might eliminate the sub-tree of the right branch resulting in a shorter tree which solves less sub-problems.

The second step of approximate policy iteration is **policy evaluation**. The evaluation of the performance of the approximate policy is determined through simulation to obtain samples, $v(s,i)$, for Equation 4.4. From this r can be determined so that the approximate value function, $\tilde{V}$ can be determined from Equation 4.3.

The last step of the approximate policy iteration cycle, **policy improvement**, is generically performed for each state [8], to improve upon the current policy based on its approximate value function, $\tilde{V}$. That is, given the approximate value function determined from the previous step, each state is examined to determine whether it is possible to improve upon its currently selected action. If so, the best action is updated for the corresponding state. This is accomplished by solving the Bellman equation for each state,

$$\bar{\pi}'(s) = \arg\min_{a \in A}(c(s,a) + \gamma \sum_{s' \in S} T(s,a,s')\tilde{V}(s')), \tag{4.5}$$

where $\bar{\pi}'(s)$ is the best action of the improved policy for state s and $\tilde{V}$ is the approximate value function of the current policy $\bar{\pi}$, determined from Equation 4.3. Evaluating and updating all state actions results in a new and hopefully improved policy. In the decision tree approach, rather than improving the policy on a per state basis, an entirely new decision tree policy is learned based on data collected from Monte Carlo simulations of executing the current decision tree policy. As in the generation of the initial policy, the decision tree uses the expected value (cost) of the current policy to guide it in the generation of the next policy. The complete Decision Tree Metaplanning (DTMP) algorithm is presented next.

4.3 DTMP

The Decision Tree Metaplanning (DTMP) algorithm consists of the three main steps of approximate policy iteration as discussed above and is summarized in Algorithm 4.

1. Initial policy generation is performed in three steps in Algorithm 4. The first step is to generate an initial decision tree from training data by calling the *treeGen*()

algorithm as described by Algorithm 3. The decision tree is augmented with the cost of computation and, when necessary, with supplemental actions. The resulting decision tree is treated as a restricted version of the exact metaplanning MDP. The 5 elements, $\{S, A, R, T, \gamma\}$, of the restricted MDP, *mdp*, are determined by *getStats*(). This MDP is solved for an optimal policy, $\bar{\pi}$, to serve as the initial metalevel policy.

2. The policy evaluation step consists of simulating the performance of the current policy, $\bar{\pi}$, and recording its expected total cost for each sample problem instance. Data from the simulation is gathered in terms of input/output pairs, $(\mathbf{x}', y')$, for the next step.

3. Policy improvement, the last step of approximate policy iteration, proceeds similarly to the process of generating the initial policy. The only difference is that the cost of computation and the performance of the current policy is properly reflected in the training data. In this step, a new decision tree is generated based on the performance of the current policy, $\bar{\pi}$, on the set of samples gathered during policy evaluation. The resulting decision tree leads to the generation of the next policy in the same manner as the initial decision tree.

Inputs to the DTMP consist of the training data, $(\mathbf{x}, y)$, and the cost of computation ϵ, where $\mathbf{x}$ is a matrix of the instantiated costs of the sub-problems and y is the accompanying optimal base-level solution which does not yet account for computation costs. DTMP begins by generating the initial decision tree, *treeInit*, by the calling the *treeGen*() algorithm of Section 4.1.1. Recall that the algorithm is initialized with data collected from having solved many problem instances to completion. The *treeGen*() algorithm is the main mechanism for state space reduction in DTMP, which relies on it to generate decision trees from which metalevel controllers are constructed.

The *treeGen*() algorithm can be seen to induce a conditional ordering constraint on computational actions which prunes away large portions of the state space. This pruning effect can be understood by viewing the *treeGen*() algorithm as a heuristic search algorithm like A*, where the heuristic function, in this case, is the value function of the current policy.

The value function is used to guide the search through the state space of the metalevel MDP. In additive time-separable problems, when computational costs are zero, the optimal policy is known to be PLAN ALL (i.e., find the optimal base-level plan). Therefore, the initial policy in DTMP is generated using this knowledge, which is why the initial input

Algorithm 4: Decision Tree Metaplanner

```
input  : x, a matrix of instantiated sub-problem outcomes. $y$, a vector where
         each row is the optimal solution cost for the corresponding row in x.
         $\epsilon$, the cost of computation. N, number of iterations to run the
         algorithm.
local  : $V$, the expected value of the current metaplanning policy, $\bar{\pi}$. $\hat{V}^*$, the
         expected value of the best policy. iter, an index variable.
output: $\bar{\pi}^*$ the best metaplanning policy found thus far.
$\hat{V}^* \leftarrow \infty$;
Generate Initial Decision Tree;
$treeInit \leftarrow treeGen(\mathbf{x}, y)$;
$mdp \leftarrow getStats(treeInit)$;
for $iter < N$ do
    Use Decision and MDP Pruning to Generate Policy;
    $\bar{\pi} \leftarrow valueIteration(mdp, \epsilon)$;
    Evaluate Decision Tree Policy;
    $[V, \mathbf{x}', y'] \leftarrow simulatePolicy(\bar{\pi})$;
    if $V < \hat{V}^*$ then
        $\bar{\pi}^* \leftarrow \bar{\pi}$;
        $\hat{V}^* \leftarrow V$;
    end
    Generate New Decision Tree to Improve Policy;
    $tree \leftarrow treeGen(\mathbf{x}', y')$;
    $mdp \leftarrow getStats(tree)$;
end
```

to *treeGen*() is said to consist of the optimal cost for each problem instance. Using the optimal cost of the solution to each problem instance can be viewed as using an optimal heuristic function to guide the search. The heuristic is optimal since it accurately reflects the cost-to-go function for each state when computational costs are zero.

When computational costs are non-zero, the optimal metalevel planning costs are not available, and the performance of an approximate policy is used instead. The approximate policy for non-zero computational costs is built upon the zero computational cost policy, and is always an overestimate of the true metalevel planning cost. Although the heuristic estimate determined in this fashion is no longer optimal, it can nevertheless lead to good metalevel control policies.

The splits performed by the *treeGen*() algorithm can be seen in the terminology of heuristic search as being greedy with respect to the heuristic function. Referring to Algorithm 3, the *treeGen*() algorithm essentially examines all possible actions out of the initial state and selects the best one. This is similar to the A* algorithm selecting which node to expand. The best action (split) in this case, is the computational action which leads to the greatest distinction in the estimated value function. In other words, the best split is, among all sub-problems that can be solved, the one that yields the best prediction of the cost of metaplanning given its outcome (e.g., knowing the outcome of a certain sub-problem reveals a lot about the problem instance). In structured problem domains, certain sub-problem will have higher utility than others. The decision tree learns from data to identify and exploit this structure in the problem domain by identifying the important sub-problems. These highly predictive sub-problems become even more useful when they are in problem domains where the plan fragments they produce are also integral parts of the plans to be executed.

Each time a split occurs in the decision tree, it effectively focuses the search of the metalevel control policy to the corresponding subset of the state space consistent with the split. The result is that large portions of the original metaplanning MDP which do not satisfy these constraints are pruned away, leaving a significantly reduced state space[2]. The *treeGen*() algorithm is also used in the plan improvement step of DTMP.

[2]For instance, the original MDP state space for the 4 arc problem consists of 4 subtrees (one for each possible computational action) extending from the root node. The decision tree selects the best action, solve sp_1, and prunes away the subtrees corresponding to the other actions, yielding the tree shown in Figure 4-3.

The remainder of the DTMP algorithm deals with iteratively improving the decision tree when non-zero computational costs must be incorporated into the tree generation process. The zero computational cost tree, *treeInit*, in Algorithm 4, represents a restricted portion of the complete plan space of the metaplanning MDP known here as the *induced*-MDP.

After the initial decision tree, *treeInit*, is learned, the next step consists of calling a function called, *getStats*(), whose purpose is to add an alternative action to each state of the decision tree and determine the rewards for those actions. Disregarding the *getStats*() function for the moment, the induced-MDP is equivalent to *treeInit*. The induced-MDP is an approximate representation of important aspects of the full MDP found in Chapter 3, with a massive reduction in the size of the state and action space, which is now expressed compactly by the decision tree. The state space is reduced due to the decision tree recognizing important sub-problems and the structural relationships among them. The action space is reduced since the metalevel planning problem no longer needs to select among all possible sub-problems to solve at each information state. Due to the decision tree, only two possibilities remain: compute the sub-problem suggested by the decision tree, or execute the current plan if it is feasibly executable.

The induced-MDP is solved by incorporating the cost of computation, denoted by ϵ, via the function *valueIteration*(). The function *valueIteration*() uses value iteration, from Chapter 3, to determine an initial policy, $\bar{\pi}$, for the induced-MDP. Introducing the cost of computation acts to prune *treeInit*, such that a branch is pruned when the expected benefit of further planning is outweighed by the expected cost of computation. For each state in *treeInit*, the policy, $\bar{\pi}$, dictates whether the metaplanner should continue planning or stop. Next, the policy evaluation step is performed by calling the function *simulatePolicy*() to determine the performance of $\bar{\pi}$ by evaluating its performance though simulation. The expected performance of the policy is given by V, along with a new set of data $(\mathbf{x'}, y')$ as a result of simulating $\bar{\pi}$. The performance of the current policy is compared against that of the current best policy, $\bar{\pi}^*$. If it is better, the best performance, $\hat{V}^*$, and best policy, $\bar{\pi}^*$, are updated. Since the objective is to find the minimum cost policy, the value of $\hat{V}^*$ is initialized to positive infinity.

This is followed by the policy improvement step, the goal of which is to use the new set of data, $(\mathbf{x}', y')$, to generate an improved policy. The new set of data accumulated as a result of testing the current policy on simulated problem instances has the advantage of

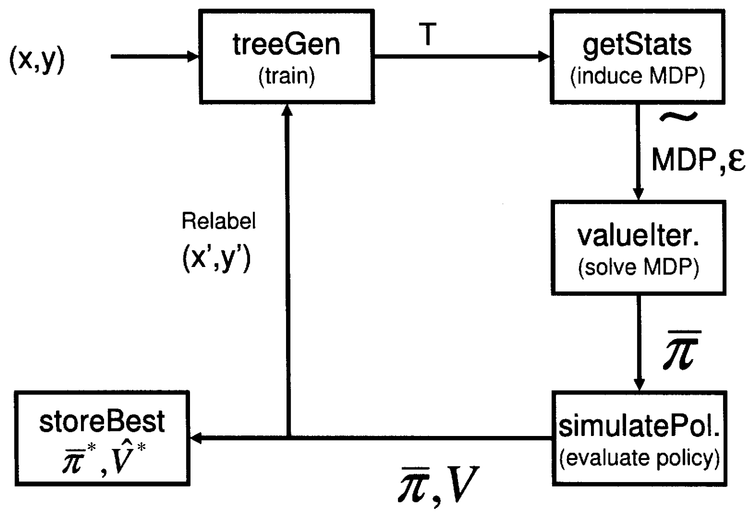

Figure 4-5: DTMP flowchart. The initial training data, $(\mathbf{x}, y)$, is fed into *treeGen*(), producing a decision tree, T. The cost of computation, ϵ, and greedy completion actions are added to each node of the tree in *getStats*(), which produces the induced-MDP, $\widetilde{MDP}$. The induced-MDP is solved for an approximate policy $\bar{\pi}$ using value iteration. The performance, V, of the approximate policy is evaluated in *simulatePolicy*(), producing new data, $(\mathbf{x}', y')$, with labels which include the cost of computation. The best policy $\bar{\pi}^*$ and its value $\hat{V}^*$ are stored. The new data is fed back into *treeGen*() to generate improved policies.

having the cost of computation for the policy, $\bar{\pi}$, explicitly incorporated into the y' vector.

Rather than solving the Bellman equation, Equation 4.5, for each state, the *treeGen*() algorithm is invoked on the new set of data, which now properly reflects the performance of the current policy, including the cost of computation. This new data is equivalent to the approximate value function of the current policy, $\tilde{V}$, of Equation 4.5. As per approximate policy iteration, the value function of the current policy is used to determine a new policy. Again, as in the generation of the initial policy, *treeGen*() uses the approximate value function as a heuristic to generate splits in the data to create a new decision tree policy for the next iteration of DTMP.

Once the new decision tree is generated, the entire process is repeated for a predefined number of steps, where the best policy is reported. A flowchart of the DTMP algorithm is shown in Figure 4-5. DTMP, unlike exact policy iteration, does not have guaranteed finite convergence, since both the policy and the value function are greedily approximated. More about this is found in Section 4.4.

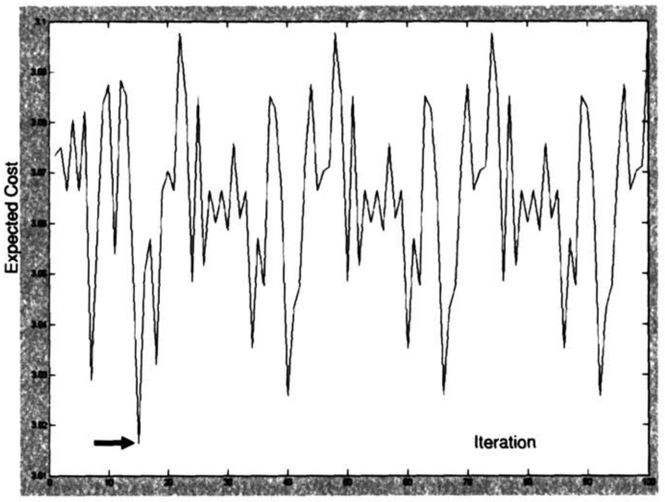

Figure 4-6: A plot of the expected performance of a sequence of oscillatory policies generated by DTMP. The number of iterations of DTMP are given by the x-axis, while the expected cost of the policy is given by the y-axis. The minimum cost policy is achieved at iteration 15 indicated by the arrow. Subsequent iterations lead to periodic oscillations which achieve a local minimum.

4.4 DTMP Issues

Repeated application of DTMP can result in oscillatory behavior of the policies. Policy oscillation and divergence are well known problems with approximate policy iteration [8]. Figure 4-6 shows an example of the oscillatory nature of DTMP algorithm, where the x-axis indicates the number of iterations the algorithm was repeated, and the y-axis is the total expected cost including that of computation for each policy. It should be noted that oscillatory behavior does not occur in all problems. Within a particular metalevel environment, oscillations can occur with some values of ϵ while not with others. Figure 4-6 also shows that iterating does have utility, since the minimum cost policy is not achieved until iteration 15, though the improvement in cost is slight. For the experimental results presented in Chapter 5, the number of iterations was artificially limited to 10, storing the policy with the best result.

The *getStats*() function, mentioned briefly, adds an additional action to each state of the current decision tree that attempts to complete the plan in a greedy manner. This is necessary, due to situations where the decision tree does not identify enough sub-problems to solve in order to generate a feasibly executable plan. That is, there are situations where a problem instance drops down to a leaf node which has not solved enough sub-problems for the master level to generate a complete solution. Under these circumstances, a heuristic plan completion algorithm that attempts to form a feasibly executable plan as quickly as possible is called, possibly reducing the performance of DTMP. Subsection 4.4.1, discusses some reasons for this phenomenon and presents an alternative way of using decision trees to potentially guarantee that the metalevel policy solves enough sub-problems to generate feasibly executable plans. The *getStats*() function gathers the information necessary (transitions and rewards) for these plan completing actions to be included in the description of the induced-MDP through simulation.

In this thesis, the main form of the plan completion algorithm is a greedy planner. For instance, the `RANDOM-DIRECTED` strategy can be used since it is computationally inexpensive and, in most cases, can find feasibly executable plans. The path completion algorithm for the 4 arc problem of Chapter 3 consists of running the `RANDOM-DIRECTED` planner from the start location to generate a complete path to the goal. Sub-problems on this path that have not been solved are solved, and the path executed. However, doing so might lead to paths that do not take advantage of the sub-problems that have already been solved while executing the decision tree policy. In the worst case, the `RANDOM-DIRECTED` planner might select a path consisting entirely of unsolved sub-problems, effectively disregarding the efforts of the metalevel. There are ways of modifying the planner to bias it towards paths that contain sub-problems that have already been solved.

Another strategy for plan completion is to store a list of candidate plans, learned from the data, at the leaf nodes. Once a leaf node is reached, the metalevel might search through the list of candidate plans to select the best one to complete the plan. This is a simplified version of the suggestion found later to use two types of decision trees, one to globally classify the problem instance, and another to generate the remainder of the plan. Since the set of candidate plans collected at the leaves are learned from data during the execution of the metalevel controller on real problems, there may be situations where none of the candidate plans learned thus far can be appropriately applied. In these instances, the only alternative is to run a plan completion algorithm to generate an entirely new feasibly executable plan. However, once generated, this new plan can be added to the set of candidate plans for future use.

X1	X2	X3	Y
1	1	0	2
1	0	1	4

a

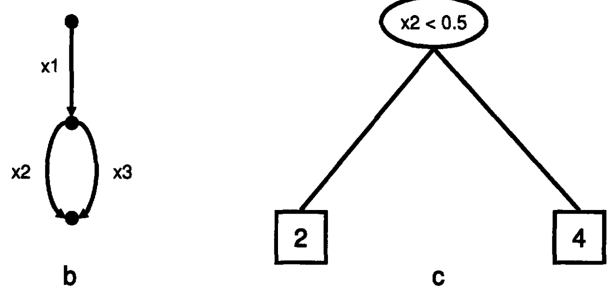

Figure 4-7: Example of an insufficient number of sub-problems for generating a feasibly executable plan. The data for generating the decision tree is given by a) and consists of three features x_1, x_2 and x_3 corresponding to the arc of the graph problem shown in b). The cost of x_1 is known to be 1 while the cost of the other arcs can vary. The resulting decision tree is given in c) and exhibits the problem of insufficient depth.

4.4.1 Insufficient Tree Depth

The goal of the decision tree metaplanning algorithm is to generate the optimal metalevel policy directly from data, thereby avoiding the costly state space enumeration needed for exact MDP solution methods. As stated previously, decision trees are intended to be employed as classifiers and predictors. However, since the goal is to use a decision tree directly as a metalevel controller, which supplies sub-problem solutions to the master level in order to generate high quality plans, their naive use can lead to decision trees that can accurately *predict* the costs of metalevel planning, without actually generating the plan.

As has been previously discussed, one problem with the decision tree method is that often the tree depth is insufficient in generating feasibly executable plans. The decision tree uses sub-problem outcomes to predict the cost of the optimal plan, but does not necessarily provide enough sub-problems to form the plan itself.

This behavior can be seen in Figure 4-7. The underlying graph structure is composed of three sub-problems, each capable of taking on binary values. Suppose that the decision tree algorithm is asked to generate a decision tree for metalevel planning given the data in the table in Figure 4-7a. Sub-problem x_1 should belong to the set of sub-problems to be solved, since it must be solved in order to generate any feasibly executable plan. However, since the decision tree tries to predict output based on data, it is sufficient to choose either sub-problem x_2 or x_3 as the splitting variable (in this case x_2) to perfectly predict the outcome. Although x_1 is an important sub-problem, its predictive utility is low and therefore disregarded by the decision tree algorithm. It is desirable to somehow emphasize the utility of sub-problem x_1 to the decision tree algorithm so that it is included in the decision tree.

4.4.2 Augmenting Sample Data

The decision tree algorithm can be made to include sub-problems that have weak predictive powers, but are important for generating feasibly executable plans. This is accomplished through augmenting the data used to generate the decision tree such that the sub-problem in question appears to have predictive power. The easiest way to do this is to generate a duplicate data set of the original, where the output column Y is set to some insignificant value (in this case 0), and the features are modified such that all subsets of the significant features are generated.

This is more easily seen through an example. Consider the problem corresponding to Figure 4-7. Suppose that for the first line of data, where sub-problems x_1 and x_2 are the sub-problems forming the plan with optimal cost equal to 2, two additional lines of data are introduced. The first includes the true value of x_1 and the alternate value of x_2 (assuming binary sub-problem outcomes) along with a predicted cost of zero. This signifies that when sub-problem x_1 takes on a significant value but x_2 does not, the result is insignificant. The second additional line attests to just the opposite, where x_2 is significant but x_1 is not. Both of these together are examples indicating that neither sub-problem alone is enough to produce a significant result. They must both be solved and placed in the decision tree. The same is repeated for the second line of the original data set for sub-problems x_1 and x_3. The augmented data and the resulting decision tree, which now includes sub-problem x_1, are shown in Figure 4-8

The augmented decision tree, can be seen as a list of feasible paths. For instance, in the example in Figure 4-8, the two optimal paths consist of sub-problems x_1 and x_2 or x_1 and x_3. Rather than storing these paths in a list, the sub-problems which constitute each path

X1	X2	X3	Y
1	1	0	2
1	0	1	4
1	0	0	0
0	1	0	0
1	0	0	0
0	0	1	0

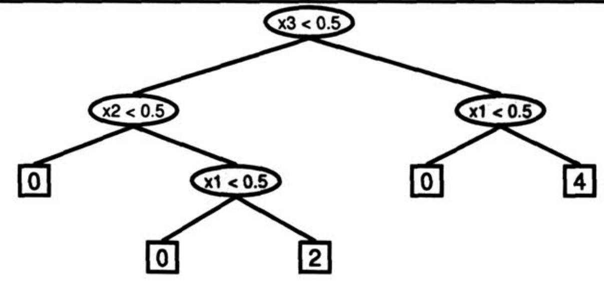

Figure 4-8: The original data in Figure 4-7a) is augmented resulting in a new decision tree which generates feasibly executable plans.

are organized in the form of a decision tree, where each branch corresponds to a feasible path.

Considering only binary outcomes, it is easy to see that the amount of data augmentation is exponential in the plan length for each line of data in the original data set. That is, if the optimal plan consists of three sub-problems, then the maximum number of augmented data samples that need to be added will be 2^3. When outcomes are not binary, then the number of augmented data lines is still exponential, but the base of the exponential is no longer 2.

One can easily imagine carrying this to the logical conclusion, where each line of the original data set is augmented. In this case, the problem of overfitting can become a concern. One possible way to avoid overfitting the augmented data tree is to apply it to a subset of the data rather than the entire set. The job now becomes one of identifying the subsets to which this process should be applied. This is accomplished in a two-step process, where the regression tree is first generated as usual. As discussed above, the original decision tree is a suitable candidate for clustering together like problem instances. The sample instances are then "dropped" down through the decision tree until they are "classified" at a leaf.

For the set of samples at each leaf, a miniature augmented decision tree (mini-tree) is generated. By doing so, the regression tree serves to funnel cases with similar optimal solutions to the same leaf node, and the mini-trees are used to complete the generation of sub-problems for the master level to form feasibly executable paths.

The joining of regression and mini-trees allows for an added level of control as to the amount of "classification" to perform before trying to identify sub-problems for feasible path generation. If the depth of the regression tree is zero, then the algorithm focuses entirely on feasible plan generation, perhaps leading to sub-optimal plans due the excessive splitting that may occur. On the other hand, if the regression tree is full depth, then the algorithm focuses entirely on predicting plan costs, risking the generation of incomplete plans. From this point of view, there is a balance that must be struck between predictive sub-problems, and plan completing sub-problems that should be a subject for future research.

4.5 Automatic State Abstraction with DTMP

In Chapter 3, the MDP formulation of the metalevel planning problem was stated to be exact when sub-problems outcomes were restricted to a finite set of discrete values. One suggestion for dealing with the case of continuous sub-problem outcomes was to discretize the range of outcomes into a finite set of bins. The question then becomes how to perform the discretization. A natural suggestion is to try a uniform discretization with bins of equal size. A discretization that is too coarse may lead to many states being lumped together, while too fine a discretization may lead to unnecessarily many states. Though not examined in this thesis, the key idea is to generate a state abstraction in an adaptive manner, making distinctions in state outcomes when they lead to improvements in the resulting policies.

Consider Figure 4-10, which corresponds to the decision tree given in Figure 4-9. The x_1-x_2 plane, represents a continuous region in which a uniform discretization of the state space may not be the best form of discretization. The decision tree learning algorithm may be able to provide a state abstraction capturing the important aspects of the environment in a compact manner. The abstract states represent all problem instances that satisfy the conjunction of the splitting conditions. For instance, the leaf node corresponding to v_2 in Figure 4-9 contains all problem instances that satisfy the condition, $(x_1 > s_1) \bigwedge (x_2 > s_2)$. This aspect potentially allows for DTMP to be extended to the case of continuous sub-problem outcomes.

Dynamic state abstraction is reminiscent of the G-algorithm by Kaelbling and Chapman [12] and the U Tree algorithm by McCallum [34]. Uther and Veloso [49] extend both

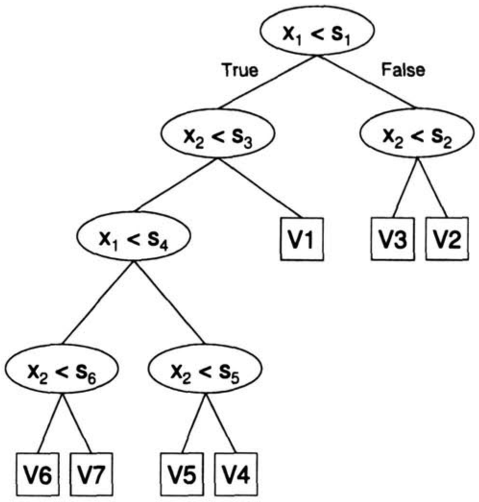

Figure 4-9: Decision tree value function and state abstraction.

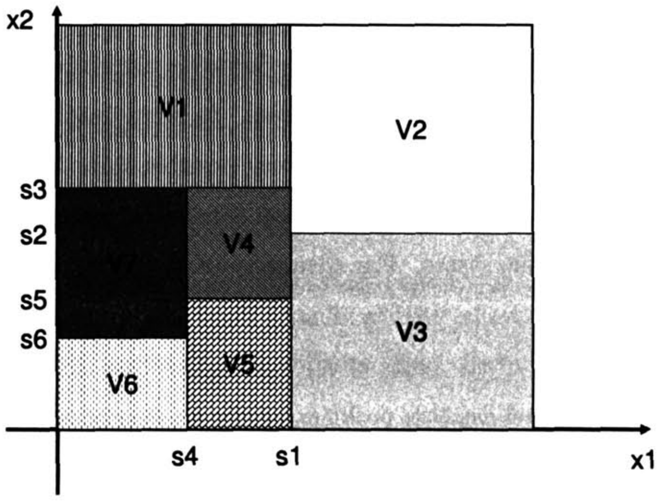

Figure 4-10: State abstraction for continuous state spaces according to the decision tree.

of these, and describe the Continuous U Tree algorithm, which is an iterative version of generating decision trees for continuous state space reinforcement learning. The decision tree acts as an automatic variable resolution state abstraction mechanism [35] uses data to determine state distinctions. The Continuous U Tree algorithm is very similar to the decision tree learning algorithm, but instead of using squared-error as a splitting criterion, the Kolmogorov-Smirnov statistical test is also applied. The K-S test is a non-parametric test used to detect whether two datasets of a given split differ significantly. If so, a split is added to the tree. Since their application area is reinforcement learning, the tree is incrementally built based on sampled experience. Only the leaves of the decision tree generated by the Continuous U Tree algorithm represent the states over which to plan.

The main difference in DTMP is that the information states include both the leaves and the internal nodes of the decision tree. In DTMP, the splitting variables and splitting points are part of the policy rather than a means to aggregate base-level states into a corresponding abstract state as in Continuous U Tree. Continuous U Tree also has no mechanism to account for the cost of computation involved in generating state distinctions (i.e., splits) while DTMP does.

4.6 Chapter Summary

This chapter presented an approximate policy iteration algorithm for solving the metalevel control problem based the use of decision trees. In many instances computing the exact solution of large-scale MDPs, considering every possible state in the state space, is computationally intractable and unnecessary when a good policy can be determined by only considering a subset of the states. The advantage of the DTMP algorithm is that decision tree learning algorithms can be used to identify the states in this subset in order to generate compact representations of the parts of the state space that matter. Once a decision tree is generated, the metalevel decision problem simply consists of selecting one of two actions. The metaplanner must decide whether to compute the next sub-problem dictated by the decision tree or stop computing and execute the current plan. As will be shown in Chapter 5, the performance of DTMP is comparable to exact solution methods and can be used to solve large, complex problem with limited computational resources.

Chapter 5

Experimental Results for Time-Separable Computation Costs

This chapter presents the results of both the exact MDP metaplanning solutions and the heuristic DTMP solutions for a variety of problem domains. In general, problem domains with stationary and additive computation costs are studied. Recall that in this case, the value function of a plan is independent of time. This property is referred to as time-separability. Results for a simplified version of the time-critical targeting problem are presented. These assumption are repeated here for convenience:

1. A prior decomposition of master and sub-problem levels exists.

2. Sub-problems consume the majority of computational resources.

3. The time costs are separable and additive.

4. A computational action involves solving sub-problems which yields a discrete set of possible values. The outcome is static once it has been determined.

5. Problem instances are drawn probabilistically from a set of base-level problem instances.

6. Execution can only occur when a feasibly executable plan has been generated.

The metalevel controllers generated through the methods discussed in Chapters 3 and 4 help the agent achieve bounded optimal behavior for each problem domain. Bounded optimality is guaranteed when the metalevel MDP can be solved exactly for the optimal policy. DTMP policies are shown to achieve a similar level of performance in problems where the exact solution is available for comparison, though bounded optimality is not guaranteed. The performance of these controllers is compared to other algorithms and heuristics, that are commonly applied to such problems, and shown to lead to performance improvements over those other algorithms. All of the problems in this chapter can be formulated as planning problems where the agent must generate a plan to reach some goal state from a start state with a minimum cost objective.

5.1 Time-Separable Goal-Directed Navigation Problems

This chapter presents the results of metalevel planning for a set of goal-directed navigation problems similar in nature to the graph problem discussed in Chapter 3. The notion of "goal-directed navigation" problems in this thesis involves generating a sequence of actions that will take the agent from an initial state to a goal state with minimum cost. More specifically, the focus will be to generate plans for discrete deterministic shortest path problems.

Shortest path problems have been well studied in the literature and serve as an abstraction for a variety of network flow problems [1]. For instance, the shortest path problem appears whenever some quantity of material (goods, computer data packets, vehicles, etc.) must travel between two points. Of particular interest is that fact that the shortest path problem, typically represented as a network of nodes and arcs, has a natural problem decomposition, suitable for the methods discussed in Chapters 3 and 4. Following the discussion in Chapter 3, the master level solves a shortest path problem with the arcs in the network representing the sub-problems. As discussed previously, the arcs can act as surrogates for arbitrarily complex sub-problems, while the manner in which the graph is configured can be thought of as constraints on the relationship between sub-problems.

5.1.1 Master and Sub-problem Level Formulation of the Shortest Path Planning Problem

The master level problem is formulated as a shortest path problem, represented in Equations (5.1-5.5) in terms of flows along arcs in a network.

$$\min \sum_{i \in NODES} \sum_{j \in NODES} c_{ij} x_{ij} \tag{5.1}$$

$$\sum_{i \in NODES} x_{si} = 1 \tag{5.2}$$

$$\sum_{i \in NODES} x_{ig} = 1 \tag{5.3}$$

$$\sum_{i \in NODES} x_{ij} - \sum_{i \in NODES} x_{ji} = 0, \forall j \in NODES. \tag{5.4}$$

$$x_{ij} \in \{0,1\} \tag{5.5}$$

The x_{ij}'s are binary flow variables representing the flows along arcs starting at node i and ending at node j. Equations 5.2-5.4 represent the flow balance constraints which state that the outflow from source node, s, sums to one, the inflow to goal node, g, sums to one, and the net flow for each of the remaining nodes sums to zero.

A sub-problem consists of determining the cost, c_{ij}, of moving along the corresponding arc. Given complete knowledge of the problem instance, all of the c_{ij}'s are known in advance. The metalevel control problem is to select sub-problems to solve in order to supply the c_{ij}'s to the master level. That is the costs, c_{ij}, are not known until the sub-problem associated with that arc has been solved. The MDP formulation for this problem has been presented in Chapter 3.

5.1.2 Shortest Path Planning Algorithms

This subsection describes the planning algorithms that have been developed to solve the shortest path planning problems in this thesis. There are a variety of ways to solve the shortest path problem (see [1] and [5] for a detailed discussion). Here, the focus will be on dynamic programming and label correcting methods. Dynamic programming, presented in Chapter 3 as a solution methodology for MDPs, applies the principle of optimality to efficiently generate optimal solutions. However, the difficulty with using dynamic programming to solve the shortest path problems addressed in this thesis is that DP requires full use of information. In order to generate the optimal solution, each node must be backed up, implying that the cost of each arc be known. This makes dynamic programming expensive when considering the cost of computation involved in solving a large number sub-problems.

Another set of shortest path algorithms, known as *label correcting methods* [5], tend to be more efficient than dynamic programming because they require fewer sub-problems to be solved. The key to the success of label correcting methods is the availability of a good heuristic evaluation function, which in effect serves as a primitive metalevel controller for suggesting the next sub-problem to solve. Label correcting methods all work on the same principle of progressively discovering shorter paths from the origin to every other node, keeping track of the node's "label" (i.e., the length of the shortest path from the origin to that node). A list of potential nodal expansions[1] called OPEN is maintained, where subsequent nodes are added to OPEN after an expansion. Label correcting methods differ mainly in the manner in which nodes (i.e., sub-problems) are selected. For instance, a breadth-first search expands nodes in a first-in/first-out manner while depth-first search expands nodes in a last-in/first-out manner. Dijkstra's algorithm [5] is one type of label correcting method called a best-first algorithm, where the node to expand is the one with the minimum label.

Derivatives of Dijkstra's algorithm have had great success in solving shortest path problems. Chief among these is A* [39] (pg. 97), a generalization of Dijkstra's algorithm combined with a heuristic function that provides a lower bound on the cost from each node to the goal. The A* algorithm reduces to Dijkstra's algorithm when the heuristic evaluator is set to zero. In general, A* with a good heuristic estimator will optimally solve the shortest path problems more efficiently than Dijkstra's algorithm.

The heuristic evaluation function is the key to the effectiveness of A*. Typically, the heuristic function is expressed in a per node basis and provides a lower bound to the true cost from the node to the goal. It can either be a fixed function, or determined through some other means such as the simulation of trajectories, but is typically represented as a function, $h(n)$, where n is a node in the graph. A heuristic function is said to be *admissible*, if it provides a lower bound on the true cost to the goal. The A* algorithm is guaranteed to

[1]Expanding a node in the context of label correcting methods involves incorporating the cost of moving from one node to an adjacent one, essentially equivalent to solving a sub-problem.

terminate with the optimal solution assuming that it is provided with an admissible heuristic function. As such, the A* algorithm is an example of a run-to-completion algorithm.

However, encoding the lower bound of the cost of computation in a useful manner is difficult since the heuristic estimate is only a property of individual nodes in the graph, while a lower bound for the cost of computation is really a function of the state of information. Naively, one may try to account for the cost of computation on a per node basis by using the estimate of the minimum number of additional sub-problems needed to generate a feasibly executable path. By using such a heuristic, a node will have the same estimate regardless of the information gathered during the course of planning, effectively negating its usefulness[2].

Recent work in the literature has modified the A* algorithm to exhibit anytime behavior, where a sequence of improving solutions is generated with increasing computation time. Likhachev et al. have developed the ARA* (Anytime Repairing A*) algorithm [30] which causes the A* algorithm to produce a sequence of improving solutions. This is accomplished by initially using an inadmissible heuristic with the A* algorithm, producing a sub-optimal path quickly. The inadmissible heuristic causes the A* algorithm to behave similar to greed depth-first search (potentially leading to the discovery of a feasible solution quickly). ARA* iteratively refines the heuristic function toward admissibility while generating intermediate solutions in the process. The final solution, assuming sufficient time to iterate, is the globally optimal shortest path. In Section 5.3, ARA* is applied to maze navigation problems. The results for the graph planning problems are presented.

5.2 Graph Planning

Both exact and heuristic metalevel controllers have been developed for progressively larger planning problems, as shown in Figures 5-1 through 5-3 for graph problems with 8, 16 and 24 arcs. The performance of the following four algorithms have been determined and compared:

1. PLAN ALL (full-state dynamic programming)

[2]What is really needed is a heuristic estimate for the cost of the plan given as a function of the current state of information. This is exactly what is provided by the value function V^* of the metalevel MDP. The value function can really be seen as the optimal heuristic function $h^*(s)$ of the MDP where s is the information state. The heuristic function, $h^*(s)$, does incorporate the information regarding the cost of computation, the graph structure and the set of solved sub-problem outcomes in the estimate, while $h(n)$ does not.

2. RANDOM-DIRECTED (greedy)

3. Metalevel MDP

4. DTMP

Both PLAN ALL and RANDOM-DIRECTED strategies represent the optimal metalevel policies for the extreme points on the time cost spectrum under the conditions of separable and additive time costs (PLAN ALL for zero computational costs and RANDOM-DIRECTED for very high costs). The results presented in each case assume that the cost of computation per sub-problem is fixed and given by ϵ, the marginal cost of computation. Binary sub-problem outcomes of $\{0, 2\}$ were assumed.

These conditions resemble those of the 4 arc graph problem in Chapter 3, where the bounded optimal policy was confirmed to be equivalent to PLAN ALL when the marginal cost of computation is zero (since solving sub-problems is free), and similarly the bounded optimal policy should be equivalent to RANDOM-DIRECTED when the cost of computation is high (since computations are expensive a solution should be found as soon as possible).

The exact solution of the metaplanning MDP was possible for both the 8 and 16 arc graphs, but not for the 24 arc graph. For 24 arcs, there are approximately 3^{24} (280 billion) information states, making an exact solution impractical. In this case, only the DTMP solution is presented.

Plotted in each chart (see Figures 5-4 through 5-6) is the performance of the different strategies in terms of the total expected cost, given as a function of the cost of computation, ϵ. The y-axis represents the performance of the algorithm in terms of expected total cost, which includes both the expected cost of the base-level plan along with the costs of computation. The x-axis represents the marginal cost of computation, ϵ, which is the utility cost per unit of computation. It is assumed that solving each sub-problem takes one unit of computation. The marginal cost of computation allows both the execution and planning costs to be expressed in units of utility. As the ratio of planning cost to execution cost increases, the cost of solving sub-problems becomes more expensive. In the subsequent charts, each data point corresponding to a particular value of ϵ is the average of the costs of each strategy on multiple problem instances.

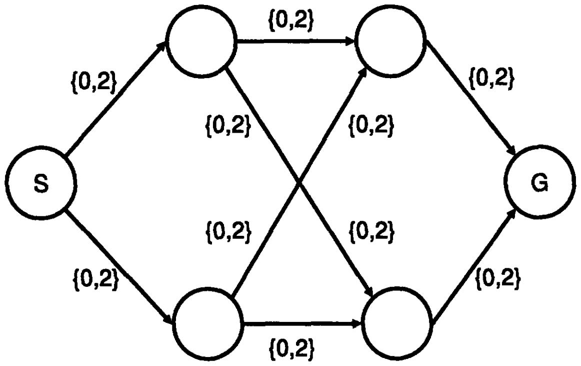

Figure 5-1: The graph for the 8 arc metaplanning problem.

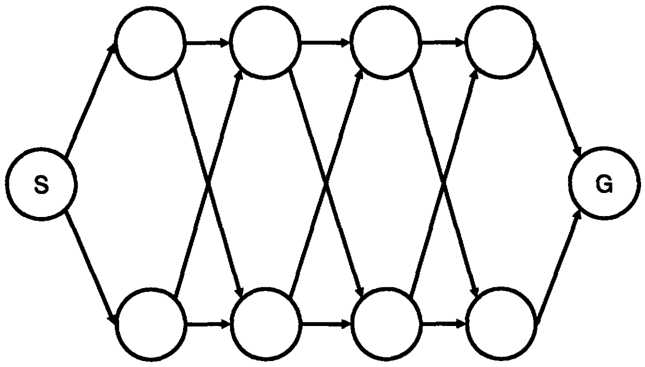

Figure 5-2: The graph for the 16 arc planning problem.

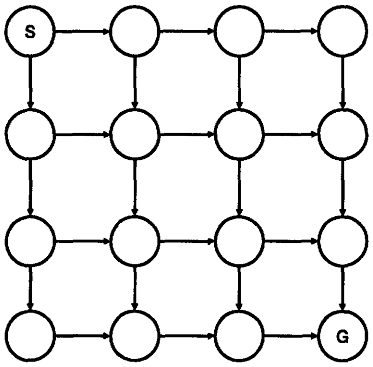

Figure 5-3: The graph for the 24 arc planning problem.

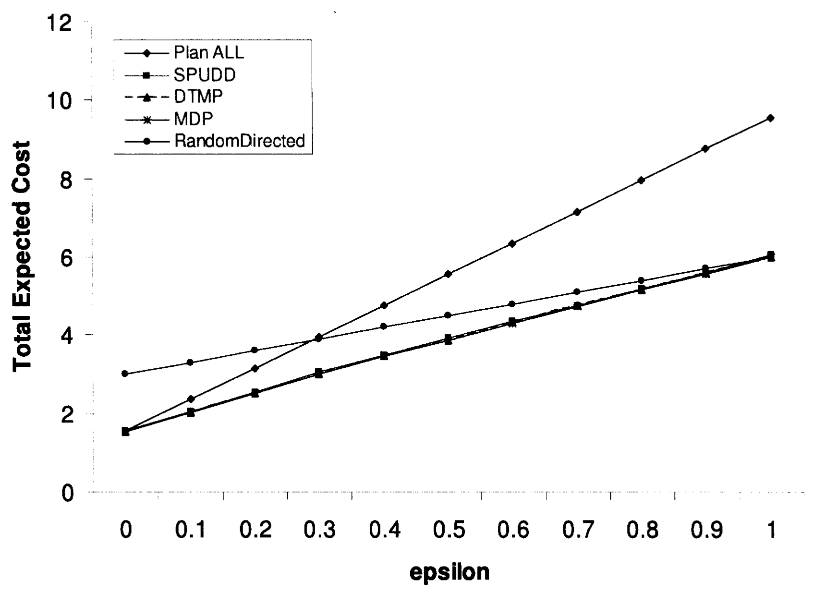

Figure 5-4: Results for the 8 arc graph problem with binary $\{0, 2\}$ arc cost distributions.

5.2.1 Graph Planning Results

Figure 5-4 shows the results for the 8 arc graph problem with binary $\{0, 2\}$ arc costs with equal probability. As compared to the PLAN ALL and RANDOM-DIRECTED strategies, the optimal metalevel planning policy, labeled MDP, exhibits bounded optimality across the range of ϵ. This plot has many notable characteristics that are common to the remaining graph problems. Those notable characteristics are highlighted here.

For this small problem, the exact metalevel MDP policy could be solved exactly using value iteration, and its performance is given by the curve labeled "MDP". As observed, from the plot in Figure 5-4, the optimal metalevel policy is PLAN ALL, when the cost of computation is zero, and switches to a policy equivalent to RANDOM-DIRECTED when the cost of computation, ϵ, is greater than 1. Notice that the value of the cost of computation where metaplanning results in the greatest gain is around the point where PLAN ALL and RANDOM-DIRECTED intersect, $\epsilon = 0.3$.

For this problem, the SPUDD algorithm [22], an alternative algorithm for optimally solving MDPs, was also used to solve the metalevel MDP planning problem. As seen in the plot, its performance is identical to the optimal policy. For this problem, neither sampling nor path completion was a major deterrent to the DTMP heuristic. Though its performance curve does not coincide exactly with the optimal policies, the differences are

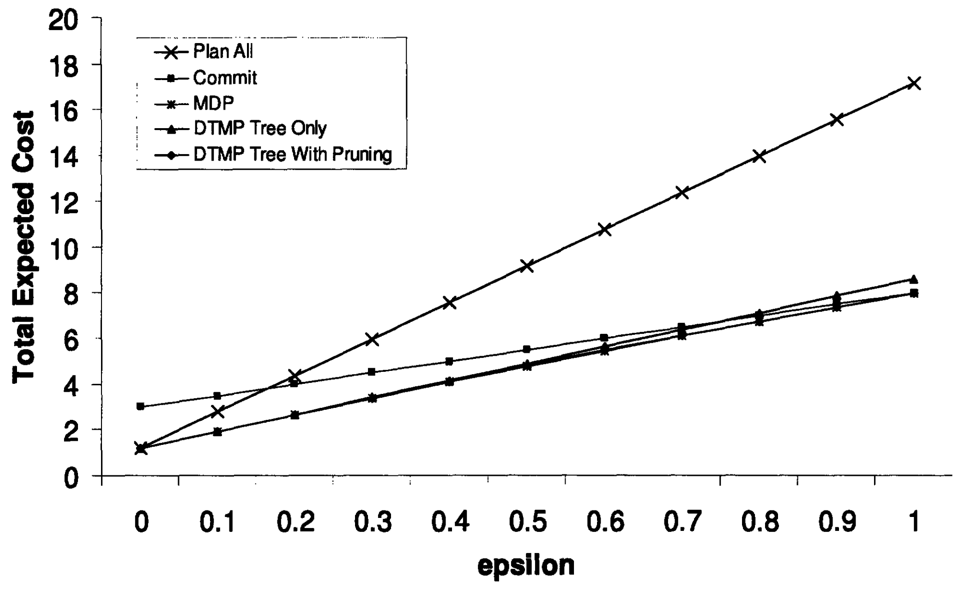

Figure 5-5: Results for the 16 arc graph problem with binary $\{0, 2\}$ arc cost distributions.

virtually indistinguishable, with a maximum difference for any data point being less than one percent. Error bars are zero for each data point, since all 256 problem instances were used to evaluate the performance of each policy.

Figure 5-5 shows the results for the case of the 16 arc graph with binary arc costs. In this problem, the probabilities for sub-problem outcomes were not equal so that each sub-problem had its own probability distribution. Again, the results of the PLAN ALL strategy are reported. The RANDOM-DIRECTED strategy is replaced by a COMMIT strategy, which commits to solving an *a priori* determined set of sub-problems, selected via solving the shortest path planning problem using expected arc costs. The performance of the optimal metalevel MDP policy, solved exactly, is also plotted.

The DTMP results include two cases, one where the decision tree policy was iteratively generated without the MDP pruning step and the other with the pruning step. This shows that the decision tree alone is unable to generate the best policy on its own. Notice that without MDP pruning, DTMP is unable to adapt to the cases with high computational costs. In the case where there is pruning, the performance of the DTMP metaplanning policy is comparable to the performance of the true optimal policy and exhibits performance that is near bounded optimal across the range of ϵ. Error bars were not plotted in the figure because they are too small to be seen. For both DTMP with and without the MDP pruning

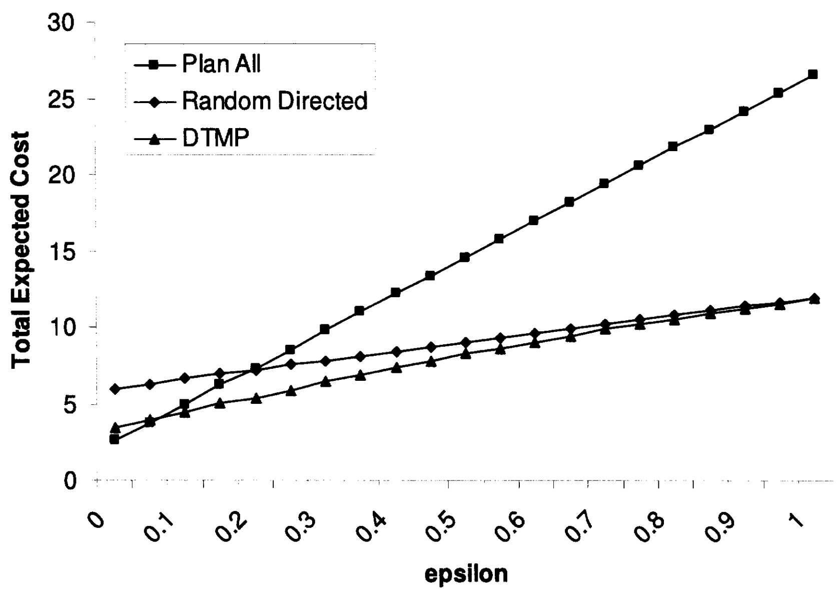

Figure 5-6: Results for the 24 arc graph problem with binary $\{0, 2\}$ arc cost distributions.

step, the standard error of the mean, was less than 0.038 for each data point, where each one is an average 5000 sample problems. This problem was too large to solve with SPUDD.

Figure 5-6 shows the result of the DTMP algorithm for the graph with 24 arcs with binary arc costs of equal probability. Though bounded optimality is not achieved, as with the previous examples, the metaplanning policies generated by the algorithm lead to improvements over the switching policy of choosing the best strategy between PLAN ALL and RANDOM DIRECTED for intermediate values of ϵ. In this case, the optimal metalevel policy was too large to be solved exactly as a full MDP or with SPUDD, making DTMP an attractive alternative that is able to yield performance gains over the other two strategies. Each data point is the average performance of the individual strategies over 1000 sample problem instances. The standard error of PLAN ALL, RANDOM DIRECTED and DTMP was no larger than 0.1 and thus not plotted. When $\epsilon = 0$, the DTMP policy clearly does not achieve the same cost as the PLAN ALL strategy. This is mostly due to the more complicated planning domain making the plan completion problem more difficult.

5.3 Maze Planning

The second type of goal-directed navigation problem addressed here takes the form of navigating through a maze. The maze planning domain is posed in the form of a shortest path planning problem, where a sub-problem decomposition is naturally defined by the individual path segments that form the maze. This problem is a parallel to the graph planning examples where the arc links correspond to a set of unspecified sub-problems. Instead, the sub-problems in the maze domain are themselves shortest path planning problems over path segments of the maze.

Each path segment is defined by the links connecting individual maze intersections, or junctions (equivalent to nodes in the graph problems). In order to keep with the directed acyclic arc convention used in the graph planning examples, it is assumed that each path segment of the maze is directed, so that no directed cycle exists. Figure 5-7, shows the layout of the maze used in the experiments. A path segment in the maze consists of a set of grid cells.

As was the case for the graph planning problems, the metalevel planner must solve a series of sub-problems to generate a least-cost, feasibly executable plan. One additional detail is the presence of a fixed number of obstacles on the path segments, which must be discovered in the process of solving sub-problems. The obstacles are assumed to be uniformly distributed over the maze, and their presence on a path segment makes it impassible. It is assumed that the maze configuration is known to the agent *a priori*, but a path segment cannot be used as part of the path to the goal until it has been confirmed to be clear of obstacles.

The maze problem is separated into two cases based on the level of detail of the sub-problem solution process. These two cases will be denoted as the plain-maze and the decomposed-maze.

In the plain-maze, solving a sub-problem is posed simply as examining the individual grid cells of a path segment for obstacles, starting from the head and ending at the tail (see Figure 5-8). It is assumed that examining or executing a single grid cell takes unit time. The cost of computation of any path segment is computed as the marginal cost of computation multiplied by the number of grid cells examined. The cost of execution is the number of grid cells that must be traversed to move across a path segment.

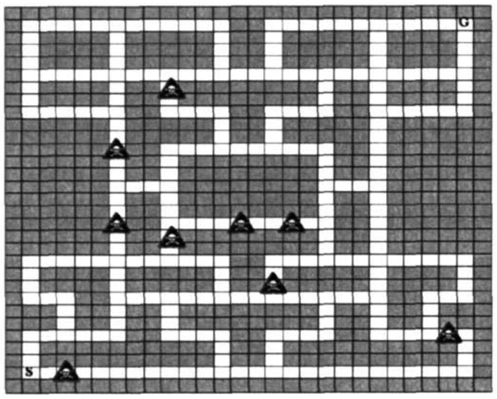

Figure 5-7: Plain-maze problem with obstacles.

In the decomposed-maze, each grid cell of the maze, is decomposed into a set of higher resolution grid cells. These higher resolution grid cells have varying execution costs, generated randomly. Figure 5-8 shows an example of the decomposition of a path segment. The sub-problems in this case consist of generating least-cost paths at high resolution through path segments of the maze. While the execution costs per arc in the plain-maze tend to be binary (either occupied or unoccupied), the arc execution costs in the decomposed-maze can take a range of values and are determined by solving the corresponding shortest path problem. Obstacles preventing the traversal of an arc are also present in the decomposed-maze and need to be discovered by the sub-problem planner.

In both cases, the maze itself is fixed and known to the agent while the sub-problems take the form of path planning from one junction of the maze to another. There are 64 junctions and 83 maze segments for the given maze. Figure 5-8 shows an example path planning problem for a path segment of the decomposed-maze consisting of four grid cells. The grid cells of that particular maze segment are magnified in the figure to reveal a finer grained, 3x3 grid cell decomposition for each of the original grid cells. This fine-grained representation is indicative of the path planning sub-problems in the decomposed-maze problem. A sub-problem consists of generating a shortest path from the grid cell labeled, h, for the head, to the grid cell labeled, t, for the tail of the path segment. The ARA* algorithm was used to solve the path planning sub-problems.

The cost function to be minimized for this particular example is expressed as an exponential in terms of the weighted sum of computation and execution costs[3], (i.e., the total

[3]This form of the cost function was chosen specifically to emphasize the importance of reaching

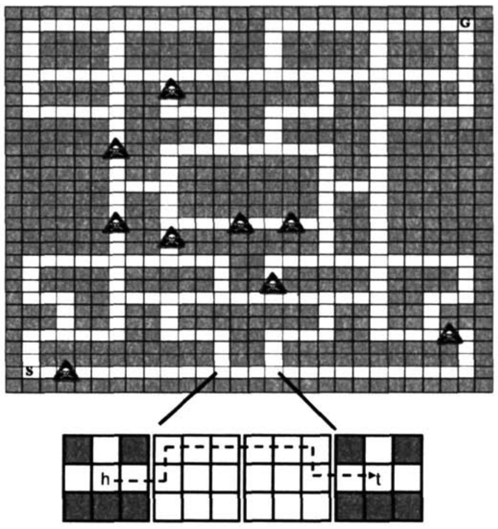

Figure 5-8: Shortest path sub-problem for the decomposed-maze domain. Each grid cell in the original domain consist of a finer grained grid. The head and tail of the path segment is indicated with h and t, and a path is given by the dashed line. The greyed cells indicate impassible regions.

time elapsed before reaching the goal location) and is given by

$$K \exp^{(w_c \frac{t_c}{N_c} + \frac{t_e}{N_e})}, \tag{5.6}$$

where K is a constant set to 100, and N_c and N_e are constants representing normalizing factors on computation and execution times respectively. These numbers were set to be proportional to the minimal times achievable for computation and execution when the maze is cleared of obstacles. The variables t_c and t_e represent the number of time steps that were used by the agent to compute and execute the plan. It is assumed that the traversal of a single grid cell during execution or an expansion of a single grid cell during planning constitutes one time step.

In the decomposed-maze, where the original grid cells are decomposed into smaller grid cells, the expansion of one small grid cell will still count for one time step, but the number of time steps needed for a single execution action will depend on the random cost assigned to the grid cell being traversed. The cost of execution of a small grid cell is at least 1 time

the goal quickly. Excessive exploration of the maze is even more expensive compared to the graph problems of the previous section. In this case, the time cost separability assumption still holds, but is no longer additive.

step plus a randomly generated additive factor.

In the maze domain, the weighting factor w_c plays the role of ϵ and is the ratio of the cost of execution to that of computation. When the value of w_c is zero, the cost of reaching the goal is a function of the execution costs only. When w_c is high, the cost of computation dominates the cost function, and slightly longer paths that can be found quickly are preferred.

5.3.1 Comments on Maze Results

The ARA* algorithm was used to implement strategies similar in nature to the **PLAN ALL** and **RANDOM-DIRECTED** strategies in the maze domain. ARA*'s anytime nature has the property that it is able to generate a range of solutions. This is controlled by two parameters, a weighting factor, w_h, to control the amount of heuristic overestimation of the true cost-to-go, and a decrement factor, δ_h, to control the amount to decrease the weighting factor per iteration of ARA*. The weighting factor is multiplied by the heuristic estimate to generate an inadmissible heuristic function. ARA* uses the Manhattan distance as the default heuristic estimate in the maze experiments. For the experiments, w_h is set to 10 and δ_h is set to 0.5, so that after 18 iterations, the weighting factor becomes 1 (making the heuristic admissible). A high weighting factor for the ARA* algorithm generates behavior similar in nature to a **RANDOM-DIRECTED** strategy (not random but greedy), while a low weighting factor effectively makes ARA* behave similar to A* search, resulting in the generation of an optimal path.

There is a choice in the reporting of the ARA* results. Since ARA* is an anytime algorithm it is possible to determine an optimal stopping time, run the algorithm for the prescribed time, and report the results. Here, a more favorable reporting for ARA* was chosen. Instead of iteratively running the ARA* algorithm for a sequence of decreasing weighting factors, only the results for running ARA* on the best weighting factors were used. This is favorable for ARA* because the computations that are performed under intermediate weighting factor values may not be the same set of computational actions for the best weighting factor had it been used as the initial value. Reporting the results for iteratively running ARA* would yield higher cost plans due to accumulating computation costs from generating intermediate solutions in previous iterations[4]. The best weighting factor is chosen by evaluating ARA* with each weighting factor individually over a set of problem instances. The result that yields the lowest expected cost for over all values of w_c is reported.

The directed nature of sub-problems, which makes each path segment of the maze a directed arc, allows for the possibility of randomly generated obstacle placements to prevent the metalevel controller from generating any feasibly executable path. In some of these cases, ARA*, not being restricted in direction, is still able to find a path. In other cases neither the metalevel controller nor ARA* could find one. Both of these cases problematic problem instances are detectable and not included in the reporting of the maze results.

Finally, to deal with the path completion problem, each leaf node of the decision tree is associated with a set of feasibly executable paths learned from data. When a leaf node is reached, a candidate plan is selected from the list in the order of expected plan length, with shorter plans being preferred. The remaining steps were either to solve the unsolved sub-problems until a feasibly executable plan was obtained or select another candidate plan when the current plan was deemed infeasible.

5.3.2 Maze Results

For the maze problems, the exact solution of the metalevel MDP is computationally intractable so that only DTMP results are given. Figure 5-9 shows the results of DTMP and ARA* for a range of computational weighting factors for the plain-maze world consisting of simple grid cells. When the weight of computational effort is zero, DTMP and ARA* perform similarly, expending effort to search the maze for the shortest possible path. As the cost of computation is increased, ARA* continues performing its default search behavior, while DTMP learns to direct the search effort to the relevant parts of the maze in order to generate good solutions quickly. As stated above, the ARA* results are given for the heuristic weighting factor, w_h, yielding the lowest expected cost for each w_c. As w_c is increased, the best heuristic weighting factor tends to increase, while the best heuristic weighting factor is 1 when w_c is zero. This is understandable since a higher heuristic weight-

[4]The ability for anytime algorithms to produce intermediate solutions, in general, results in some computational overhead. By setting the weighing factor to a specific value, the computational cost due to having to generate intermediate results is bypassed.

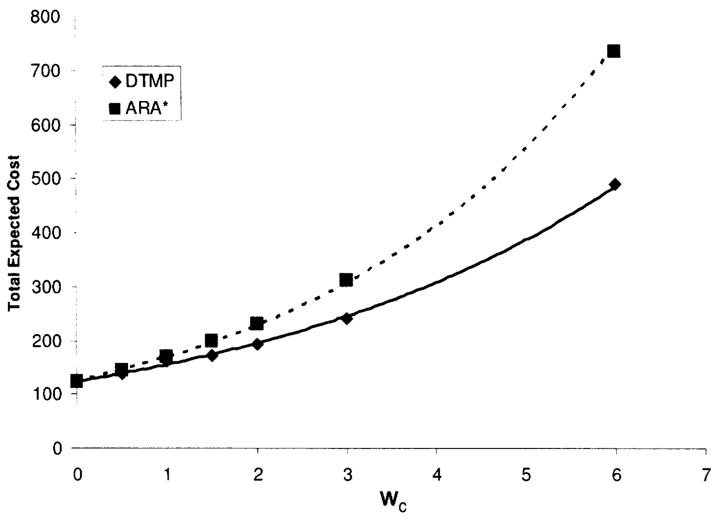

Figure 5-9: Plain-maze world results with exponential costs.

ing factor results in greedy planning behavior. While both the DTMP and ARA* curves are exponential with the cost of computation, DTMP produces plans that tend to better balance the effects of computation on the objective function and the resulting plans. The main benefit of the metalevel controller in this situation is its explicit control over which computations to perform. In comparison, the computational actions taken by ARA* are not subject to control. The algorithm only allows for explicit control of computation time.

Figure 5-10 shows the results of DTMP and ARA* on the decomposed-maze domain. Again, DTMP and ARA* perform comparably when the cost of computation is zero. ARA* performs slightly better since it can generate optimal paths, while DTMP, due to issues with path completion does not guarantee optimality. Again, as more weight is put on the cost of computation, the advantage of DTMP becomes apparent. The ARA* algorithm does not explicitly account for the cost of computational actions, while DTMP does. As is typical with most anytime algorithms, the optimization of the amount of computation to allocate is performed externally of the algorithm, and an agent is asked to make the best use of the solutions provided by the anytime algorithm. However, for the maze domain, ARA* often generates expensive (long execution time) paths with very little computational effort while less expensive (short execution time) paths take much more computational effort to discover. Although the results of DTMP could not be compared against the exact metalevel MDP solutions, the DTMP algorithm appears to be generating metalevel controllers with a semblance of bounded optimality. The fact that it outperforms ARA*, which can claim

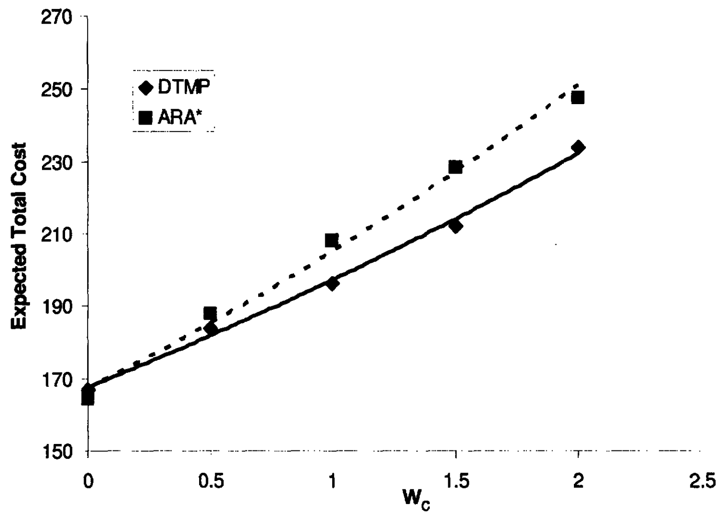

Figure 5-10: Decomposed-maze results with exponential costs.

to be bounded optimal when then optimal stopping time is used, is evidence that supports this hypothesis.

5.4 Chapter Summary

This chapter has demonstrated the utility of metalevel planning with feedback from subproblem outcomes, and in particular the controllers generated by DTMP, on a variety of problem domains. The first set of problems fell into the category of deterministic shortest path planning problems. These problems were chosen because they are flexible abstractions that can be made to represent many other interesting planning problems. The metalevel MDP formulation of these problems assumed a separable cost of time.

While there are many efficient algorithms for solving shortest path problems, they are, in general, unable to account for the computation costs (expressed either explicitly or implicitly) involved in plan generation. One alternative was to use available anytime shortest path planning algorithms, such as ARA*, to account for the cost of time. While anytime algorithms offer additional flexibility in terms of a solution quality versus computation cost tradeoff, the sequence of computations performed by the algorithm is not under the control of the agent. In anytime shortest path algorithms, the only control over the algorithm is the length of computation time. The DTMP algorithm, by comparison, offers even more flexibility since it can be used to generate a specific algorithm to suit the situation. This gives the metalevel controller the ability to direct computational effort towards high utility sub-problems while also controlling computation time. For the graph problems, the DTMP-generated metalevel policies were shown to perform favorably in comparison to traditional PLAN ALL and reactive RANDOM-DIRECTED strategies (as well as the simple switching policy) across a range of computational costs. More importantly, it was possible to confirm that the metalevel policies led the agent to perform in a bounded optimal manner. In the maze domain, the DTMP metalevel policies also outperformed plans generated using anytime algorithms.

It has been shown that bounded optimality is achievable for small problem domains by solving the metalevel MDP exactly with the approaches developed in Chapters 3. For larger problems, where exact solution methods are intractable, DTMP has been shown to be a viable substitute. This accomplishment is mainly due to the ability of the metalevel controller to learn to exploit structure in the problem domain and to actively use feedback to select high utility computations. Both the MDP and decision tree approaches infer the utility of solving sub-problems by accounting for the current state of computation in terms of known sub-problem outcomes, the distribution of problems instances for a particular problem domain, and effect of the cost of computation on the overall cost.

In the next chapter, DTMP is adapted to problems that are not time-separable. For these problems, the cost of computation cannot be explicitly represented as they were in this chapter, but is implicitly defined by the nature of the base-level problem.

Chapter 6

Vehicle Routing with Time Windows

Metalevel planning for problems with deadlines and hard temporal constraints presents a special challenge not found in the set of problem domains discussed in Chapter 5. In those prior planning domains, the value of a sub-problem, once solved, remained constant for the remainder of the planning episode. For time-critical problems, however, a sub-problem can lose value or become invalidated with the passage of time. The ramification for metalevel planning is that the cost of computation is no longer stationary or time-separable, but implicitly defined by the dynamics of the problem definition. The time-critical targeting problem presented in Chapter 1 is representative of the types of problems that fall into this category. In this chapter, the targeting problem is formally introduced in terms of the vehicle routing problem with time windows (VRPTW).

The vehicle routing problem (VRP) and its variants are well-known problems in the Operations Research community. The basic vehicle routing problem is a generalization of the traveling salesman problem (TSP), which can be stated simply as the problem of determining the least cost tour for a set of cities. A tour is a Hamiltonian cycle [37] through the set of cities, where each city is visited once by the salesman and must end with the salesman returning to the starting city.

This is a combinatorial optimization problem that is known to be NP-complete [1]. The VRP is a generalization of the TSP to multiple salesmen or vehicles. In this case, a set of exactly K circuits which cover the customer demands must be found, where K corresponds to the number of vehicles.

In its most basic form, the VRP is known as the *Capacitated* VRP (CVRP) and consists of a set of customers that have demands for goods which must be delivered by a fleet of identical vehicles based at single supply depot. The demands are known in advance and may not be split. Each vehicle has a maximum capacity for carrying goods and the objective is to minimize the total travel cost to satisfy all customers [46].

While there are many variants of the VRP, the one considered in this thesis is one with the time-criticality property, namely the vehicle routing problem with time windows (VRPTW). The constraints are similar to the vehicle routing problem, where tours must be generated to satisfy customer demands with minimum cost. Additionally each customer has an associated start and end time corresponding to the time interval over which a customer can be feasibly visited. This problem is known to be NP-hard [46]. It can be viewed as analogous to the time-critical targeting problem by replacing customers with targets. The time windows now represent windows of opportunity for striking targets, where they appear at the start time and disappear after the end time. A single tour in the VRP will be called a mission in the context of time-critical targeting. The deadlines, or end times, are what cause missions to become invalid over time. A mission consisting of a set of targets whose deadlines have all expired will have no value. While the metalevel control problem, in principle, is still to balance the cost of computation with the benefit of generating feasibly executable plans, the metalevel control problem has become more difficult due to the possibility that the completed plans generated by the metalevel controller are no longer guaranteed to remain executable throughout the planning episode. It will be shown that the DTMP algorithm can be adapted to this case and perform well with little additional overhead.

6.1 Exact MDP Formulation of a Vehicle Routing Problem

The vehicle routing problem with time windows (VRPTW) cannot be addressed by the original metalevel MDP formulation in Chapter 3. The main difficulty is the presence of time windows. Time windows are presented as a tuple $\{t_s, t_e\}$, which represents the time interval of availability of each target. That is, arriving at the target prior to the start of availability, t_s, or after the end of availability, t_e, results in no value accumulation.

The metalevel MDP formulation can be modified to accommodate the effect of time windows through the addition of time in the description of the information state. Recall that in the time-separable case, it was sufficient to represent the state as a tuple of sub-problem costs. In this case, the state description consists of a tuple of pairs, where each pair contains the cost of a sub-problem along with the latest time to execute the plan associated with it. It is assumed that the plan generated for each sub-problem has accounted for target start times. Additionally, there is a state variable added to indicate the current time, so that the information state is

$$s = \{< sp_1, ls_1 >; \ldots ; < sp_n, ls_n >; t_{now}\}$$

where sp_i is value of executing sub-problem i, ls_i is the latest start time the plan for a sub-problem i can be feasibly executed, and t_{now} is the time elapsed since the start of planning. The latest start time for a plan can be determined by taking, for each target in the plan, the planned arrival time, assuming that t_{arr} is given in terms of absolute time, and subtracting it from its end of availability time. This difference is referred to as the slack in the plan,

$$slack(j) = t_e^j - t_{arr}^j,$$

where the superscript j indicates the corresponding target, and the latest start time for each sub-problem plan is the minimum slack time over all targets,

$$ls = \min_{j \in TARGS} slack(j) + t_{now}.$$

The latest start time provides information on feasible execution, where a sub-problem plan is not executable once $ls_i > t_{now}$. Assuming that the set of sub-problem outcomes, planning times and time windows each takes on a finite number of integer values, the state transitions are similar to those of the separable time cost metalevel MDP with a few differences. First, every compute action results not only in a state transition for the sub-problem outcome, but also for the latest start time of the resulting plan. In addition, a compute action increments the value of t_{now}. Another possible effect of a compute action may be the invalidation of previously feasible sub-problem plans.

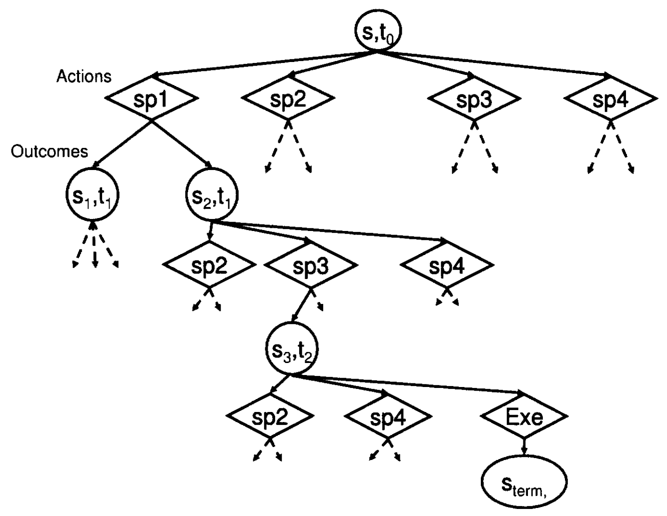

Figure 6-1: The VRPTW as a sequential decision problem.

Figure 6-1 shows the sequential decision problem for the VRPTW. The circles represent the information state just as before, except the t_i's indicate the value of t_{now} and the s_i's consist of the both the plan costs and their latest start times.

The size of this state space is exponential in the number of possible sub-problem outcomes times the number of possible latest start times, such that

$$|S| = O((|M||T|)^{|SP|}|T|)$$

where $|M|$ is the size of the set of discrete sub-problem outcomes and $|T|$ is the size of the set of discrete time values. Compared to the complexity of the MDP formulation of Chapter 3, this problem is much larger due the appearance of T, the representation of time, at the base of the exponential. Clearly this problem can become computationally intractable very quickly. Solving the problem exactly is generally not practical. In the following section, a modified version of the DTMP algorithm, described in Chapter 4, that overcomes those computational issues at the cost of a loss of optimality, is introduced.

6.2 Master and Sub-problem Level Formulation of VRPTW

The master and sub-problems for the vehicle routing problem with time windows can be addressed via a composite variable formulation. Composite variables, (see Armacost [2] for details) are sub-problems typically employed in integer programming and network flow problems, where the master level problem can by expressed as a set covering problem (e.g., selecting enough sub-problems so that all targets are covered). This fits exactly with the assumption of the existence of a hierarchical problem decomposition. The composite variable formulation of the VRPTW in this thesis consists of sub-problems, each of which solve a traveling salesman problem with time windows (TSPTW), along with a master level integer programming problem for handling the high level constraints of the VRPTW, such as not selecting mission combinations where a single target is visited more than once or assigning the same vehicle to two different missions simultaneously.

The TSPTW sub-problem consists of determining a traveling salesman tour (mission) for a given subset of targets along with their time windows, the vehicle assigned to the mission, and the current location of the vehicle. A feasible solution to a sub-problem is a mission consisting of the optimal (minimal cost) order to visit each target in the assigned subset, the arrival times at each target, and the value of the mission. The value of the mission is determined by the sum of the target values in the mission minus the travel costs.

A feasible mission consists of a plan which visits each of the targets assigned to the sub-problem once, while respecting the target time windows, and the return-to-base constraint. Missions are not guaranteed to be feasible. An infeasible mission might be the result of having too many targets assigned to one vehicle such that it cannot feasibly meet all deadlines.

6.2.1 Master Level Formulation

The master level acts as a high level plan coordinator that generates the final plan through the selection of a combination of sub-problem solutions (missions). The master level selects the sub-problems to solve in a manner that ensures that the constraints of the original VRPTW are satisfied. In its simplest form, the master level problem formulation for the

VRPTW consists of choosing a set of missions which ensure that targets are not visited by more than a single vehicle while maximizing the combined value of the selected missions. The time window constraints are handled by the sub-problems and do not appear in the master level. In addition, typical capacity constraints in terms of fuel and the number of weapons carried that would also be addressed by the sub-problem solver are not considered in this thesis in order to reduce the time-critical problem to its bare essentials. The master level integer program can be written as follows

$$\begin{aligned}
\max \sum_{k \in VEH} \sum_{i \in MISS_k} v_{ik} x_{ik} & & \quad (6.1)\\
\sum_{k \in VEH} \sum_{i \in MISS_k} m_{ijk} x_{ik} &\leq 1, \forall j \in TARG & \quad (6.2)\\
\sum_{i \in MISS_k} x_{ik} &\leq 1, \forall k \in VEH & \quad (6.3)\\
x_{ik} &\in \{0,1\}. & \quad (6.4)
\end{aligned}$$

where $MISS$, VEH, and $TARGS$ are sets representing the available missions (available sub-problem solutions), vehicles and targets over which the master level has to plan, respectively. The missions are individually indexed per vehicle, such that $MISS_k$ is the set of missions developed by the sub-problem solvers for vehicle k. The value of executing the ith mission of vehicle k is represented by v_{ik}. The mission selector variables (the "composite variables") are represented by x_{ik} and, as indicated in Equation 6.4, are binary, taking on the value of 1 when mission i for vehicle k is selected and a zero otherwise.

There are two major constraints represented by Equations 6.2 and 6.3. The first states that a target should not be visited more than once by any vehicle. The mission detail vector, m_{ijk}, is binary, with a 1 indicating that target j is to be visited by the ith mission of vehicle k. In a typical vehicle routing problem, Equation 6.2 would be an equality, constraining the solution such that each target must be visited exactly once. In a time-critical situation, however, the best set of missions to execute may not cover all targets. In fact, the time it takes to compute a set of missions that completely covers all targets may lead to a solution that is significantly inferior to a solution that only covers a subset of targets. Relaxing this constraint allows for the possibility for missions to be executed sooner. It also has the implication that some vehicles might not be assigned to a mission.

The second constraint, Equation 6.3, states that vehicles are only allowed to execute at most a single mission. This is a typical constraint to represent the possibility that a vehicle might need to be refueled or reloaded with ordinance after executing its mission. As in the previous problems, it is assumed that all missions are simultaneously executed only after a complete plan has been computed.

6.2.2 Sub-problem Formulation

A sub-problem consists of finding the lowest cost route for a specific mission, that is, a single vehicle mission associated with a subset of targets. When each vehicle is assumed to have equal capability and to start from the same base, missions generated for one vehicle are also feasible for all other vehicles. A problem instance is assumed to consist of a fixed number of targets over a given geographical region. In general, it is assumed that problem instances are generated according to some distribution over target locations and time windows. The manner of defining sub-problems (i.e., mission generation) is dependent on target indexing (more below). Targets for each problem instance are assigned an integer index or target identification (targetID) number. Based on the targetID, the complete set of sub-problems consists of missions that can be potentially generated from the power set, $\boldsymbol{P}(\text{TARGS})$, or the set of all subsets of target indices, ranging from sub-problems consisting of single targets to sub-problems consisting of the complete set of targets[1].

Each sub-problem is a TSPTW and is solved through a dynamic programming algorithm with pruning, developed by Dumas et al. [16]. The algorithm either finds an optimal tour, visiting each target in the mission once, while respecting the time window constraints or returns that the specified mission is infeasible. A mission is infeasible when a tour cannot be generated to visit all of the targets satisfying their time window constraints. Infeasible missions are not added to the pool of missions, $MISS$, for the master level to select. The output of solving a sub-problem is an optimal tour, the target arrival times, and the cost of executing the traveling salesman tour. Each of the targets has an associated target value, and the value of mission i for vehicle k, v_{ik}, is computed as the difference between the total

[1]The mission consisting of the complete set of targets is equivalent to solving the entire VRPTW, and is not feasible. In practice, the set of feasible sub-problems consisted of missions with far fewer targets. This fact was used to reduce the set of sub-problems considered during training.

target value of the mission and the cost of the traveling salesman tour,

$$v_{ik} = \sum_{j \in TARG_i} TV_j - MC(i) \tag{6.5}$$

where $TARG_i$ is the set of targetID's pertaining to mission i, and TV_j is the target value of target j. The mission cost, $MC(i)$, or cost of executing the tour is part of the outcome of solving the TSPTW sub-problem. In this thesis, the travel distance of each mission was used as the mission cost, although, in general, other measures such as travel time and mission risk may be incorporated.

$$\begin{array}{cc} & \text{Missions} \\ \text{Targets} & \begin{bmatrix} 1 & 0 & 0 & 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 1 \end{bmatrix} \end{array} \begin{bmatrix} x_{11} \\ x_{21} \\ x_{31} \\ x_{41} \\ x_{51} \\ x_{61} \\ x_{71} \end{bmatrix} \text{Missions for vehicle 1.} \tag{6.6}$$

The sub-problems take care of generating feasible missions for each vehicle, which can be seen in the expression given by 6.6. It shows all seven possible missions for three targets generated for vehicle 1. Each ij^{th} element in the matrix corresponds to an entry of the mission details m_{ij1} in Equation 6.2. Each row of the matrix corresponds to a target, and each column corresponds to a mission. The targets assigned to a mission are indicated with a 1 in the corresponding row of m_{ij1} so that the first column corresponds to a mission which prosecutes the first target only and the last column is a mission for all three targets.

This problem is conceptually similar to the graph planning and shortest path planning problems discussed previously. The major challenge is in addressing the problem of changing constraints, which is different from previous cases where the feasibility and time cost of each sub-problem was stationary. For the VRPTW, the cost of time cannot be explicitly represented and is a function of the problem dynamics, that is, target values change as a function of time. The cost of time manifests itself in this domain as the loss of opportunity to execute missions. The metalevel planner must weigh the consequences of solving additional sub-problems with the risk that the currently valid missions corresponding to the current best plan may become invalid with additional computational delay.

In order to generate the metalevel plan, sub-problems are assumed to take some characteristic planning time, learned through simulation. In addition, each mission has an associated latest start time, indicating the latest time at which the mission can be executed. The determination of latest start times was discussed in Section 6.1. As more sub-problems are computed, previously generated missions may "expire" due to the current time exceeding their latest start times. This effect is taken into account when generating the metalevel control policy, with training data that reflects the possibility that additional planning invalidates previous missions.

6.2.3 Consistent Target Indexing

The assumption in this thesis is that the sub-problems, given by the hierarchical decomposition, are defined externally as part of the problem description, and the best metalevel controller is learned for the given decomposition. In the case of the vehicle routing problem, since sub-problems are defined as subsets of targets, the manner in which the targets are indexed plays an important role in the definition of sub-problems. As a consequence, poor or inconsistent target indexing will affect the decision tree learning process.

In the graph and maze problems of Chapter 5, sub-problems naturally correspond to the arcs in a graph or segments in a maze. These sub-problems are consistently defined over every problem instance, such that, referring to Figure 3-1, the arc SA, corresponding to sub-problem 1, is always known to be followed by the arc AG, corresponding to sub-problem 3. With consistent indexing, the learning algorithm is able to derive a relationship between these two sub-problems.

However, in the case of the vehicle routing problem, each sub-problem is defined in terms of a subset of targets. These subsets are, in turn, determined by the set of target indices. One drawback to defining sub-problems in such a manner is that the actual targets in a given subset will be highly dependent on the way in which the targets are indexed. A simple example where inconsistent indexing leads to problems is dialing a phone number on a telephone where the keys are unlabeled and scrambled (inconsistently indexed) after each phone call. When scrambling occurs, it may be impossible to ever successfully dial the right number. However, if the keys are unscrambled (consistently indexed), even if the keys are unlabeled, one might eventually learn the digit corresponding to each key. Consistent indexing in this case involves mapping each key to the correct digit.

In the same vein, a haphazard target indexing scheme will make it difficult, if not impossible for the learning algorithm to learn relationships between sub-problems. As another example, inconsistent target indexing is akin to randomly assigning a sub-problem index to each arc in the 4 arc problem for each problem instance. If arc SA is denoted as sub-problem 1 in some problems instances, while arc SB is denoted as sub-problem 1 in other instances, the learning algorithm will have a difficult time learning the true effect solving the individual arcs. Since the learning algorithm only knows about sub-problems and not arcs, the learned consequence of solving sub-problem 1 will be averaged over the consequences of solving arcs SA and SB, respectively.

It is especially desirable to avoid this issue in the vehicle routing problem, since target indexing can exacerbate the issue of learning relationships among sub-problems. Given any instantiation of target locations over the same geographical area, the indexing of targets can occur in any number of ways. A consistent way of indexing the targets is sought. Note that, in addition to problems associated with learning a metalevel control policy, the problem of consistent target indexing also affects the execution of the metalevel control policy. Suppose that during training, a particular indexing scheme is used for the set of training instances. In the field, the same indexing scheme must be used. The results of using the metalevel controller with a set of sub-problems generated under a completely different indexing scheme may be unpredictable and/or perform very poorly.

Figure 6-2 shows the difference between "consistent" versus "inconsistent" target indexing for a simple example. Here, "consistent" notionally means that targets are indexed across problem instances in such a way that the sub-problems (missions) that are defined according to this indexing will enable the learning of significant relationships among missions. On the other hand, "inconsistent" target indexing hinders the learning process.

Figure 6-2a shows one target indexing scheme for one instantiation of the problem. Suppose that it is known that the targets labeled 1 and 2 in Figure 6-2a tend to appear together in the same geographic area such that "good" missions will have targets 1 and 2 constitute one mission and target 3 be another mission. Figures 6-2b and 6-2c show examples of consistent target indexing on another problem instantiation (targets are in different locations), allowing a metalevel controller to learn that targets 1 and 2 belong

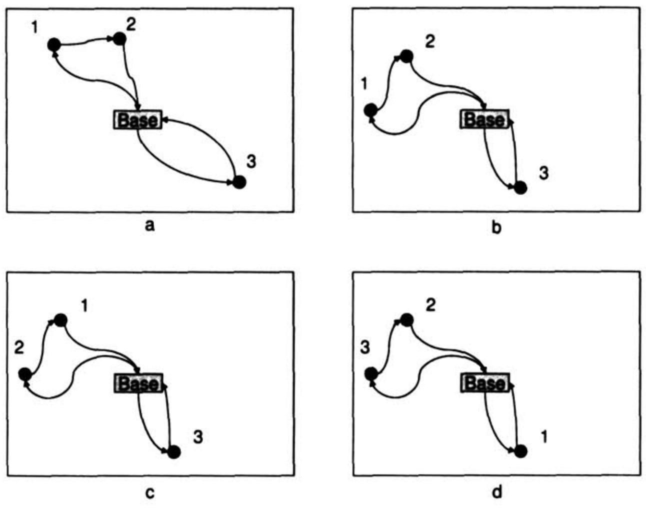

Figure 6-2: Consistent target indexing is a key to learn relationships between subproblems. a) Original target indexing. b) and c) Two ways of target indexing for a new problem instantiation that are consistent with a. d) Target indexing that is not consistent with a. Target 1 in d is not in the same quadrant as target 2 as in a, b or c. This inconsistent indexing will make learning very difficult.

together. Figure 6-2d gives an example where the targets are indexed in some other fashion, so that a mission with targets 1 and 2 would not be a "good" mission. Thus, if one were trying to learn which sets of targets were likely to constitute "good" missions the labeling in Figure 6-2d would hinder the learning of high utility missions by this inconsistent target indexing. Decision tree learning will have difficulty in conceptualizing between good and bad missions (in this case, it is assumed that "good" missions consist of targets that are clustered together).

In an effort to enforce more consistent target indexing, one could imagine enforcing an order on targets by overlaying a grid over the geographic region, as shown in Figure 6-3. This can help induce an ordering on the target locations based on grid numbering. Targets falling within the first quadrant are labeled first, followed by targets in the next quadrant and so on. If there are few targets appearing within a quadrant, the order of labeling might be assigned randomly. For instance, swapping the indices for targets 1 and 2 in the lower left quadrant of Figure 6-3 may be acceptable. One can consider having finer grained grids to accommodate more complex problems with higher target densities. This method for enforcing consistency is employed for the experiments in Section 6.4. An alternative indexing/labeling possibility is to apply a minimum spanning tree algorithm [1]

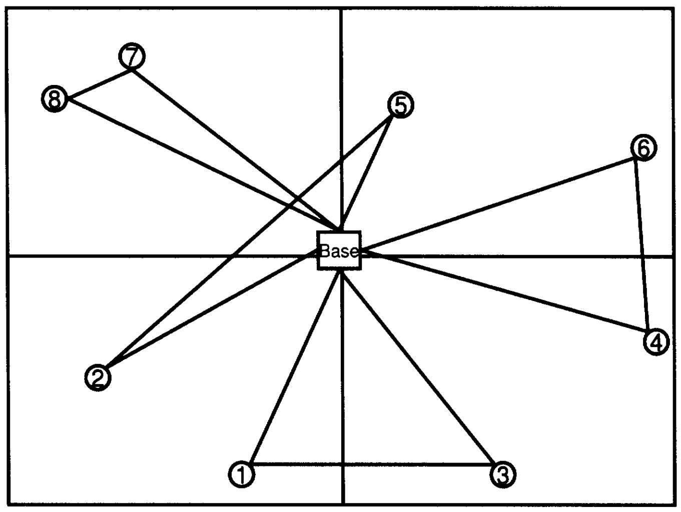

Figure 6-3: Partitioning a geographical region in order to enforce consistent target indexing.

(pg. 510-542) which can be used to index targets according to the order in which they are added to the minimum spanning tree. The ideal method for performing target indexing to generate meaningful sub-problems has not been fully explored but should be a subject for future research. Next, another way of defining sub-problems, which avoids the problem of target indexing is discussed.

A spatio-temporal based approach for defining sub-problems can be explained by referring to Figure 6-4, which shows a spatio-temporal view of the vehicle routing problem with time windows. The x-y plane shows the base and target locations, the curved dotted lines represents a spatial tour, and the t-axis represents time. Each location has a straight line running upwards with horizontal bars on these lines representing the target time windows. The arrows connecting the time windows represent the state of the vehicle in both time and space during tour execution. From this perspective, one can develop another way of decomposing the problem into a set of sub-problems.

Rather than indexing sub-problems based on target indices, which has foreseeable problems with consistent indexing, it may be possible to generate sub-problems by clustering on a spatio-temporal basis. For instance, sub-problems can be clustered according to ellipsoids of fixed volume and labeled according to their distances from the base. The logic behind this is that targets which are in physical proximity of one another and whose time windows are close in proximity should be clustered together. This bypasses the problem

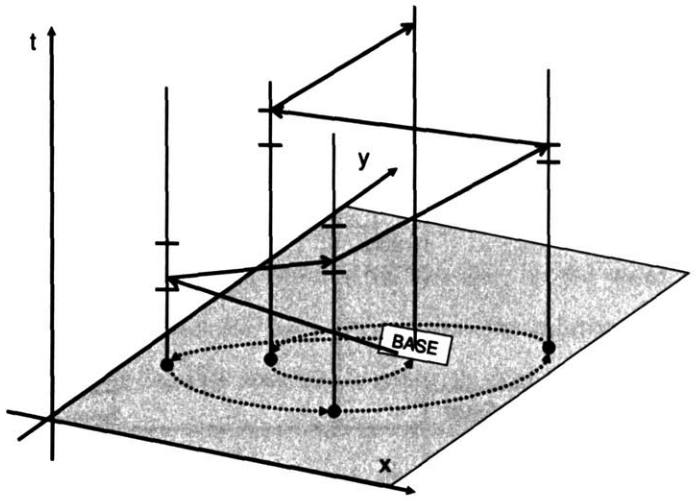

Figure 6-4: A spatio-temporal view of the VRPTW.

with consistent target indexing because sub-problems are generated independently of target indexing for each scenario, but does introduce the problem of having to perform additional computations to identify sub-problems for each problem instance. Although the effectiveness of this method for defining sub-problems has not been empirically evaluated, it should perform well, since, intuitively, "good" missions should correspond to a cluster of spatially and temporally "close" targets.

6.3 Adapting DTMP for the VRPTW

As stated above, solving the exact MDP formulation of the VRPTW quickly becomes computationally intractable as the number of sub-problems increases. For this reason, DTMP is used, as before, to learn a policy for conditionally selecting computations to perform. Under the condition that the variability of sub-problem computation times is low (i.e., it takes approximately the same amount of time to solve a specific sub-problem regardless of the problem instance), the time it takes to solve any subset of sub-problems can be determined explicitly by the particular set of sub-problems solved. That is, if solving sub-problem sp_i takes t_i seconds and solving sp_j takes t_j seconds, then it is implicit that the information state of having solved for sp_i and sp_j results in a total elapsed time of approximately $t_i + t_j$. Essentially, the depth of the decision tree can be used to determine how much time has elapsed, obviating the need to explicitly represent time as a feature.

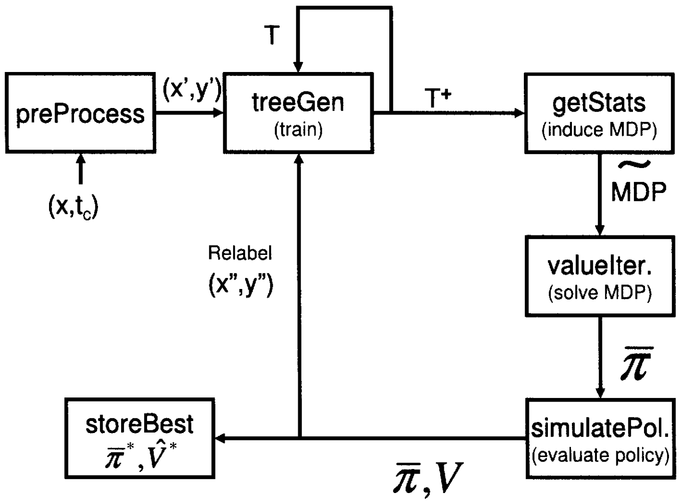

Figure 6-5: The DTMP algorithm modified for the VRPTW. The main modifications are the addition of a preprocessing step, mini-tree generation, and a change to the way the induced-MDP is created.

However, in the case where there is a wide variation in computation times, the information state of having solved sp_i and sp_j in a short amount of time differs significantly from having taken a long amount of time. In the former case, there may be sufficient time to perform additional computations, whereas, in the latter case, the best action may be to execute. Without accounting for time, both information states are treated as the same state and assigned an "averaged" action, perhaps leading to reduced performance. In the remainder of the discussion, it is assumed, and can be empirically verified, that the variability of sub-problem computation times is low, so that time is implicitly defined by the information state.

There are three main modifications to DTMP that have been developed to help it deal with the additional complexity posed by the VRPTW. The remaining steps of the DTMP algorithm remain unchanged.

- The first modification is the addition of a preprocessing step to filter the initial training data.
- The second modification makes use of the mini-trees discussed in Chapter 4.
- The third modification is a change to the structure of the induced-MDP and its subsequent pruning.

Figure 6-5 (cf. Figure 4-5) shows the DTMP algorithm modified for the VRPTW. The *preProcess*() function takes as input, $(\mathbf{x}, t_c)$, where $\mathbf{x}$ is the set of sub-problem costs for each problem instance and t_c is the vector of expected computation times for solving each sub-problem. Preprocessing produces training data in the familiar form of $(\mathbf{x}, y)$, where y is the vector of labels for the training instances. This data used in training to generate a decision tree, T, which is referred to hereafter as the *main regression tree* in order to distinguish it from the mini-trees. Recall that the mini-trees were introduced as an approach to generating feasibly executable plans to deal with the issue of insufficient tree depth. In particular, mini-trees are used to replace the greedy path completing action that is typically added by *getStats*() to each state of the induced-MDP. The mini-trees are generated by "classifying" the training data such that each data instance is associated with a leaf of the main regression tree. The data instances in each leaf are augmented as described in Chapter 4 and trained to generate a mini-tree. The composite decision tree that is generated as a result of appending each mini-tree to the corresponding leaf node of the main regression tree is denoted as T^+ in Figure 6-5. As an alternative, a mini-tree can be generated for all nodes (information states) of the main regression tree as well. How the MDP pruning step is performed depends on whether mini-trees are generated for each information state of the main regression tree, T, or only for the leaf nodes. In the latter case, pruning only affects the main regression tree, while pruning in the former will consider the entire composite tree, T^+.

6.3.1 Preprocessing

The purpose of the preprocessing step is to ensure that the decision tree learns to select mission combinations that yield high value, but can also be feasibly executed. There are two reasons for preprocessing the training data. The first is to reduce the number of sub-problems used for training the decision tree, and the second is to ensure that the set of missions used for training yields feasibly executable plans. The set of sub-problems used to train the metalevel controller for time-critical targeting problems is filtered such that missions that take an excessive amount of time to solve are not included as features in the set of training samples. Here "excessive" refers to the condition where the expected computation time, t_c in Figure 6-5, of a mission is greater than its *latest start time*, *ls*.

When this is true, the mission cannot be successfully executed because it will have violated at least one target's time deadline, t_e. For instance, the sub-problem which consists of the entire set of targets is typically caught and filtered in the preprocessing step because many targets may have their deadlines exceeded by the time planning for the mission has completed. The remaining set of missions, which are feasible missions on their own, are subjected to an additional round of filtering, wherein all combinations of missions (which satisfy global constraints) are examined for each training data instance. Since the global mission consists of a combination of missions, where each mission is generated sequentially, the total computation time of generating every mission within a combination is compared to the latest start time of each mission within the combination. Again, if the total computation time violates any of the latest start times of the individual missions, that combination is filtered. This process is repeated for each instance of the training data.

If a combination allows for feasible execution, the value of executing this combination and the corresponding missions are stored in memory. After examining all combinations of missions, the total mission value of the *winning combination* (i.e., the one resulting in the highest total mission value, while remaining feasibly executable) is added as the label for that training data instance. This is performed for each instance of the training data, such that sub-problems that are part of the winning combination for any data instance are included in the list of *eligible sub-problems* over which to train the decision tree. At the end of preprocessing, each instance of training data for the decision tree consists of a "label" determined by the total mission value for the winning combination and a feature vector which contains sub-problem costs for only the set of sub-problems found on the eligibility list.

Sub-problems that are found on the eligibility list are already known to be feasibly executable when combined with other sub-problems. This will help to reduce the probability of the decision tree learning to select a set of infeasible mission combinations. Without this step, however, the decision tree might erroneously learn to only select high valued mission combinations leading to missions that cannot be feasibly executed after accounting for computation time. Preprocessing was the main mechanism used to generate the results for the experiments in Section 6.5.

6.3.2 Mini-trees

Recall from Chapter 4, learning a decision tree for planning can yield a tree that produces good predictions of the expected cost of the best plan without actually having solved enough sub-problems to generate a feasibly executable plan. To address this problem of insufficient tree depth, it was noted that training data, classified at each of the leaf nodes, could be used to generate small decision trees, mini-trees, to complete the plan. Rather than taking the entire row of sub-problem features in the data for training, as is done for training the main regression tree, a restricted set of sub-problems is used, essentially consisting of the one's that produced the best plan for each training instance. These sub-problems are identified, for each training instance of a leaf node of the main regression tree, by isolating the set of sub-problems used to produce the label (plan cost or value) for that training instance. This set of sub-problems is compiled, and, along with the original label for each data instance, is used as training data for the mini-trees. Each leaf node must have its mini-tree individually generated, since the data being used is unique to that node. The details of mini-tree generation may be found in Section 4.4.

6.3.3 Pruning the Induced-MDP for VRPTW

Although the cost of computation is not explicitly known as in Chapter 3, the MDP pruning step of the main regression tree remains relatively unchanged from that described in Section 4.3. Recall that the pruning step takes the learned decision tree and uses value iteration to determine when execution should occur in each branch of the tree by weighing the benefit of further computation against the cost of doing so. Even though the cost of computation is not explicitly represented, the benefit of additional computation can still be inferred from the reward the agent receives from taking the execute action. For instance, the agent might learn that performing the first five computational actions yields high global mission value, but taking the sixth action yields a much lower value, perhaps because many targets in the best global plan become unavailable after the sixth computation. In this example, the pruning step would remove the sixth and all subsequent computational actions from that branch of tree. However, it may be the case that the seventh computational action may somehow yield an extremely high value plan. This, in turn, would be reflected up the decision tree so that the sixth computational action will have value and therefore will not

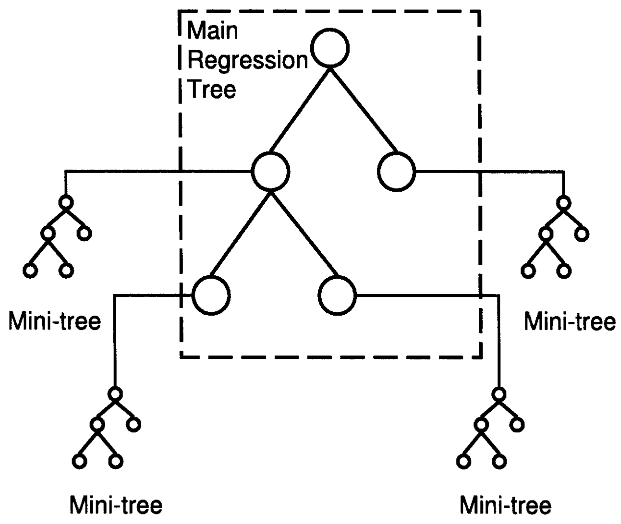

Figure 6-6: Composite decision tree with mini-trees at each node. Mini-trees are generated for each node of the main regression tree. The actions available at each node of the main regression tree are either to continue following the regression tree, to follow the attached mini-tree, or to execute the current plan. The actions available at each mini-tree node are to either continue following the mini-tree, or execute the current plan.

be pruned.

Pruning can occur in two ways depending on the manner in which the mini-trees are appended to the main regression tree. By default, mini-trees are assumed to only be attached to the leaf nodes. In this case, pruning should primarily focus on the main regression tree, since the mini-trees might completely be pruned away if pruning were performed in a bottom-up manner as described in Chapter 4.

The first approach to pruning involves a coarse adjustment of a tuning parameter in the decision tree learning algorithm that adjusts its size. Recall that in Chapter 4, the variable, *splitmin*, in the *treeGen*() algorithm, specifies the number of training samples needed at a node to warrant a split. This variable can used to control the size of the main regression tree learned from data. Large values of *splitmin* yield more compact decision trees, since each node of the decision tree must contain many more data points in order for a split to occur. A search over values of *splitmin* can be performed to generate a main regression tree and its associated mini-trees. The best sized tree can be determined through trial and error by evaluating each one's performance and storing the best.

The second, more principled, way to perform pruning is exactly the same as that described in Chapter 4, except that it considers the entire composite decision tree for pruning and each node of the main regression tree has an associated mini-tree. Figure 6-6 shows such a composite decision tree. The main regression tree is generated to full depth and a mini-tree is trained for each node, treating it as though it is a leaf node. The induced-MDP created by the composite tree can be solved using value iteration, where at each node in the main regression tree the choice of action is either to continue down the main regression tree, divert to the corresponding mini-tree, or execute the current best plan. The choice of action for each mini-tree node is to continue down the mini-tree or execute the current best plan. In this case, bottom-up pruning is acceptable since the plan-completing mini-trees are available at each node rather than being restricted only to leaf nodes. The solution to the MDP will naturally prune away the set of sub-trees (portions of the composite tree) that do not yield value. This method of pruning is more expensive than adjusting *splitmin*, since is requires dealing with a much larger composite decision tree. Its advantage is that it offers a finer grained control over which branches of the composite tree to prune.

For the experiments of Section 6.5, a two-level composite decision tree was used, where mini-trees were only appended to the leaf nodes. Initially, a shallow main regression tree is generated, using preprocessed data as discussed above. The training data is then "classified", such that leaf nodes of the main regression tree are associated with each problem instance in the training sample. The set of training samples in each leaf node of the main regression tree is augmented and used to train plan-completing mini-trees as described in Subsection 4.4.2. The bottom-up MDP pruning step was not used to prune the composite tree, as *splitmin* was already set to a large value, resulting in a compact main regression tree. Preprocessing limited the number of sub-problems used in training, resulting in very shallow mini-trees as well. Without pruning, the metalevel policies simply consisted of performing the computations suggested by the composite decision tree until a leaf node was encountered. The best mission combinations were then generated using the set of solved sub-problems. In addition, the metalevel policy was generated using only a single pass of DTMP and was not iterated. Referring to Figure 6-5, the results of Section 6.5 were generated using only a restricted portion of the modified version of DTMP. DTMP was halted as soon as T^+ was obtained, and used directly as the metalevel policy. At present, both the effect of iterating on this modified version of DTMP as well as the effect of generating mini-trees for each node of the main regression tree have not been fully explored. Additional experiments will need to be performed in the future to test their properties.

6.4 Multiple Traveling Salesmen Results

Before to presenting the VRPTW results, the results for a vehicle routing problem with no time windows are given. The reason for doing so is to show that DTMP can be applied without modification for the version of the problem where there are no hard temporal constraints and performs well. This is in contrast to the additional steps, discussed in Section 6.3, needed by DTMP for the version with temporal constraints. This problem is simply a variant of the standard traveling salesman problem, where the goal is to assign a fixed number of homogeneous salesmen to cover all cities. The base-level objective is to maximize the total value gathered by all salesmen minus their travel costs. The constraints are that each salesman starts at the same location, is limited to a maximum travel distance (e.g., limited due to fuel), and must complete a Hamiltonian tour [37]. In addition, no location can be visited by more than a single salesman. The absence of time windows keeps the costs stationary, similar to time-separable problems of Chapter 5. The cost of time is assumed to be additive.

This problem can be separated, as before, into sub-problems, where sub-problems are described as standard TSPs with no time windows. A sub-problem consists of a specific assignment of a subset of locations to a particular salesman and the solution[2] is a standard TSP tour over those locations. An optimal planner, which solves all $2^{|L|}$ sub-problems, where $|L|$ is the number of locations, and generates the final plan using an integer program (IP) to select the optimal combination of tours respecting the visit-once constraint, was chosen to serve as a baseline for comparison.

The greedy planner, used for comparison was implemented by alternately assigning the salesmen to the nearest unassigned location while ensuring that each salesman respected the travel-distance constraint and was not optimized in any way. Note that the nature of the greedy algorithm in this case refers to being greedy with respect to the distance to the next target added to a vehicle's mission and not with respect to computation time. This can be contrasted with the graph problems of Chapter 5, where a greedy (least computation cost) plan was generated by producing a directed path to the goal for high costs of computation.

[2]Not all assignments result in feasible sub-problems due to the travel distance constraint.

ϵ	Optimal	DTMP	Greedy
0.0	53.9704	53.2449 $\pm$ 0.763	46.948
1.0	38.451	52.197 $\pm$ 0.777	46.948
9.0	-85.697	52.195 $\pm$ 0.777	46.945
1E2	-1.5E3	52.095 $\pm$ 0.775	46.905
1E4	-1.6E5	50.005 $\pm$ 0.794	42.611
1E5	-1.6E6	43.330 $\pm$ 0.895	3.569
1E6	-1.6E7	8.272 $\pm$ 1.166	-3.9E2

Table 6.1: Results for the multiple traveling salesman problem.

This particular greedy algorithm was selected over another, which minimizes the time to compute a solution, since it seems to more reasonably resemble a "fast heuristic" that might be implemented by a human to solve the multiple TSP problem.

The problem scenario consists of 4 salesmen and 8 locations. The locations were separated into groups of two, such that each group was assigned to one of the four quadrants over a square region delineated by lower left coordinates (0,0) and upper right coordinates (2,2). The home base was centered at (1,1). The travel-distance constraint was set to 2 for each salesman using the Euclidean measure of distance. Problem instances consisted of independently sampling locations uniformly over their respective quadrants. The illustration in Figure 6-3 shows the setup for this problem. Table 6.1 gives the results for this problem.

From Table 6.1, it is clear that DTMP is competitive with the optimal strategy when computation costs are zero as in previous results for shortest path planning. Note that for this problem, wall-clock time was used in calculating the cost of computation rather than counting the number of discrete operations as in previous examples. However, the computational cost multiplier ϵ is still employed in reporting results, where the objective of the metalevel problem is to maximize the expected value of the executed missions minus ϵ times the time elapsed from solving sub-problems. The computational cost multiplier is included to represent the cost of per unit time of the resources that are consumed during the planning process.

Each target is assumed to be worth 10 utility points, and the value of a mission is computed as the total target value of the mission subtracted by the travel costs involved in executing the mission. Atypical for this example is the fact that increasing values of ϵ did not result in DTMP converging to a the greedy solution. This is due both to the plan completion algorithm, which selected computationally inexpensive sub-problems to generate the remainder of the plan and to the fact that the greedy algorithm presented here is not greedy with respect to computation time as explained in above. As a result, the greedy strategy performs well, but still incurs a very small computation time which is compounded by the cost of computation multiplier. As the value of ϵ increases, the computation cost begins to have more effect on the total cost. This is the reason that the value of the greedy algorithm eventually becomes negative.

6.5 VRPTW Results

This section presents the results of the VRPTW for a problem scenario with two vehicles and nine targets. Although this is a relatively simple problem scenario, it serves to illustrate the value of the DTMP algorithm in the case of hard temporal constraints because it possesses all of the problem features of a more complicated problem, but, due to its size, allows for the resulting DTMP missions to be easily interpreted in order to determine whether the algorithm is performing reasonably. Each target has an associated time window that is assumed to be fixed across problem instances. Eight of the nine targets are assumed to be of equal value, fixed in location across problem instances, and known *a priori* to the planner. It is also assumed that there are pre-planned initial missions that are executed by each vehicle at the beginning of each planning episode. Individual problem instances are generated by varying the location of the ninth target (pop-up target). This ninth target is known as the *emergent target* as its location is not known *a priori*. It emerges randomly over a given region partway through plan execution. This problem may, at first, appear to be different from the development of the basic VRPTW problem in Section 6.2. However, the difference is minor, since, after the emergent target appears, the basic VRPTW problem with nonhomogeneous vehicles is recovered with the exception of a side constraint which forces to global plan to include a mission which prosecutes the emergent target before its deadline.

The initial scenario is presented in Figure 6-7, which shows the vehicles as triangles, the known targets as circles, and the emergent target (shown here only for illustration purposes, as it is not initially known where it will appear) as a square. Note that in Figure 6-7, only eight of the targets are labeled[3]. At the start of the scenario, each vehicle is assumed to

[3]For this problem, the emergent target is assumed to always appear at a fixed time, but at random locations within the dashed region. In general, a policy can be learned for random emergence times. The first four targets in vehicle V2's default plan will have been prosecuted by the time the emergent

be executing its pre-planned initial plan, indicated by lines in the figure, which completes a Hamiltonian circuit. The direction of travel of each vehicle is indicated by an arrow. The dashed region is the area in which the emergent target will appear. The emergent target is used to represent a target of high importance, and the planner is constrained to formulate a plan to prosecute it in a timely manner before it disappears. It is assumed that one of the two vehicles must be reassigned to include the emergent target in its new plan. This implies that some of the targets that were originally in a vehicle's initial plan may need to be discarded to satisfy this constraint. The only change to the master level problem formulation is the additional constraint, given by Equation 6.7, that the emergent target be must be included in a single mission,

$$\sum_{k \in VEH} \sum_{i \in MISS_k} m_{iETk} x_{ik} = 1, \tag{6.7}$$

where the index ET indicates the target index of the emergent target.

Prior to the appearance of the target, the vehicles are executing their initial plans. The mission generation problem begins immediately after the emergent target appears. The objective is to efficiently generate a plan (a mission for each vehicle) such that the emergent target is prosecuted while covering as many of the remaining targets as possible by both vehicles. As before, a balance must be struck between the utility of planning against its cost (in the form of exceeding deadlines).

Only a DTMP metalevel policy has been generated for this scenario. The solution of the metalevel MDP formulation is computationally intractable as it is even more computationally difficult than the metalevel MDP formulation in the case of the shortest path problem. Figure 6-8 shows the results of the new plans generated by DTMP. The figure depicts a snapshot of the current scenario at the point in time when the emergent target appears. Some time has elapsed since the start of execution of the initial plan. By this time vehicle V2 has prosecuted the first four targets, indicated by stars, in its initial plan, and vehicle V1 is loitering above target 3, waiting for it to become available. The new missions that are generated to account for the emergent target are shown in the diagram by the new line segments. Again, it is assumed that planning must be completed prior to execution. During planning, each vehicle is assumed to be loitering over its current position. At the

target appears and are not labeled.

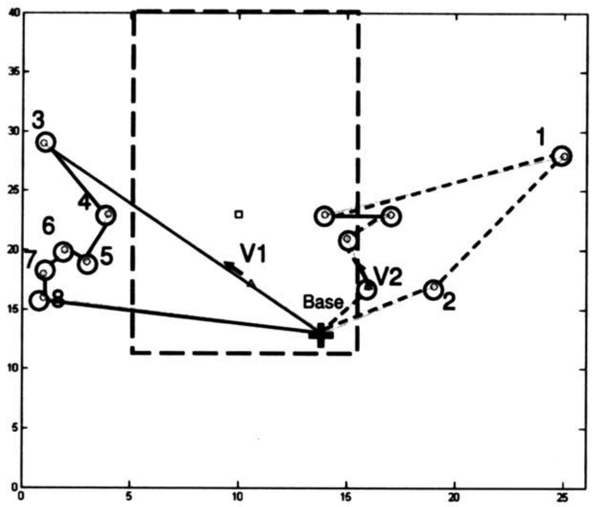

Figure 6-7: The initial nine-target VRPTW scenario. Targets are indicated by circles and eight of the ones that will remain after the emergent target appears are numbered. The two vehicles, labeled V1 and V2, are indicated by triangles, and the directions in which they are traveling are indicated by arrows. The base is indicated by the cross (center). The line segments connecting the targets represent the current plan being executed by each vehicle. The square is used to indicate an emergent target, which is shown here for illustration purposes only, as the location at which it will appear is not known initially. The dashed region represents the bounded region where it can appear.

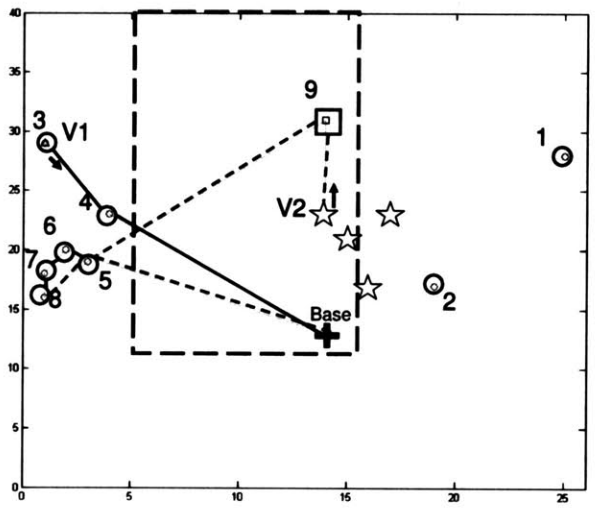

Figure 6-8: The new plans generated by the DTMP metaplanning policy as a result of the appearance of the emergent target. Some targets cannot be covered under the constraint that the emergent target must be prosecuted by one of the vehicles. The targets that have already been prosecuted by vehicle V2 are indicated by stars.

end of planning, for this scenario, vehicle V1 has been reassigned to targets 3 and 4, while vehicle V2 has been reassigned to target 9, the emergent target, and targets 5, 8, 7 and 6. In this case, targets 1 and 2, originally part of vehicle V2's initial plan, are abandoned because their time window deadlines cannot be satisfied under the constraint that the emergent target be prosecuted by one of the two vehicles.

For this problem realization, it appears that sending vehicle V2 only to the emergent target and allowing vehicle V1 to continue with its default plan is the most sensible course of action. However, it should be noted that, in this implementation, all missions are generated from scratch when the emergent target appears, so that prior plans are always "discarded" and must be regenerated if they are to be used. For this example, it is likely that the total computation time involved in generating a six-target mission, containing targets 3, 4, 5, 6, 7 and 8, along with a single-target mission, for target 9, consumes more time than generating a two-target mission, for targets 3 and 4, and a five-target mission, containing targets 9, 5, 8, 7, and 6. In the former case, excessive computation time might have prevented the set of missions from being feasibly executed, while in the latter case, planning time was sufficiently fast to allow for feasible execution. Also note that Figure 6-8 shows but a single realization of the appearance of the emergent target. For other realizations, it is possible that vehicle V1 is reassigned to cover the emergent target.

Along with DTMP results, the results of an anytime algorithm developed for the VRP, called ANYTIME VRP, are also reported as a baseline for comparison. This algorithm is simple and works by randomly selecting a fixed number of sub-problems, which are then solved and given to a master level IP to generate a complete plan. It is iterative in nature, where, during each iteration, additional sub-problems are randomly chosen to be solved and added to the pool of missions available to the master level. The master level generates a new plan considering the entire pool of missions, resulting in plan improvement. Figure 6-9a shows a set of instantiated performance profiles for this algorithm along with its expected performance profile shown in Figure 6-9b. The performance profiles plot the object-level utility of the algorithm on the y-axis against the time spent in computation, in seconds, on the x-axis. Recall that object-level utility only accounts for the utility of a plan without regard to the effect of computation time. Targets are each worth 10 utility points, and traveling between targets incurs a travel cost proportional to the distance traveled.

As in ARA*, ANYTIME VRP can generate a range of solutions as a function of computation

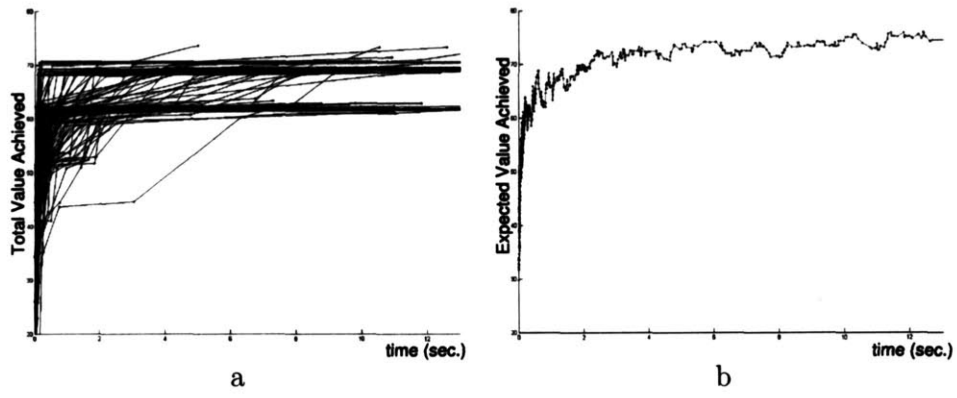

Figure 6-9: Performance profiles of ANYTIME VRP, where total plan value is plotted against computation time. A sample of performance profiles over 100 problem instantiations is shown in a, while b shows the average or expected performance profile.

time. DTMP is compared against ANYTIME VRP for short, medium and long computation times. These computation times correspond to stopping after the first feasible solution is found, stopping at an intermediate time where the rate of improvement "flattens out" and stopping after fully computing all sub-problems, respectively. From the performance profiles, it can be seen that twelve seconds is all that it takes to run ANYTIME VRP to completion. This time scale for computations is not commensurate with the time scale of the time windows in the scenario, which is given in terms of minutes. This implies that computation time for this problem does not have real-world consequences, primarily due to the reduced complexity of solving sub-problems for so few targets. For this reason, a multiplication factor on the amount of time spent is introduced, and acts similarly to ϵ. This factor, in effect, makes sub-problems appear to be more computationally difficult.

The results of ANYTIME VRP and DTMP for multiplication factors of 1, 10 and 60 are given in Table 6.2. When the multiplication factor is 1X, then the dynamics of solving sub-problems are unchanged from the original, and complete planning should perform best. The expected value of DTMP is competitive with the LONG computations, which is the objective of bounded optimality. However, for particular problem instances is it possible for the ANYTIME VRP to do much better, as evidenced by the performance profiles in Figure 6-9.

As the act of solving sub-problems becomes more difficult, the complete planner, LONG, begins to lag behind the other strategies. This is because the amount of time spent com-

ϵ	Long	Medium	Short	DTMP
1X	64.465 ± 1.243	59.785 ± 0.909	37.665 ± 1.228	64.9654 ± 0.944
10X	4.372 ± 4.637	52.114 ± 1.100	35.527 ± 1.135	64.7324 ± 1.177
60X	1.9345 ± 2.097	42.302 ± 1.682	37.276 ± 1.067	63.533 ± 1.359

Table 6.2: Results for the vehicle routing with time windows problem, where LONG, MEDIUM and SHORT represent the results of ANYTIME VRP under long, medium and short computation times.

puting sub-problems starts to near the time scale of minutes. This results in real-world consequences, as many of the targets are no longer available after deliberation. Under the 10X condition, the MEDIUM computation time case (stopped computation after 0.5 seconds) is the only one that is competitive with DTMP.

From the results, it appears that the SHORT computation time case is invariant to increases in sub-problem difficulty. This is because it outputs a solution as soon as one is available, regardless of quality. When sub-problems take a long time to solve, represented at the 60X condition, then both the long and medium computation strategies feel the effect, resulting in reduced performance. DTMP maintains its competitive advantage because it has learned to quickly select the high utility sub-problems to solve.

The VRPTW is a challenging problem on its own. Solving for the optimal metalevel controller in this domain is an even more difficult problem, but the results presented here illustrate of the viability of using DTMP to learn good metalevel policies.

6.6 Chapter Summary

This chapter discussed how DTMP might be adapted to handle planning problems with hard temporal constraints. These problems differ significantly from the set of problems presented in Chapter 5 due to the fact the time constraints make it difficult to express the cost of computation as a stationary function. Instead, the cost of computation and the value of sub-problems is function of the planning time. In order for the metalevel controller to perform well, the metaplanning problem to be solved must reason about the possibility of losing the value associated with being able to execute a currently planned mission against the benefit that might be gained by spending additional time to plan new missions. To do so, a feature representing the current time should ideally be added to the set of features for decision tree training. However, under conditions where sub-problem computation times have low variability, the elapsed time due to computation can be inferred from the current information state, making the explicit representation of time as a training feature unnecessary.

The results that have been presented are the outcomes of preliminary experiments on a restricted version of the modified DTMP algorithm whose full characteristics remain to be analyzed. The modified version of DTMP includes a preprocessing step and an adjustment to pruning the induced-MDP. The preprocessing step is necessary to limit the number of sub-problems to be considered during decision tree training as well as to improve the chances for the decision tree to generate feasibly executable policies. Two methods were discussed for performing the MDP pruning step. The first involves directly controlling the size of the decision tree at the training stage, and the second involves performing value iteration on a much larger decision tree. The results generated for the VRPTW heavily relies on the preprocessing step to restrict the set of eligible sub-problems for training. However, the preprocessing step may be too conservative in the sense that the set of sub-problems considered for training is severely restricted. Allowing DTMP to be trained on a less restricted set of sub-problems may be beneficial, but more investigation will be needed to determine how to do so.

While the results presented for the VRPTW metalevel planning problem exclusively addresses only time window constraints, additional side constraints typical of vehicle routing problems, such as capacity constraints, service times, and nonhomogeneous vehicles can also be tackled given a good enough problem decomposition. The success of the results is highly dependent on the problem decomposition (definition of sub-problems), and some ways for bolstering the problem decomposition chosen for this problem were also discussed.

The modifications to DTMP should allow it to address a more general form of the VRPTW than presented in this chapter, where the randomness is attributed only to the probabilistic appearance of the emergent target. A more general metalevel control problem might consist of a metalevel environment where the targets are probabilistically distributed in space, while their corresponding time windows are probabilistically distributed as well. Under these conditions, DTMP, along with a good way of defining sub-problems, should be able to learn the expected costs and benefits of computing sub-problems. The key to learning a good metalevel policy is, as always, to weigh the cost of computation against its benefit.

The next chapter concludes this thesis with a summary of thesis contributions along with suggestions for future work.

Chapter 7

Conclusion

This chapter summarizes the contributions of this thesis and discusses avenues for future research. The focus of the work in this thesis has been to examine the problem of decision making under limited computational resources. It has been shown that bounded optimal decision making can be accomplished through the direct metalevel control of computational actions. The availability of a closed-loop metalevel controller with atomic computational actions offers advantages over previously established complete decision procedure approaches.

This thesis began by formally defining the metalevel control problem as an MDP. Although there are algorithms for solving the metalevel exactly MDP, the size of the state space grows exponentially with the number of computational actions or sub-problems, making them computationally intractable. DTMP, an alternative solution based on approximate policy iteration combined with decision tree learning was developed to learn metalevel controllers for substantially larger problems. This heuristic solution takes advantage of problem structure to effectively prune the state space resulting in a much smaller MDP model.

Experimental results were generated to demonstrate the bounded optimality of the metalevel controllers developed through exact means, and used to show that DTMP-generated controllers result in comparable (though not bounded optimal) performance. When problems were too large to be solved exactly, DTMP is shown to perform better than alternative approaches. In the next section, each of these will be discussed in more detail

7.1 Summary of Thesis Contributions

The thesis contributions towards developing agents that exhibit bounded optimal behavior are as follows

- The model of the discrete metalevel control problem in terms of Markov decision processes for both time-separable and non-separable problems.
- The DTMP approximation algorithm for heuristically generating metalevel control policies.
- Experimental results verifying the bounded optimal behavior of the metalevel controllers developed in this thesis for a variety of problem domains. Bounded optimality is automatically achieved by the optimal policies generated from solving the metalevel MDP formulation exactly. DTMP policies are shown to perform comparably to optimally generated policies when direct comparisons are possible. DTMP allows for metalevel policies to be generated for much larger problems, and these policies perform favorably in comparison other competitive strategies.

The first contribution is the modeling of the metalevel planning problem as a sequential decision making problem in information space. This formulation was able to accommodate three important problem aspects: 1) the manner in which the master level utilizes the information provided by the sub-problems, 2) the functional relationship between sub-problems (problem decomposition), and 3) the cost of time. The MDP formulation of the metalevel problem, as discussed in Chapter 3, is given for the time-separable case. In Chapter 6, time was added to the set of state variables to handle problems that are not time-separable.

The discrete MDP formulation of the metalevel problem, where the cost of computation is separable and additive, encompasses the work of Russell and Wefald [42]. In addition, the MDP formulation can also be used to obtain the optimal satisficing results of Simon and Kadane [44] as well as the results of Etzioni [18].

The utility of the metalevel controllers is especially significant in situations with the occurrence of similar episodic planning instances that have enough variability such that each must be solved in real-time. The solutions to these problems are compiled policies that serve to orient the plan generation process towards the efficient selection of computational actions. These metalevel policies are important when the cost of computation plays a major part of the agent's objective function.

The second contribution, as discussed in Chapter 4, is the DTMP algorithm, a heuristic approach for learning good metalevel controllers. One of the main benefits of DTMP is the generation of good policies while maintaining an economic representation of the relevant state space. Although it is possible for the size of a decision tree to be exponential in the number of information states, this is not often experienced, unless no inherent problem structure can be learned by the decision tree. Learning to exploit problem structure leads to vastly reduced state spaces for solving the metalevel planning problem with little effect on the empirical performance of the resulting policy. As discussed previously, decision trees have been used for state abstraction in the literature.

The most interesting aspect of their use in this thesis is that, in addition to state space abstraction, they also serve as the structure upon which the metalevel policy is built. As such, their added benefit, apart from state space reduction, is the reduction of the action space. Rather than selecting from $n+1$ actions per state, as in the exact MDP formulation, where n is the number of sub-problems, the number of actions is resulted to just two actions per state, compute or execute.

The last contribution is the set of experimental results, which includes a simplified version of a real-world example for time-critical mission planning formulated as a vehicle routing problem with time windows. Experiments show that in cases where the planning problems are small enough in size and the planning costs are separable and additive, it is possible to generate metalevel controllers that are bounded optimal. This is a direct consequence of the equivalence of an optimal MDP policy and a bounded optimal program. The approximate policies generated by the DTMP approximation algorithm are also shown to be comparable to the exact solutions. For larger problems, the computation of exact policies is not possible. In these cases, only DTMP results are generated, which compared favorably with the results of other competitive algorithms.

7.2 Future Work

There are many directions to pursue to further extend and generalize the work in this thesis. This section discusses several desirable problem features not directly addressed by the metalevel planning formulation and is left for future work.

7.2.1 Incorporating Anytime Algorithms Into Sub-problem Solutions

The computation times for sub-problems in this thesis have been assumed to take fixed values (based on their expected computation times), such that they behave like run-to-completion algorithms. It would be useful however to add the flexibility of using anytime algorithms to solve sub-problems. With the addition of anytime sub-problems, the metalevel policy would consist of actions that would not only dictate which sub-problems to solve, but how much time to spend computing their solutions. Considering anytime computation at the sub-problem level may offer computational opportunities not available with run-to-completion algorithms. One opportunity may be the ability to curtail the solution of a sub-problem during mid-computation if the rate of improvement is not performing as expected and reallocate computational effort to a different sub-problem. As another example, the metalevel controller may discover that the rate of improvement for a particular sub-problem is higher than expected and decide to allocate additional computation effort to it. One attempt towards accomplishing this may be to combine the work of Zilberstein [51] on anytime compilation with the process of generating metaplanning policies in this thesis.

7.2.2 Interleaving Planning and Execution

One of the limiting aspects of the metalevel problem formulation is that planning must be completed prior to plan execution. An appropriate extension of the metalevel planning problem would be to allow planning and execution actions to be interleaved. Interleaving planning and execution gives the agent the ability to enact behavior of the form of compute-move-compute-move etc., rather than a single episode of compute-move as found in this thesis. This allows the agent to initiate execution prior to the generation of a complete plan. This should allow for more system flexibility and faster response time. The danger with executing prior to the completion of a plan would be the potential need for physically backtracking or, in the worst case, to cause the agent to move to a perilous state from which it could not escape. Currently, there are approaches for real-time planning such as Korf's Real-Time A* (RTA*) [28] and Koenig's Min-Max Learning Real-Time A* (Min-

Max LRTA*) [27] which accomplish the interleaving of planning and execution actions, but do not explicitly consider of the cost of computation. It may be fruitful to adapt the work in this thesis to develop a metalevel controller for these algorithms. Even more interesting would be to develop a metalevel control scheme for continuous planning and execution actions that can occur concurrently.

7.2.3 Dynamic Environment

The formulation of the metalevel planning problem is most applicable to environments that are either static or change slowly enough to not have an appreciable effect on planning. In this thesis, sub-problems plans, once solved, are assumed to hold constant for the remainder of the planning episode. In dynamic environments, this assumption no longer holds.

For example, in the case of the maze domain in Chapter 5, if the obstacles were allowed to roam the maze, the static plan generated as a result of the current metalevel controller will have severely reduced utility. Accounting for problem dynamics is an important feature to add to the current metalevel formulation. A typical approach for dealing with a change in information is to incorporate it by replanning.

Supposing that the effect of a dynamic environment on metalevel planning to change the outcome of a sub-problem. The corresponding sub-problem needs to be replanned to reflect the current changes in information so that the master level can account for them in the generation of a complete plan. Replanning often might incur a significant amount of computational overhead, reducing the overall effectiveness of the agent. One of the problems that will be faced by the metalevel controller will be to determine when replanning is beneficial. If the information update is insignificant, a replan may not be warranted and can be ignored with little penalty.

Designing the metalevel controller to accommodate a dynamic environment will be intimately related to incorporating the interleaving of planning and execution actions. This is because whether the agent decides to execute a portion of the plan will depend on whether it has changed significantly enough to warrant a replan prior to execution. On the other hand, since executing a portion of the plan may result only in local consequences, it may be possible to only consider incorporating local information updates of the environment. As with most problems, the exact details will be problem dependent.

7.3 Concluding Remarks

This thesis has addressed the problem of generating bounded optimal agents through the closed-loop control of their computational actions. Under the restrictions of the particular formulation of the metalevel planning problem, bounded optimality has been shown to be achieved through the generation of optimal metalevel policies. Under time-critical circumstances or in cases where computations are costly, the approaches discussed in this thesis generate economical policies which balance the utility of generating good plans against the cost of generating them. The work of this thesis addresses a small part of the overall problem of bounded optimality. While full bounded optimality in the strict sense is difficult to achieve, it is assured that continued research in this general area will yield further knowledge for developing more advanced algorithms to allow for agents to make more effective use of their system resources.

Bibliography

[1] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin. *Network Flows.* Prentice-Hall Inc., 1993.

[2] A. P. Armacost. *Composite Variable Formulations for Express Shipment Service Network Design.* PhD thesis, Masschusetts Institue of Technology, 2000.

[3] J. A. Barnett. How much is control knowledge worth? A primitive example. *Artificial Intelligence*, 22:77–89, 1984.

[4] K. P. Bennett. Global tree optimization: A non-greedy decision tree algorithm. In *Computing Science and Statistics*, number 26, pages 156–160, 1994.

[5] D. P. Bertsekas. *Dynamic Programming and Optimal Control*, volume 1. Athena Scientific, 2000.

[6] D. P. Bertsekas. *Dynamic Programming and Optimal Control*, volume 2. Athena Scientific, 2000.

[7] D. P. Bertsekas and J. N. Tsitsiklis. *Neuro-Dynamic Programming.* Athena Scientific, 1996.

[8] D. Bertsimas and J. N. Tsitsiklis. *Introduction to Linear Programming.* Athena Scientific, 1997.

[9] M. Boddy. *Solving Time-Dependent Problems: A Decision-Theoretic Approach to Planning in Dynamic Problems.* PhD dissertation, Brown University, Department of Computer Science, 1991.

[10] M. Boddy and T. Dean. Decision-theoretic deliberation scheduling for problem solving in time-constrained environments. *Artificial Intelligence*, 67(2):245–286, 1994.

[11] L. Breiman, J. Friedman, R. Olshen, and C. Stone. *Classification and Regression Trees.* Chapman and Hall/CRC, 1984.

[12] D. Chapman and L. Kaelbling. Input generalization in delayed reinforcement learning: An algorithm and performance comparisons. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI 91)*, pages 726–731, 1991.

[13] V. Conitzer and T. Sandholm. Definition and complexity of some basic metareasoning problems. In *Proc. of International Joint Conference on Artificial Intelligence (IJCAI 2003)*, pages 761–764, 2003.

[14] T. Dean. Decision-theoretic control of inference for time-critical applications. Technical Report CS-89-44, April 1990.

[15] T. Dean and M. Boddy. An analysis of time-dependent planning. In *Proceedings of 7th National Conference on Artificial Intelligence*, pages 49–54, 1988.

[16] Y. Dumas, J. Desrosiers, E. Gelinas, and M. Solomon. An optimal algorithm for the traveling salesman problem with time windows. *Operations Research*, 43(2):367–371, 1995.

[17] D. Einav and M. R. Fehling. Computationally-optimal real-resource strategies. In *IEEE Conference on Systems, Man and Cybernetics*, Los Angeles, California, November 1990.

[18] O. Etzioni. Tractable decision-analytic control. In *First International Conference on Principles of Knowledge Representation and Reasoning*, pages 114–125, 1989.

[19] R. Greiner, A. Grove, and D. Roth. Learning cost-sensitive active classifiers. *Artificial Intelligence Journal*, 139(2):137–174, 2002.

[20] T. Hastie, R. Tibshirani, and J. Friedman. *The Elements of Statistical Learning: Data Mining, Inference and Prediction.* Springer-Verlag, 2001.

[21] D. Heckerman, E. J. Horvitz, and B. Middleton. An approximate nonmyopic computation for value of information. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 15(3):292–298, 1993.

[22] J. Hoey, R. St-Aubin, A. Hu, and C. Boutilier. SPUDD: Stochastic planning using decision diagrams. pages 279–288, 1999.

[23] E. J. Horvitz. Reasoning under varying and uncertain resource constraints. In *National Conference on Artificial Intelligence of the American Association for AI (AAAI-88)*, pages 111–116, 1988.

[24] E. J. Horvitz. *Computation and Action Under Bounded Resources.* PhD dissertation, Stanford University, Medical Information Science, 1990.

[25] E. J. Horvitz. Reasoning about beliefs and actions under computational resource constraints. In M. Henrion, R. D. Shachter, L. N. Kanal, and J. F. Lemmer, editors, *Uncertainty in Artificial Intelligence 5*, pages 301–324. Elsevier Science Publishers B.V., North Holland, 1990.

[26] R. A. Howard. Information value theory. *IEEE Transtactions on Systems Science and Cybernetics*, 2(1):22–26, 1966.

[27] S. Koenig. Minimax real-time heuristic search. *Artificial Intelligence Journal*, 129(1-2):165–197, 2001.

[28] R. Korf. Real-time heuristic search. *Artificial Intelligence*, 42(3):189–211, 1990.

[29] K. Larson. *Mechansim Design for Computationally Limited Agents.* PhD dissertation, Carnegie Mellon University, School of Computer Science, 2004.

[30] M. Likhachev. ARA*: Anytime A* with provable bounds on sub-optimality.

[31] C. X. Ling, Q. Yang, J. Wang, and S. Zhang. Decision trees with minial costs. In *In Proc. 21th International Conference on Machine Learning*, 2004.

[32] M. L. Littman. *Algorithms for Sequential Decision Making.* PhD dissertation, Brown University, Department of Computer Science, 1996.

[33] J. S. Malasky. Human machine collaborative decision making in a complex optimization system. Master's thesis, Massachusetts Institute of Technology, Sloan School of Management, 2005.

[34] A. McCallum. Instance-based utile distinctions for reinforcement learning with hidden state. In *International Conference on Machine Learning*, pages 387–395, 1995.

[35] A. Moore. Variable resolution dynamic programming: Efficiently learning action maps in multivariate real-valued state-spaces. In *Proceedings of the Eighth International Workshop on Machine Learning*, pages 333–337, 1991.

[36] S. Murthy. Automatic construction of decision trees from data: A multi-disciplinary survey. *Data Mining and Knowledge Discovery*, 2(4):345–389, 1998.

[37] K. H. Rosen. *Discrete Mathematics and Its Applications.* McGraw-Hill Inc., 1995.

[38] S. M. Ross. *Stochastic Processes.* John Wiley and Sons, Inc., New York, New York, second edition, 1996.

[39] S. J. Russell and P. Norvig. *Artificial Intelligence A Modern Approach.* Prentice Hall, 2000.

[40] S. J. Russell and D. Subramanian. Provably bounded-optimal agents. *Journal of Artificial Intelligence*, 2:575–609, 1995.

[41] S. J. Russell and E. Wefald. Principles of metareasoning. In Ronald J. Brachman, Hector J. Levesque, and Raymond Reiter, editors, *KR'89: Principles of Knowledge Representation and Reasoning*, pages 400–411. Morgan Kaufmann, San Mateo, California, 1989.

[42] S. J. Russell and E. Wefald. *Do the Right Thing Studies in Limited Rationality.* The MIT Press, 1991.

[43] H. A. Simon. *Models of Bounded Rationality.* MIT Press, 1982.

[44] H. A. Simon and J. B. Kadane. Optimal problem solving search: All-or-none solutions. *Artificial Intelligence*, 6:235–247, 1975.

[45] D. E. Smith. Controlling backward inference. *Artificial Intelligence*, 39(2):145–208, 1989.

[46] P. Toth and D. Vigo. *An overview of vehicle routing problems*, pages 1–23. Society for Industrial and Applied Mathematics, 2001.

[47] P. D. Turney. Cost-sensitive classification: Empirical evaluation of a hybrid genetic decision tree induction algorithm. *Journal of Artificial Intelligence Research*, 2:369–409, 1995.

[48] P. D. Turney. Types of cost in inductive concept learning. In *Workshop on Cost-Sensitive Learning at the Sevententh International Conference on Machine Learning*, 2000.

[49] W. Uther and M. Veloso. Tree based discretization for continous state space reinforcement learning. In *AAAI '98/IAAI '98: Proceeding of the Fifteenth National/Tenth Conference onf Artificial Intelligence/Innovative Applications of Artificial Intelligence*, pages 769–774, 1998.

[50] J. von Neumann and O. Morgenstern. *Theory of Games and Economic Behavior.* Princeton University Press, 1947.

[51] S. Zilberstein. *Operational Rationality Through Complilation of Anytime Algorithms.* PhD dissertation, Universtiy of California at Berkeley, Department of Computer Science, 1993.

[52] V. B. Zubek and T. G. Dietterich. Pruning improves heuristic search for cost-sensitive learning. In *Proceedings of the 19th International Conference on Machine Learning*, pages 27–34, 2002.