COMPOSER: A Decision-theoretic Approach to Adaptive Problem Solving

BY
JONATHAN MATTHEW GRATCH
B.A., University of Texas at Austin, 1986

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1993

Urbana, Illinois

INSERT RED BORDER FORM HERE

ABSTRACT

While the general formalization of planning and scheduling problems is NP–hard in worst–case complexity, in practice, for specific distributions of problems and constraints, domain–specific solutions have been shown to perform in much better than exponential time. Unfortunately, constructing and improving custom techniques is a knowledge–intensive and time–consuming process that requires a deep understanding of the application. The goal of our work is to develop techniques to allow for the automatic transformation of a general problem solver into one customized for a particular application. In this thesis we develop a decision-theoretic framework with which to describe the problem. We then present a statistically sound learning algorithm built upon this framework. Our COMPOSER method implements a hill-climbing search through a space of possible problem solving modifications. The learning system attempts to identify a sequence of modifications that improve problem solving performance with respect to the problem distribution and performance requirements for a given application.

The thesis includes two applications that illustrate the technique's power and flexibility. In the first, we show how COMPOSER can improve the performance of the PRODIGY planning system, and that it outperforms several state of the art learning techniques. We then discuss an application of the approach to the real-world domain of scheduling satellite communications. Using problem distributions based on NASA mission requirements, COMPOSER identified strategies that both decrease the amount of CPU time required to produce schedules, and increase the percentage of problems that are solvable within computational resource limitations. We also provide formal proofs of the performance characteristics of the system.

To Dad

ACKNOWLEDGEMENTS

Another damned, thick, square book!
Always scribble, scribble, scribble!

William Henry
Duke of Gloucester

It may seem peculiar, but I always find the acknowledgements to be the most enjoyable section of any thesis I have read; a window into the person behind the work. I am grateful for this opportunity to thank the many special friends and colleagues that helped me reach this point. Of course, there is no way I can fully acknowledge the debt I owe to my advisor, Jerry DeJong. He patiently listened each time I wanted to run away and become a park ranger, anthropologist, *et cetera ad nauseam.* He always treated me as a friend and guided me with many discussions, arguments, and his famed stubbornness. Most importantly he filled me with his enthusiasm and devotion to the "big picture."

Besides Jerry, perhaps the largest intellectual influence on my current work comes from Steve Minton, both through is many writings in the field of machine learning and his comments and enthusiasm. He has also done the field a great service by providing his PRODIGY system to the research community and fostering a climate of intellectual sharing and cooperation. Russ Greiner has had a profound influence through his many interesting comments and suggestions, and by keeping me hard at work through his role as "the competitor." Steve Chien has filled me with enthusiasm when I needed it most. I am glad I've had the opportunity to collaborate with him. He is also a great friend and superb critic.

Illinois is a stimulating place to do research and I benefited from its rich intellectual environment. I am grateful to the past and present members of my research group, Michael Barbehen, Scott Bennett, Steve Chien, Melinda Gervasio, and Dan Oblinger for their knowledge and friendship. As my "cube-mates," Dan and Michael have suffered through the dubious honor of facing every question I didn't have an immediate answer for. I owe Scott a major thanks for providing a superb computing environment. Also thank the many professors and students that filled my environment with stimulating discussion: Tim Chou, Greg Collins, John Collins, Dennis DeCoste, Alan Frisch, Dedre Gentner, Caroline Hayes, Pat Hayes, Larry Holder, Steve Lavalle, Yong Ma, Dave Page, Eduardo Perez, Lenny Pitt, Bharat Rao, Brian Ross, and Larry Watanabe. Anita Govindjee, Cherry Zinger, Hal Render, Greg Rogers, Scott Bennett, Michael Barbehenn, and Dave Page helped make life in the cornfields a lot more fun, as did my favorite haunts Nature's Table (may it rest in peace), Mabel's, the Blind Pig Co., and WEFT 90.1 FM.

Others who helped shaped this research through their many comments and helpful discussions include Colin Bell, Jon Doyle, Oren Etzioni, Pat Langley, Nick Lewins, Adam Martinsek, Ray Mooney, and Alberto Segre. Thanks to Steve Chien, Usama Fayyad, and especially Rich Doyle for giving me the opportunity to spend a summer at JPL. As Steve says, you can't beat a place where you can ski and surf in the same day, and the research 'tain't bad either. This research was conducted at the Beckman Institute at the University of Illinois. Funding was provided by the National Science Foundation under grants NSF-IRI-87-19766 and NSF-IRI-92-09394.

My greatest debt is to my family – to my parents who made me what I am, and to Linda, Nancy, and Aaron who have made my house in Houston such a wonderful place to come home to. To my father, I am glad for all the patience, wisdom, and gentle guidance. To Linda, I am grateful to all of the love you have brought to my dad and me. To Aaron and Nancy, thanks for love and being my brother and sister.

TABLE OF CONTENTS

Chapter **Page**

Chapter **Page**

Chapter		Page

Chapter **Page**

LIST OF FIGURES

Figure **Page**

Figure **Page**

Chapter 1

Introduction

"A round man cannot be expected to fit in a square hole right away.
He must have time to modify his shape."

– Samuel Clemens

Currently, there is a wide gulf between general approaches and effective approaches to problem solving. AI has developed powerful formalisms like non-linear planning or constraint satisfaction. These potentially apply to a wide variety of applications but, in practice, such approaches exhibit poor problem solving behavior. The greatest success has come from custom techniques like expert systems, reactive planners [Miller92, Schoppers92], or specific programs. These approaches are devoted to a single application, require extensive human investment to complete, but often result in a satisfactory solution. When general approaches show success, it is only after extensive domain specific adjustments. The resulting system, while derived from a general technique, bears more resemblance to a custom approach.

In this thesis we develop the argument that individual applications place unique constraints on a general problem solving architecture, and that to achieve high levels of performance, a problem solver must be customized to meet these requirements. However we do not see this as an argument for abandoning general purpose problem solvers (as [Agre87, Maes90]). Rather we recognize the important advantages of such systems. General purpose systems place substantially fewer demands on the system designer. For example, general approaches require only limited information such as "the rules of the game," or domain theory, that people find, if not easy to articulate, at least far easier to articulate than strategic knowledge (for example, see [Schoenfeld82]. Instead we argue that it is possible to provide both the simplicity of general purpose algorithms and the performance of custom techniques. This thesis explores the use of machine learning techniques to automatically customize a general approach to the demands of a specific application.

1.1 APPLICATION SPECIFIC CONSTRAINTS

A specific application places constraints on the the characteristics of any intended problem solver. We characterize these constraints into three aspects. (1) There is some evaluation criterion that measures how well the problem solver performs in the application. This is what is called in operations research the *objective function.* It is an explicit characterization of what it means to succeed at the application. (2) There is some domain of discourse. This is the set of tasks that could arise in the application for which the problem solver is expected to produce solutions. An encoding of the domain of discourse is generally called a domain theory. (3) There is some pattern of task occurrence. This is a characterization of how the tasks arise in the application. For example, some tasks may arise frequently, some not at all and there may be some predictable sequences of task occurrence. Many times we can view this pattern as a random selection according to a fixed probability distribution over the set of possible tasks in the domain of discourse.

The first aspect, the evaluation criterion, is perhaps the least investigated in classical AI problem solving, but has been studied extensively in other areas such as operations research and expert systems. AI has been driven by the combinatorial explosion to focus almost exclusively on computational efficiency as a metric of problem solving success. This metric has been further simplified into a basic question of worst-case tractable vs. intractable inference. While efficiency is an important concern, it is not the only concern for real applications. Tractability is often not the dominant issue in expert system or operations research approaches to problem solving. These systems have been applied to many industrial applications and system developers have been forced to consider many attributes of problem solving behavior in addition to efficiency. For example, scientists who pioneered operations research during World War II included scientists from sociology and psychology to help characterize a suitable evaluation criterion [Taha82 p. 2]. Expert systems researchers have faced such non-efficiency issues as the explainability of their diagnostic systems.

The second aspect, the representation of the domain of discourse, is a central focus in AI. Researchers (have in the past and continue to) investigate the implications of many different representational commitments, each of which supports a variety of more or less general problem solving architectures.

The final aspect is the pattern of tasks that arise in the application. Due to the emphasis on worst-case analysis, distributional information has only been recently emphasized in the problem solving literature (see [Cheeseman89]). Similarly, while such information is central to much of machine learning, it has only recently been a focus in learning for problem solvers. These learning approaches have carefully investigated how to improve performance on individual problems, but they have less carefully considered how these improvements apply across some distribution of problems. We feel that exploiting the constraints embodied in the pattern of tasks is crucial for effective customization and it plays a dominant role in this thesis.

1.2 THE IMPORTANCE OF APPLICATION SPECIFIC INFORMATION

Different applications obviously place different demands on a problem solver. Exploiting such information should lead to a significant improvement in problem solving performance. This is almost an obvious point as ultimately, the metric for successful problem solving is defined by the application's evaluation criterion. Nevertheless, it is useful to explore the many ways application constraints can influence the prospects for a high evaluation. The discussion will focus on methods for improving problem solving efficiency as that is the attribute of problem solving performance chiefly investigated, however similar arguments could be made for other evaluation criteria.

General purpose problem solvers provide general representations that can encode a variety of domains of discourse. It is typically intractable to reason with these powerful representations [Bylander92, Chapman87, Eorl92, Freuder82]. However, this full generality is often unnecessary for a particular application. There are a variety of effective problem solving techniques that apply when a domain is representable in restricted representation. For example, Dana Nau summarizes complexity results for several specializations of STRIPS operators, some of which support worst-case *polynomial* planning. Constraint satisfaction problems can be solved in polynomial time if the constraint graph is expressible as a tree [Dechter87, Freuder82]. Korf shows a domain supports polynomial problem solving if it exhibits the property of *serial decomposability* [Korf87]. Many integer programming programming problems can be efficiently solved in practice by exploiting their structural properties [Fisher81]. Compilation techniques also exploit structural properties (e.g., partial evaluation [Harmellen88]).

Even for worst-case intractable algorithms, there is growing recognition that they can perform well under certain distributions. For example Goldberg suggests that satisfiability problems are frequently solved in $O(n^2)$ time [Goldberg82]. Recent work has focused on characterizing these easy distributions [Cheeseman89, Mitchell92] or devising techniques that can exploit specific distribution information when it is available [Borgida89]. Research into self-organizing systems [Melhorn84 pp. 252–285] and dynamic optimization [Laird92]. exploit the fact that knowledge of the expected distribution of tasks can facilitate a problem solver with substantially better expected performance. A simple example arises in the task of, given a name, searching a linear list of records for one with the same name. Knowing the distribution we can produce substantial improvements by ordering the list of records by their likelihood of being queried.

A sensitivity to application constraints is also apparent when one observes successful problem solving systems, both computational and biological. Expert systems have been successfully applied to a wide variety of real-world applications. These systems are carefully crafted by a domain engineer to apply to a narrow range of problems and their distribution sensitivity can be seen in the so-called "brittleness" of these systems. Evolution is a prime biological example of a distribution–dependent method. The ethologist Konrad Lorenz makes the connection clearly when he states that "the gain in knowledge [through evolution], achieved by trial and error by the genome – which retains what is fittest for it – results ... in the formation of an image of the material world within the living system" [Lorenz78 p. 23]. That organisms are tied to the current particulars of their environment is seen by the fact that changes in that environment may degrade an organism's performance to the point of extinction.

Humans are able to adapt to a wide variety of problem solving situation through their ability to learn. However, here too there evidence from psychology that human performance is "brittle" in much the way of expert systems. The gestalt psychologists demonstrated this clearly with the identification of *einstellung* or functional fixedness. This is the phenomena where people develop fixed methods of solving problems and continue using these methods after the experimenter changes the distribution of problems, even when the old methods are highly sub-optimal in the new environment [Duncker45, Luchins50]. This makes little sense when we look at the solution in isolation, but makes perfect sense when we realize that people frequently perform repetitive tasks and the use of stereotypical solutions can greatly reduce cognitive load. There is also evidence that when people learn, the knowledge is far less general than previously supposed. Ross and Medin show that when people generalize from examples they are often reluctant to generalize, and their learn concepts contain many specific details of the training examples [Medin89].

We suggest that taken together, the limitations of application independent techniques, the power of application specific techniques, and the apparent application specificity of successfully computational and biological problem solvers, make for a compelling argument for using application specific information in the design of problem solvers.

1.3 THE CUSTOMIZATION PROBLEM

Any application imposes some criterion on what is to be considered a successful problem solver. We have argued that a problem solver must be customized to perform well with respect to the evaluation criterion, taking advantage of any constraints that arise from the application domain and pattern of tasks. We would also like to free the system designer from the extensive study and experimentation involved in uncovering and exploiting such information. This suggests the use of machine learning to automatically perform the customization process.

Abstractly, we define the *customization problem* as the learning problem of transforming a general purpose problem solving system into one customized to a particular application. Specifically, we require (1) a general purpose problem solving system, (2) a encoding of the application's domain of discourse for suitable to the problem solving system, (3) an explicit representation of the application's evaluation criterion, and (4) an explicit representation of the application's pattern of tasks, which might be a representative sequence of training examples. Given these four inputs, the customization problem is to construct a modified problem solver that maximizes the evaluation criterion.

One should note that this definition says nothing about how learning should proceed. The goal of maximizing performance should be considered with caution. For many applications there is a provably optimal problem problem solver, but it may be a very difficult problem to construct it. In other cases there is no known optimal solution and an optimal solution may not even exist. Thus the performance of a implemented customization system will likely be more modest – improve the performance of the general problem solver, or maximize its performance with respect to some finite set of possible modifications.

There exist a variety of machine learning techniques that modify problem solving behavior. These algorithms have typically been directed towards specializations of the customization problem. For example, *speed-up learning* is a restriction such that the evaluation criterion is a function solely of computational efficiency. Learning techniques have gradually evolved from simply addressing domain constraints to considering both the domain and pattern of tasks. Concern about domain representation can be traced back at least as far as Mostow [Mostow81] who introduced the term *operationalization* to refer to the process of re-expressing correct but non-useful domain knowledge (for Mostow, high-level advice from an expert) into a useful form. Mitchell, Keller, and Kedar-Cabelli [Mitchell86] adapted Mostow's notion to form their *operationality criteria*, a syntactic criteria for assessing the effectiveness of re-expressions of domain knowledge. Steve Minton's PRODIGY/EBL system extended this structural approach to take advantage of the pattern of tasks as they arise in the domain. His PRODIGY/EBL system incorporated a empirical method of assessing the effectiveness of learned modifications with respect to a particular task distribution [Minton88b].

Although there is considerable progress in learning techniques, the customization problem is far from solved. The chief complication is ensuring that learned modifications actually improve problem solving ability as defined by the particular application. Unfortunately, even for the specific case of speed-up learning, it has proven difficult to insure this fundamental property. It is an empirical fact that many learning approaches can dramatically *degrade* problem solving behavior in some situations. In speed-up learning, this property has come to be known as the *utility problem* [Minton88b]. Seemingly innocuous changes to a problem solver's domain encoding can have drastic effects on the success of a learning approach see [Etzioni90b]]). Techniques intended as distribution-independent can fail under some distributions (see [Pérez92] and Chapter 3). The difficulty is that modifications typically involve some tradeoff. Changes that increase performance on one class of problems may degrade performance on others. This fundamental issue must be resolved before learning can provide an effective means of customizing problem solvers

1.4 A GUIDE TO THE THESIS

In this thesis we place the customization problem on a sound mathematical foundation and propose one technique for solving this problem. In Chapter 2 we illustrate how the customization problem can be formulated in terms of decision theory. This provides a precise and well understood characterization of the problem. It also suggests a natural approach to solving the customization problem in terms of statistical decision theoretic inference. We introduces our COMPOSER approach, an implementation of such a technique. COMPOSER is a general statistical tool for selecting proposed modifications of a learning technique, given an application specification.

Chapters 3 and 4 describe two applications of COMPOSER. In the first application we describe the customization of the PRODIGY planning system over several artificial problem solving applications. The evaluation compares COMPOSER to several other state of the art learning approaches. In the second application we describe the customization of a Jet Propulsion Laboratory scheduling system in the real-world application of spacecraft communication scheduling. The evaluation compares COMPOSER to an expert constructed domain-specific search strategy. Chapter 5 provides a sensitivity analysis of COMPOSER, examining the algorithm's performance as a function of its parameters. Chapter 6 describes limitations to the technique and future work. Chapter 7 discusses conclusions.

Chapter 2

COMPOSER: Decision-Theoretic Customization

The previous chapter provided an abstract definition of the customization problem. In this chapter we illustrate how this can, with some restriction, be formalized as a problem of decision theoretic inference. This characterization will provide the foundations for COMPOSER, a statistical customization technique. Many approaches have been proposed for modifying a problem solver. COMPOSER acts as a general tool for evaluating these proposed modifications. It is a generate and test statistical approach for identifying a good customization to an initial problem solver, given a space of potential modifications.

The chapter is developed in four parts. Section 2.1 develops the formalization of the customization problem in terms of decision theory. Decision theory provides a sound foundation, but it is neutral to many of the efficiency concerns involved in constructing a practical algorithm. Section 2.2 describes the COMPOSER algorithm, one efficient approach for decision-theoretic customization. Section 2.3 goes into greater detail about the COMPOSER algorithm and discusses issues involved in applying the technique to different applications. Finally, Section 2.4 discusses the computational properties of the approach.

2.1 DECISION THEORY

To evaluate the customization problem we must provide a formal basis for discussing the merits of learned modifications. In the general case the exact pattern of tasks associated with an application can involve a random element. Therefore, the problem of identifying a good modification is essentially a problem of decision making under uncertainty. A customization system should decide to adopt a modification, from a set of alternatives, if it "improves" overall problem solving performance, given this uncertain pattern of tasks. To evaluate the alternative modifications, we must carefully articulate what it means for a problem solver to be improved with respect to this pattern. The measure of success depends on evaluation criterion of the application, and we must describe how this criterion can be expressed in a way that is useful to the customization system.

Decision theory provides a widely excepted framework for decision making under uncertainty (see [Berger80]). Jon Doyle has persuasively argued for the merits of decision-theory as a standard for evaluating artificial intelligence systems [Doyle90] and it has seen increasing acceptance, both in artificial intelligence at large [Horvitz89, Russell89, Schwuttke92, Wellman92] and machine learning in particular [Gratch92a, Greiner92a, Laird92, Subramanian92]. We will use decision theory as a common framework for characterizing the behavior of modified problem solvers. Decision theory relies on the observation that different decisions can result in different outcomes, and that, under some natural assumptions, preferences over these outcomes be characterized by a real-valued *utility function.* An outcome A is preferred to outcome B iff the utility of A is greater than the utility of B. When uncertainty is involved, it may not be possible to ensure a decision results in a single outcome. Instead, a decision is characterized by a set of outcomes and the probability distribution across this set. However, it is possible to characterize preferences over this set of outcomes by combining their individual utilities into an *expected utility* of the set. This is the utility of all possible outcomes weighted by their probability of occurrence. Under uncertainty, a decision maker should choose the decision option that maximize expected utility.

Decision theory provides a formal framework for representing preferences over outcomes and provides tools for decision making that is consistent with these preferences. We will illustrate how this framework naturally applies to the customization problem. But first, one caveat is important. Decision theory provides a framework for expressing preferences but it does not provide much guidance as to what preferences to express. In decision theory, aspects of an outcome that are related to desirability are called *attributes*. It is left to the designer of a decision-theoretic system to identify relevant attributes and combine them into a utility function. Even when only one attribute is relevant, it is important to distinguish between the value of an attribute and the utility associated with that value. Similarly, expected utility is not necessarily the same as averaging over the attributes that make up utility. Expected utility averages over utilities, not the attributes. Thus, as Doyle illustrates [Doyle90], bicycles designed to fit the average sized rider do not well serve an evenly mixed population of tall adults and short children. Similarly, an expected utility based on problem solving time might be other than just average running time (e.g., utility might be a *non-linear* function of running time).

2.1.1 Decision-Theoretic Interpretation of the Customization Problem

The customization problem states that the evaluation criterion describes the performance of a problem solver over some *pattern* of tasks. For our representation of the problem we impose the restriction that this pattern can be characterized as the random selection of tasks according to a fixed probability distribution over the set of possible tasks defined by the domain of discourse. This restriction states that there is some probability of occurrence associated with each task, and that this probability is independent of what tasks we have already seen, and that this probability does not change with time. This is a common simplification that applies to a great many applications. However, it does impose some restrictions that we discuss these in Chapter 6. While this distribution is generally unknown in advance, it can be reconstructed by randomly drawing an arbitrary large sample of tasks according to the distribution.

With the fixed distribution restriction one can represent the evaluation criterion for the application in terms of the expected utility of the problem solver over a distribution of problems. In particular, we require explicitly characterizing the behavior of a problem solver on a particular problem, by a finite numeric utility value.[1] This value must measurable in the sense that it is possible to construct procedures that determine this value given an observation of problem solving behavior. The behavior of a problem solver over a distribution of problems is characterized by the expected utility of the problem solver with respect to that distribution. A learning algorithm acts to modify the behavior of a problem solver; essentially taking an initial problem solver PS_{old} and creating a new problem solver PS_{new}. Therefore, we can characterize the value of a learned modification as simply the difference in expected utility between the modified and unmodified problem solvers.

1. The condition that the value is finite has important consequences. For example a possible utility function is the number of search nodes explored while processing a problem. If problem solving is semi-decidable, then some problems conceivably have infinite utility, which would lead to an undefined expected utility. To make an acceptable utility function based on this attribute we must impose some resource bound on the problem solver.

The customization problem, therefore, can be reformulated in decision theoretic terms as the problem of identifying a learned modification that improves the expected utility of problem solving.

Formally, let *PS* be a problem solver, *D* be the (possibly infinite) set of possible problems in the domain of discourse, Pr_D be a function that returns the probability of occurrence for any problem in *D,* and $U(PS, x)$ be the utility function, characterizing the behavior of the problem solver over a given problem *x*. The expected utility of *PS* with respect to the distribution, written $E_D[U(PS)]$, is defined as:

$$E_D[U(PS)] = \begin{cases} \displaystyle\int_D U(PS, x)\,\Pr_D(x)dx \ \ (D \ \ continuous) \\ \displaystyle\sum_{x \in D} U(PS, x)\,\Pr_D(x) \ \ (D \ \ discrete) \end{cases} \tag{1}$$

This formulation highlights some additional issues. A learning algorithm may have some choice as to what modification to adopt. A modification that reduces expected utility should clearly be avoided. Given several modifications that improve expected utility, we would prefer the customization algorithm to adopt the modification with the largest improvement in expected utility. An algorithm that always choose the modification with the largest improvement in expected utility would be considered *optimal with respect to the space of modifications*. Unfortunately, this ignores the computational cost involved in finding and adopting modifications. In many cases it may not be computationally feasible to identify the optimal decision (e.g. [Greiner89]). A second issue is that optimal decision making requires that the customization algorithm have perfect knowledge of the task distribution and perfect knowledge of how modifications effect utility. This is rarely possible and instead customization algorithms must estimate this information with training data, drawn according to the distribution. If these estimates differ from the true information, the algorithm may make incorrect decisions. Our definition of customization success judges the value of a modification against a particular problem distribution, and it is assumed that this distribution remains unchanged before and after learning. Because of these issues we arrive at at formal definition of the customization problem.

DEFINITION 1: THE CUSTOMIZATION PROBLEM. *Given (1) an application, described by a utility function, domain of discourse, and access to examples tasks drawn from a fixed probability distribution over the domain of discourse, (2) a space of possible problem solvers, defined by an initial problem solver,* PS_{old}*, and a set of possible modifications, and (3) a specified confidence level, adopt a modification such that, with the specified confidence, the expected utility of the resulting problem solver,* PS_{new}*, is greater than or equal to expected utility of* PS_{old}.

This decision-theoretic interpretation provides a well defined metric by which to judge a customization technique. This is, in fact, the natural interpretation of learning curves presented in empirical evaluations of learning algorithms. In these empirical evaluations a learning algorithm is validated over some distribution of problems, the ability of the problem solver is rated by some attributes of performance (generally one attribute, like correctness or time) and the average performance, or expected utility, is demonstrated.

2.1.2 Composability Problem in Incremental Algorithms

Definition 1 lends itself to a natural solution: list every possible modification and empirically observe the difference in expected utility each affords across a sufficiently large sample of problems. Most learning techniques, however, treat learned knowledge in a more incremental fashion. Instead of treating the process of constructing a modified problem solver as a monolithic single step, a modified problem solver is built up incrementally by combining several individual atomic units. For example, SOAR constructs a new problem solver by incrementally incorporating individual chunks [Laird86]. PRODIGY/EBL builds a learned control strategy by combining individual control rules [Minton88b]. This compositional treatment of learning affords the learning algorithm greater flexibility and it allows for more focused learning behavior. Just as individual planning operators can be flexibly combined into a variety of goals, individual learned units can be combined to address the particular combination of problem solving inefficiencies. Unfortunately, this compositional treatment creates new difficulties. The learning algorithm must assess how the individual atomic units combine to influence expected utility.

The *composability problem* is the problem of identifying an effective modification, given that it must be constructed from multiple individual units. Solving this problem has been a major obstacle to previous approaches. Ideally, it would be possible to provide a local measure of the quality of individual atomic units. With such a measure, a customization system could easily construct a good composite modification by combining several units with good local evaluation. Mitchell *et. al.*'s operationality criteria was intended as such a measure [Mitchell86]. Unfortunately, for most (if not all) incremental learning techniques proposed, there does not appear to exist reliable local measures as the utility of a composite is determined by complicated interactions between its individual components. These interactions are such that the overall behavior of a composite modification may be difficult to predict, *even if we exactly know how these components perform in isolation.*

2.1.2.1 Example Composibility Problem

A complete discussion of how the composability problem effects learning algorithms is found in [Gratch91b]. We illustrate the difficulty with a simple example of building a composite modification to reduce search from individual search control heuristics. Let utility be a simple function of a single attribute of planning behavior: time to search on a given problem. heuristics reduce search, but they introduce the overhead of evaluating them. In isolation, such a heuristic increases utility if the search saved exceeds this evaluation cost. The PRODIGY/EBL learning algorithm is an example of such an approach [Minton88a]. There are many ways such heuristics can interact so that the improvement of multiple heuristics is not the sum of the improvements of the heuristics in isolation. Two heuristics may avoid the same search nodes and as there is no added benefit in ignoring an area twice, the utility of the two together is not equivalent to the sum of their improvements in isolation. A subtle example occurs when a heuristic has different evaluation costs in different portions of search. A second heuristic that removes portions of this search space may substantially change the average match cost of the first heuristic.

A particular interaction between two heuristics is illustrated in Figure 1. This shows a hypothetical search space of fifteen nodes. Supposed r and s are two heuristics prune the nodes in sets R and S respectively when compared to search without heuristics. $|R|$ is the number of nodes trimmed by r. $|S|$ is similarly defined. When used in isolation, r is checked six times (i.e. $15 - |R|$). It successfully applies twice: at node 2 saving nodes

3–8 and at node 9 saving nodes 10–12. Heuristic *s* is checked eight times (i.e. 15 – |S|) and succeeds at node 1, saving nodes 9–15. Assume the average evaluation cost of *r* is M_r, the average cost of *s* is M_s, and the average time to expand a node is *g*.

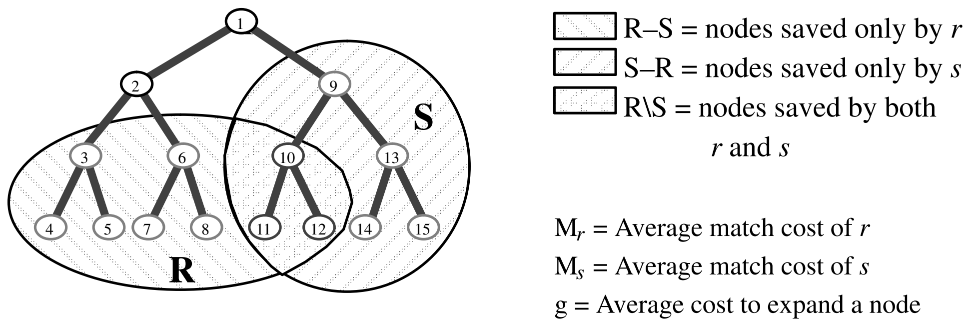


Figure 1: Example of interacting heuristics

Let $U(X, p)$ be the utility of a problem solver using the set of rules X on problem p. The interaction between two rules on a problem is the amount to which their utilities are not additive:

$$\text{Residue} = U(\{r, s\}, p) - [U(\{r\}, p) + U(\{s\}, p)] \;=\; |R-S| \times M_s + |S-R| \times M_r - |R \cap S| \times g$$

This residue is a measure of the interaction between heuristics. Heuristics combine synergistically if this value is positive. If negative, they engage in a harmful interaction. Two heuristics with positive utility can potentially combine to yield a strategy worse than neither. Such interactions would seem to preclude effective local criteria for determining the benefit of a composite modification.

2.1.3 Decision Theoretic Interpretation of the Composability Problem

When a modification is made up of many individual units we can, under certain circumstances, consider the expected utility of a modification in terms of the these units. A composite modification, when applied to a problem solver, results in a modified problem solver. It is often the case with learning algorithms that individual units can also be treated as composite modifications. Thus applying an atomic unit to an initial problem solver also results in well-defined problem solver. When this condition holds we can view a composite modification as a sequence of problem solvers: $PS_{old}, PS_1, PS_2, ..., PS_{new}$. For example, SOAR modifies a problem solver by apply a sequence of chunks. Each intermediate step is a valid problem solver (the original problem solver using the current set of chunks).

With the decision-theoretic interpretation we can characterize the effect of a atomic units in terms of its their effect on expected utility. Let *PS* be a problem solver and *Apply*(τ, *PS*) be the problem solver that results from applying an atomic element τ to *PS*. We refer to τ as a *transformation* to *PS*. We define the *incremental utility* of τ be the difference between the expected utility of *Apply*(τ, *PS*) and the expected utility of *PS*.[2] We denote incremental utility as $\Delta U_D(\tau|PS)$, meaning the conditional change in expected utility provided by transformation τ over distribution D given problem solver PS. Letting *Apply*(*t, PS*) = PS^{τ} we can state this formally as:

$$\Delta U_D(\tau|PS) = E_D[U(PS^{\tau})] - E_D[U(PS)]$$

or equivalently:

$$\Delta U_D(\tau|PS) = \begin{cases} \int_D \left[U(PS^{\tau}, x) - U(PS, x)\right] \Pr_D(x)dx \quad (D \;\; continuous) \\ \sum_{x \in D} \left[U(PS^{\tau}, x) - U(PS, x)\right] \Pr_D(x) \quad (D \;\; discrete) \end{cases} \tag{2}$$

The total change in expected utility over an entire composite modification is simply the sum of the incremental utilities of the atomic units: $\Delta U_D(\tau_0|PS_{old}) + \Delta U_D(\tau_1|Apply(\tau_0, PS_{old})) + \ldots$

The definition of incremental utility clarifies two important properties of transformations. First the effect on utility that results from a transformation is dependent on the distribution D. Second, the effect is conditional on the problem solver to which it is applied. In general the incremental utility of a transformation will vary unpredictably as we change either the distribution or the problem solver to which it is applied. The conditional nature of incremental utility ensures that in general we cannot identify a globally maximal composite modification simply with local utility evaluations, independent of problem solver to which a transformation is applied. To identify a globally maximum composite modification, we must consider an potentially explosive number of conditional utility values.

2.2 COMPOSER ALGORITHM

Decision Theory provides a sound mathematical foundation on which to construct a customization routine. The effectiveness of a composite modification can be judged in terms of its expected utility. Individual components of a composite modification can be assessed by their incremental utility. The foundation does not, however, dictate how to identify an adequate composite modification in the perhaps vast space of possibilities. It also does not state how expected or incremental utility values can be determined for a given application. Therefore, to construct a practical customization procedure we must build some addition commitments atop this foundation. This section describes these commitments as they are embodied in the COMPOSER customization algorithm.

To perform customization, COMPOSER requires not simply a problem solver, but one with a flexible control architecture. In general, searching problem solvers can be viewed as non-deterministic search algorithms. For example Chapman's Modal Truth Criteria specifies the control decisions, but not a particular control strategies [Chapman87]. A flexible problem solver can then be characterized by a set of c*ontrol points* $CP_1 \ldots CP_n$, where each control point CP_i is associated with a particular control decision. For example, a CSP problem

2. In other learning algorithms incremental utility is simply referred to as utility. For example, Minton discusses the "utility of control rules" in regards to PRODIGY/EBL. We feel this additional terminology is necessary to highlight the difference between the utility of a problem solver, which is of interested to the user, and the utility of the a transformation which is only of interest to the learning algorithm.

solver might be characterized by two control points, one associated with choosing a variable to instantiate next, and one to choose a value to assign the variable. A problem solver is flexible if there is a set of alternative methods $M_{i,1}...M_{i,k}$,[3] for each of these control points. A *control strategy* is a selection of a method for each control point associated with the problem solver (e.g, $STRAT = \langle M_{1,3}, M_{2,6}, M_{3,1}, ...\rangle$). A default control strategy can be represented by a default method for each of these control points. Machine learning methods act by proposing atomic transformations to the methods comprising a problem solver's control strategy.

The decision-theoretic foundation provides a well defined metric for evaluating alternative control strategies. COMPOSER should search through this space of alternative control strategies for one with high expected utility, with respect to the application's distribution of tasks. We first describe COMPOSER's search strategy, local hill climbing. We then describe the statistical aspects of the approach. Finally we discuss factors involved in applying COMPOSER to a particular application.

2.2.1 Statistical Hill-climbing

Because of the composability problem, it is quite difficult to determine the best sequence of transformations. With interactions it is possible that two or more transformations will have positive incremental utility in isolation, but have negative incremental utility when combined together. In fact, if we want to find the best problem solver in the space, unless we are able to make strong statements about the way transformations interact, we must evaluate every problem solver expressible with the grammar of modifications, and this can often prove infeasible. However, Definition 1 only requires that we *improve* the expected utility of the initial problem solver. COMPOSER satisfies this definition through a greedy hill-climbing procedure for exploring the space of transformations. COMPOSER begins with the original problem solver and incrementally adopts transformations that are estimated to possess positive incremental utility. Each new transformation is assessed with respect to the problem solver that results from applying the previous transformation. COMPOSER uses statistical methods to estimate incremental utility from training examples drawn randomly according to the distribution of problems. This hill-climbing approach successfully avoids the difficulty of negative interactions. However it cannot recognize and exploit positive interactions. Therefore the search will be restricted to local optima.

2.2.1.1 Transformation Generator

COMPOSER implements a generate and test architecture for learning composite modifications. The application designer must provide a *transformation generator* that specifies the possible transformations at each step in the search. At each step in the search the generator takes the current problem solver and possibly some training examples, and determines a set of possible transformations. COMPOSER's statistical hypothesis tester then adopts one of these proposed transformations by evaluating the set on further training examples. The transformation generator can be formalized as a function $TG : PS \times \vec{X} \rightarrow \{\tau_1, \ldots, \tau_k\}$ that maps a problem solver and an optional vector of training examples into a set of candidate transformations. Each transformation in the set should map the problem solver into some new, possibly improved, problem solver. For example, when SOAR reaches an impasse, the transformation generator takes the current problem solver and the single training problem that produced the impasse, and generates a set of chunks. STATIC's transformation generator takes the original problem solver and no training examples, and generates a set of control rules. COMPOS-

3. Note that a method may consist of smaller elements so that a method may be a set of control rules or a combination of heuristics.

ER takes the output of such transformation generators and feeds it into a statistical testing procedure. This procedure examines the set of possible transformations and chooses one with positive incremental utility. By proceeding through a series of generate and test steps, COMPOSER increasingly improves the expected utility of problem solving.

2.2.1.2 Modeling Statistical Error

Figure 2 illustrates how COMPOSER explores the transformation space. At each step the algorithm evaluates a set of alternative transformations to the current problem solver, and chooses one that has positive estimated incremental utility. COMPOSER proceeds through a number of such steps until the training data is exhausted. When this occurs, the current problem solver is returned as PS_{new}.

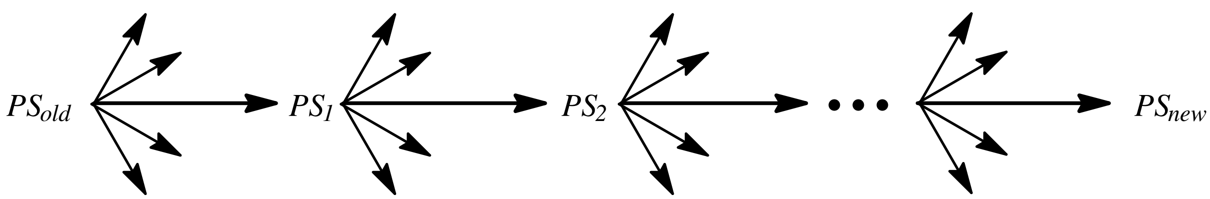


Figure 2: A typical search through the transformation space.

The search proceeds by developing estimates of the incremental utility for each transformation at each step. Each of these estimates may involve some statistical error. To satisfy Definition 1, COMPOSER must ensure that the *overall* error remains below some threshold δ. Formally:

$$Pr(E_D[U(PS_{new})] < E_D[U(PS_{old)}]) \le \delta \tag{3}$$

To achieve this, COMPOSER must account for several sources of error. Each time COMPOSER identifies a new transformation to adopt, it has done so by comparing a set of estimates, one for each transformation considered at that step. There will be some probability of error for each of these estimates, and these individual errors combine into a somewhat larger probability of error for the overall decision of which transformation to adopt at that step. PS_{new} is produced as a result of several individual steps, and there is some probability that a given step has negative incremental utility. Thus, COMPOSER must account for the error in the estimates of transformations, how these errors combine at each step, and how errors combine across steps.

The strategy we adopt in COMPOSER is to satisfy Definition 1 by bounding the error of each incremental utility estimate in such a way the the overall error is expected to remain below δ. The bound for each estimate is determined by the function *Bound*(δ, *i*, |T|), which returns the error level as a function of the overall error, the current step in the hill-climbing search, and the size of the set of transformations at that step. As we will discuss in Section 2.3, there is reason to provide some flexibility in the definition of this function. A default definition and several standard options are discussed in that section.

It now simply remains to construct a statistical procedure to estimate incremental utility to the specified error bound. We rely on the statistical observation that the expected value of a function can be estimated by averaging multiple observations of the function's value, where these observations are drawn randomly from the probability distribution. Referring back to Equation 2, notice that the incremental utility of a transformation τ is the expected value of $U(PS^{\tau}, x) - U(PS, x)$. This is, for an arbitrary problem x, the difference in utility between solving it with the problem solver PS and solving it with the transformed problem solver $PS^{\tau} = Apply(\tau, PS)$. We can construct an estimate of incremental utility by randomly drawing problems according to the distribution D and, for each transformation under consideration, averaging several observations of this value. Call $\overline{\Delta U}_n(\tau|PS)$ the estimated incremental utility from n data points.

To gather the necessary data points, COMPOSER must determine, given a current problem solver PS, a set of transformations T, and a problem x, the difference in utility between the current and each of the transformed problem solvers: $\forall \tau \in T$, $U(PS^{\tau}, x) - U(PS, x)$. Recall that, by definition, the utility of a problem solver on a problem is measurable by observing the behavior of the problem solver on the problem. Thus, given a set of m transformations, we can compute the necessary incremental utility values by solving the problem with PS and then with $PS^{\tau 1}$ ... and then with $PS^{\tau m}$. Processing each training example involves m+1 problem solving attempts. The complexity of processing an example is therefore tied to the complexity of each of the m+1 problem solvers. We call this *brute-force* processing and is the default method used by COMPOSER. We show in Section 2.3 that for many applications there are other more efficient methods for obtaining these data points.

Estimates become more accurate as we increase the number of data points. The problem, then, is to determine a suitable sample size for each estimate. Specifically, COMPOSER should determine a sufficiently large sample size such that it can bound the probability that incremental utility is estimated to be positive (negative) when in fact it is negative (positive). The estimates should be accurate but for efficiency reasons we would like them to be based on as few examples as possible. COMPOSER relies on a *sequential* statistical technique to determine how many examples are sufficient to make this inference [Govindarajulu81]. Sequential procedures differ from the more common fixed–sample techniques in that the number of examples is not determined in advance, but is a function of the observations. Sequential procedures provide a test called a *stopping rule* that determines when sufficient examples have been taken. An important advantage of sequential procedures is that the average number of examples required is smaller than the number required by a fixed–sample technique.

The intuition behind the stopping rule is quite simple. Given a sample of values and a confidence level α, we can construct confidence intervals that contain, with probability $1 - \alpha$, the true incremental utility lies.[4] In other words, there is only probability α that the true incremental utility lies outside the confidence interval.[5] With more examples, the width of interval shrinks. The stopping rule determines how many examples are needed for the confidence interval to shrink to the point of being entirely above or below zero incremental utility. This is illustrated in Figure 3. When this occurs, we can state, with probability $1 - \alpha$, that the transformation has positive (negative) incremental utility if the estimate is positive (negative).

4. This holds under certain statistical assumptions with are described below. This is a two-sided statistical test and thus overly conservative. We are not aware of an acceptable one-sided test for this problem.

5. More precisely, for any given sample, the mean is either in or out of the interval. Averaging over multiple random samples the mean will be in the computed confidence interval with probability $1 - \alpha$.

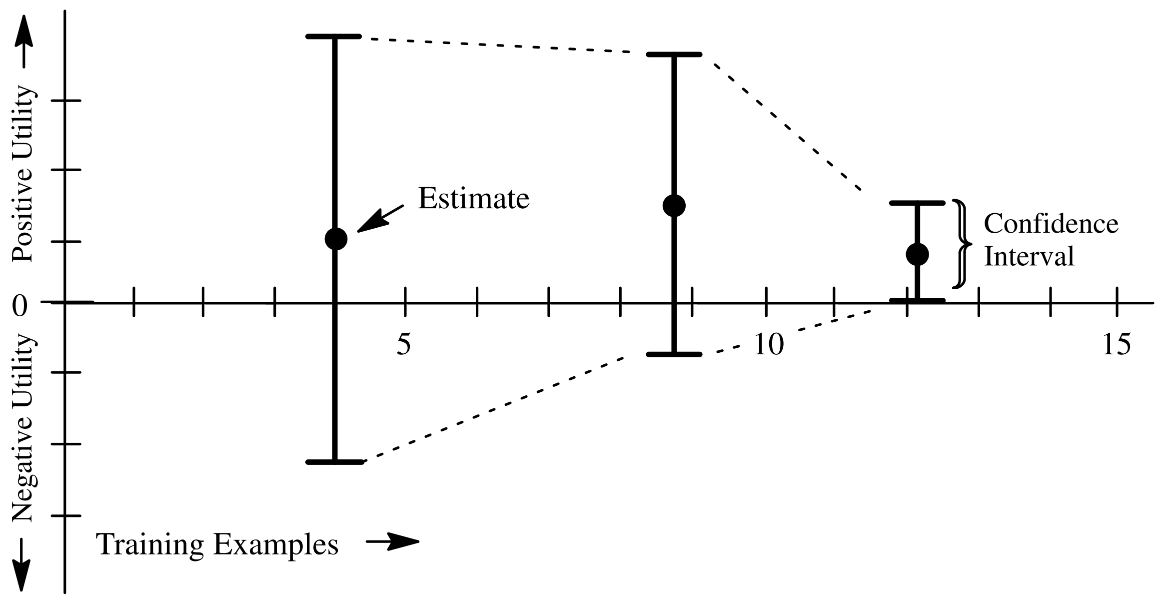


Figure 3: Sizing the confidence interval. We would like to take enough examples such that the confidence interval lies entirely above or below the axis.

2.2.1.3 Determining a Sufficient Sample Size

COMPOSER determines the sample size for each incremental utility estimate using a stopping rule proposed by Arthur Nádas [Nádas69]. Examples are taken until the following equation holds:

$$n \geq n_0 \ AND \ \frac{S_n^2(\tau|PS)}{\left[\overline{\Delta U_n}(\tau|PS)\right]^2} \leq \frac{n}{Q(n,\alpha)^2} \tag{4}$$

where n is the number of examples taken so far, $\overline{\Delta U_n}(\tau|PS)$ is the transformation's average improvement, $S_n^2(t|PS)$ is the observed variance in the transformation's improvement, α is the acceptable error in the estimate, n_0 is a small finite integer indicating an initial sample size, and $Q(n, \alpha)$ is a function that models the discrepancy between the true and estimated incremental utility. As in the *Bound* function, there is reason to provide some flexibility in specifying $Q(n, \alpha)$. A default function and some standard options are discussed in the Section 2.3. The initial sample size, n, is related to the definition of $Q(n, \alpha)$ and is also discussed in Section 2.3. If Equation 4 holds, the transformation will speed-up (slow down) *PS* if $\overline{\Delta U_n}(\tau|PS)$ is positive (negative) with confidence $1-\alpha$.

The overall COMPOSER algorithm is illustrated in Figure 4. After each problem solving attempt, COMPOSER updates the statistics and evaluates the stopping rule for each candidate transformation. If no transformation has attained the confidence requirement, another problem is solved. If the stopping rule identifies transformations with positive incremental utility (there may be more than one), COMPOSER adopts the one with highest positive incremental utility, and removes it from the candidate set. Statistics for the remaining candidates are discarded as they reflect the previous problem solver. If the stopping rule identifies candidates with negative incremental utility, they are eliminated from further consideration. Eliminating a candidate does not affect the current problem solver, so the statistics associated with the remaining candidates are unaffected. This cycle is repeated until the training set is exhausted. Each time a transformation is adopted the expected utility of the resulting problem solver is increased, giving COMPOSER an *anytime* behavior [Dean88].

Given: PS_{old}, $TG(\cdot)$, δ

[1] $PS := PS_{old}$; $T := TG(PS)$; $i := 0$; $n := 0$; $\alpha := Bound(\delta, i, |T|)$;

[2] While more examples and $T \neq \emptyset$ do

[3] $n := n + 1$

[4] $\forall \tau \in T$: Get $\Delta U_n(\tau|PS)$ /* Gather statistics and find transformations that have reached significance */

[5] $$\alpha\text{–}estimates := \left\{ \tau \in T : n \geq n_0 \text{ and } \frac{S_n^2(\tau|PS)}{\left[\overline{\Delta U_n}(\tau|PS)\right]^2} < \frac{n}{\left[Q(n,\alpha)\right]^2} \right\}$$ /* Collect all trans. that */ /* have reached statistical */ /* significance. */

[6] $$T := T - \left\{ \tau \in \alpha\text{–}estimates : \overline{\Delta U_n}(\tau|PS) < 0 \right\}$$ /* Discard trans. that decrease expeced utility */

[7] If $\exists \tau \in \alpha\text{–}estimates : \overline{\Delta U_n}(\tau|PS) > 0$ Then /* Adopt τ that most increases expected utility */

[8] $$PS = Apply(x \in \alpha\text{–}estimates : \forall y \in \alpha\text{–}estimates \left[\overline{\Delta U_n}(x|PS) > \overline{\Delta U_n}(y|PS)\right], PS)$$

[9] $T := TG(PS)$; $i := i + 1$; $n := 0$; $\alpha := Bound(\delta, i, |T|)$

Return: *PS*

Defaults: $$Bound(\delta, i, |T|) := \frac{\delta}{|T|} \quad , \quad Q(n,\alpha) := x \;\; where \int_x^\infty \left(1/\sqrt{2\pi}\right) e^{-0.5y^2} dy = \frac{\alpha}{2}, \qquad n_0 := 15$$

Figure 4: The COMPOSER algorithm

2.3 IMPLEMENTATION TRADEOFFS

The principal impediment to COMPOSER's approach to customization is managing the computational expense of identifying good modifications. To maximize COMPOSER's performance we would like it to efficiently process each example and to use as few examples as possible to perform its statistical inferences. More efficient use of examples allows more hill-climbing steps with the same number of examples, and presumably a greater increase in expected utility. When applying COMPOSER to a particular application there are several ways in which this expense can be mitigated. This can be addressed in three ways:

1. Configuration of the transformation generator
2. Configuration of the error modeling procedures
3. Configuration of methods for gathering incremental utility statistics

As COMPOSER is intended as a general tool, it provides default policies for each of these factors. This section describes the rational behind these default settings and describe alternative approaches. The configurable aspects allow the application designer to take advantage of domain specific knowledge to improve learning efficiency.

2.3.1 Transformation Generator

Any interesting customization technique will possess a large vocabulary of individual atomic units, which, in turn, will result in a vast space of possible composite modifications. COMPOSER imposes hill-climbing to manage this complexity. The transformation generator should propose a relatively small number of transformations to any existing control strategy. This said, there may be many ways to implement a transformation generator, given a vocabulary of modifications. Furthermore, there may be additional information available to the application designer that could further reduce the complexity of the search.

There are two basic issues involved in constructing an effective transformation generator. First there is the issue of the quality of learned modifications. Hill-climbing restricts any identified solutions to local maxima. A transformation generator imposes some topology over the space of possible modifications. Ideally the topology is such that hill-climbing leads to global solutions. The second issue is one of customization efficiency. The expense of customization is tied to the number of alternative transformations considered at each step. To increase efficiency we would like to reduce this branching factor as much as possible. Obviously these two goals are incompatible. If we are only concerned with expected utility, we could force COMPOSER to consider all possible modifications in a single step.

As a default method, we recommend a transformation generator that performs all considers all single-step modifications to the current control strategy. Thus if we have three control points, and two possible modifications to each control point, the transformation generator would propose size alternate strategies. Of course, it is best if this set can be restricted by an analysis of the current problem solver's behavior. For example, many learning techniques, like PRODIGY/EBL's control rule generator, possess a vast set of possible atomic units, but only propose a small number in response to observed inefficiencies.

Alternative methods are possible if some information is available about how different control points interact. For example, if it is known that the effects of one control point are completely independent of another, one can structure the space of modifications to reduce the search. This structure can be viewed as the consequence of asserting certain types of relations between control points. We have identified three types of relations that may be useful in devising a transformation generator. Independence relations indicate cases in which a control point can be viewed as primarily independent of another control point. A set of control points A can be divided into sets B and C if every control point in A is independent of every control point in B. Dominance relations indicate that the changes in utility from changing methods for one control point are much larger than the changes in utility for another control point. Again, a set of control points A can be divided into sets B and C when every control point in B dominates every control point in C. Finally, inconsistency relations indicate when a method M_1 for control point X is inconsistent with method M_2 for control point Y. This means that any strategy using these methods for these control points need not be considered.

2.3.2 Modeling Statistical Error

COMPOSER provides two functions to model statistical error. The basic tradeoff in setting these functions involves properly limiting the chance of statistical error while at the same time allowing inferences with as few training examples as possible. Generally it is not possible to exactly quantify statistical error. The two basic approaches to addressing this problem are to use worst-case bounds which are conservative on statistical error but require many examples, and approximation strategies which require many fewer examples but may in some circumstances produce higher than expected statistical error.

2.3.2.1 Choosing **Bound**(δ, **i**, $|T|$)

The Bound function defines the error level for each incremental utility estimate in such a way that the overall probability of learning a worse problem solver is less than δ. To do this, one must account for two sources of error, the error at each step in the hill-climbing search given that we are choosing a step from a set T of estimates, and the cumulative error over each step in the hill-climbing search. We will look at these sources individually. We use α to denote the error of each estimate, β_i to denote the error in step *i*.

2.3.2.1.1 Error in a Step

On step *i* we are investigating a set T of possible transformations. Given that the error of each of the estimates is α, the expected error for the step, β_i, will tend to be greater than α, This is most clearly seen in the worst case where every member of T has negative incremental utility. The correct decision for this step is to not adopt any transformation. However, there is probability α that a given transformation will be estimated to have positive incremental utility and be adopted. Typically, the larger the number of transformations, the greater the probability that at least one of the estimates will be in error. Thus, β_i is a function of the size of T. The actual relationship depends on the covariance between the distribution of incremental utility data points associated with each transformation. Unfortunately, this information is generally unavailable.

The situation is more complicated, but yields less expected difference between α and β_i, when there are transformations with both positive and negative incremental utility in T. COMPOSER adopts the first transformation that has positive estimated incremental utility. One way of viewing this is that each transformation is racing against the others. In this situation there are two conditions for making an error at the step. First, a transformation with negative incremental utility must be estimated to have positive incremental utility. Second, the mistaken estimate must win the race. It must have a smaller stopping time than each transformation transformations with positive incremental utility.

It is difficult, if not impossible, to characterize the precise relationship between the size of T and β_i. We have considered three methods for setting α to achieve a step error of β_i:

$$WORST-CASE : \alpha := \beta_i/|T| \quad \text{(default)} \tag{5}$$

$$INDEPENDENCE : \alpha := 1 - \sqrt[|T|]{1-\beta_i} \tag{6}$$

$$BEST-CASE : \alpha := \beta_i \tag{7}$$

In the worst case (5) the error can grow linearly with the size of the number of transformations. This captures the worst possible relationship between transformations: every transformation has negative incremental utility and the covariances between the distributions are the worst possible. This situation will probably never arise in practice and can result in large sample sizes if the size of T is large, however, it does provide strong guarantees that the observed statistical error will not be higher than expected. The worst-case is our default model. The best-case model, Equation 7, assumes that the error does not grow appreciably as the size of T grows. We have performed some empirical evaluations that show that this assumption can be reasonable if T is relatively small. The advantage of this assumption is that the size of T does not have to be known in advance so we can allow the transformation to add new transformations into T as we are evaluating the existing members. An intermediate model assumes that the error of one estimate is independent of the others (Equation 6). This is more reasonable than the worst-case bound, but for small β_i there is little difference between this and the worst-case bound. Thus we use the simpler Equation 5 as our default.

2.3.2.1.2 Error Across Steps

Each time COMPOSER takes a step there is some chance of adopting a transformation with negative incremental utility. Let β_i denote this error for the *i*th step. If there is a one in ten chance that COMPOSER will make an error on each step, then after ten steps there is a good chance that an error was made on at least one step. However, even with one bad step the final problem solver may still be a significant improvement. By default, we implement a liberal policy. COMPOSER allows an error of δ at each step, the rational being that it is worth one step backwards to quickly take nine steps forward. A second, worst-case, approach is to limit the probability that COMPOSER will adopt *any* incorrect step to less than or equal to δ (i.e., $\Sigma\beta_i \le \delta$). This more than guarantees that COMPOSER satisfies the error requirement, but will require many more examples than the former approach. There are different ways to implement the worst-case approach depending on if we know the number of possible steps in advance. We have considered three methods for setting β_i in terms of the overall error parameter δ:

$$\mathit{LIBERAL\text{–}BOUND} : \beta_i := \delta \quad \text{(default)} \tag{8}$$

$$\mathit{WORST\text{–}CASE\text{–}BOUND/LIMITED\text{–}STEPS} : \beta_i := \delta/k \tag{9}$$

$$\mathit{WORST\text{–}CASE\text{–}BOUND/UNLIMITED\text{–}STEPS} : \beta_i := \frac{6\delta}{i^2\pi^2} \tag{10}$$

The default policy, Equation 8, relies on the assumption that the magnitude of the incremental utility of incorrect steps is comparable to the magnitude of the incremental utility of correct steps so that even if some steps reduce utility, the final result will tend to improve on the initial problem solver. This assumption has held across several simulation experiments. In situations where efficiency must be sacrificed for rigor, one of the later two equations should be used. In the case were the number of steps can be limited in advance, simply divide the error evenly over each of the *k* steps (Equation 9). When the number of steps is unbounded in advance, the error at each step must be such that no matter what the final number of steps, the total error sums to less than δ. Equation 10 satisfies this requirement.[6]

Typically we do not know the number of steps in advance. The limitation of Equation 10 is that the allowed error for each successive step decreases so the number of examples required fore each subsequent step sharply

6. This equation was suggested to us by Russell Greiner. and is the basis for his PALO algorithm (see [Greiner92a] and Section 3.4.4).

increases. This unfairly punishes later steps. Furthermore, there will always be a finite number of steps therefore the potential error that was reserved for later steps is wasted. For this reason we prefer Equation (8) as our default method. The problems of wasted error and punished later steps does not occur when the number of possible steps is known in advance. However, it is interesting that then number of examples required by (9) and (10) only differs by a small factor when many steps are involved (see Appendix C).

Once the application implementor chooses a model for the error within a step and a model for the error across steps, these should be combined into a overall function $Bound(\delta, i, |T|)$ which combines the two choices into a error level for each estimate for the ith step in the hill-climbing search.

2.3.2.2 Choosing Q(n, α) *and* $\mathbf{n_0}$

The function $Q(n, \alpha)$ characterizes the normalized expected discrepancy between the estimated incremental utility and the true incremental utility. The estimate is the average of a finite sample of n data points. This sample will be roughly representative of the actual distribution of data points as it is drawn randomly from the fixed distribution. However this sample will be more or less representative depending on random chance, and so the mean of the sample will be more or less close to the true mean of the distribution. The Nádas stopping rule requires a bound on the *normalized* difference, d, between the sample mean and the true mean. The normalized difference is the difference divided by the observed variance in the sample mean. The expected normalized difference can be modeled by a probability distribution function which shows the likelihood that a particular normalized difference arises from a random sample. Such a distribution function is illustrated in Figure 5. The bell shaped curve shows the likelihood of observing different normalized differences. The function $Q(n, \alpha)$, also called the $\alpha/2$th *quantile* of this distribution, is the positive difference d such that the probability of observing a difference greater than this is less than or equal to $\alpha/2$.

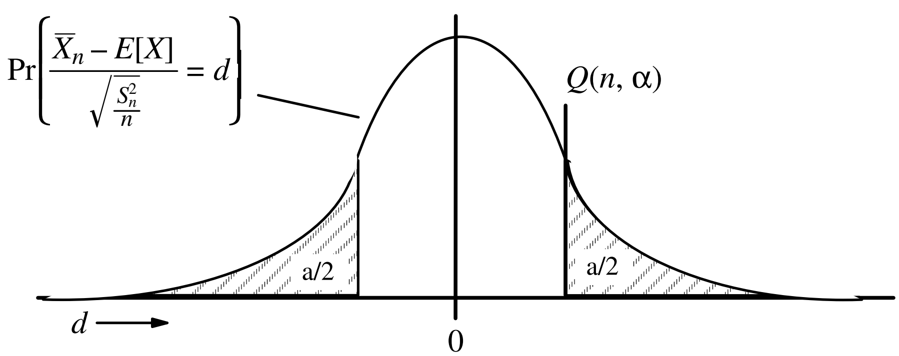


Figure 5: Probability distribution of the normalized difference between the sample mean and true mean of the original distribution. $Q(n, \alpha)$ is the value such that the probability of achieving a distance greater than this is less than or equal to $\alpha/2$.

It is rarely possible to know the exact distribution of differences and thus Q for a given learning situation. Fortunately, such distributions have the remarkable property that as n increases, the shape of the distribution converges to a normal distribution with zero mean and unit variance (also called a *standard normal distribution*). This property is asserted by the *central limit theorem* in statistics (see Appendix C.2). This fact implies, under some weak conditions,[7] that function $Q(n, \alpha)$ can be approximated satisfactorily using the quantile of a standard normal distribution. The approximation improves as the sample size increases. This is the motivation behind the n_0 parameter. Taking a sufficiently large initial sample of data points ensures an accurate approximation. We have investigated two approximation methods for $Q(n, \alpha)$:

$$STANDARD - NORMAL : Q_{\Phi}(n,\alpha) = x \text{ such that } \int_x^{\infty} \left(1/\sqrt{2\pi}\right) e^{-0.5y^2} dy = \frac{\alpha}{2} \quad \text{(default)} \tag{11}$$

$$T : Q_t(n,\alpha) = x \text{ such that } \int_x^{\infty} \frac{\Gamma[(n+1)/2]}{\sqrt{\pi n}\,\Gamma(n/2)(1+y^2/r)^{(n+1)/2}} dy = \frac{\alpha}{2} \tag{12}$$

The first, COMPOSER's default, is based on the standard normal distribution model. The second is based on a model called the *student t* distribution. This second model should prove more accurate when there is high variance in the sample, but it is more expensive to compute. For a given learning situation, the function $Q(n, \alpha)$ and n_0 should be chosen to best model the expected discrepancy in the given learning situation. If an exact model can be determined then an initial sample size is unnecessary. In general, the higher the variance in incremental utility values, the greater n_0 should be to ensure one of the approximation models is a close approximation. Smaller values for δ require more precise modeling of the error and therefore a better approximation. In general, the smaller the requested error level, the greater the n_0 should be to ensure a close approximation to δ. If n_0 is set too small, the likely result is a higher-than-requested statistical error. If n_0 is too large, an excessive number of examples is required to perform statistical inference. The optimal setting for n_o is difficult to determine, however the general experience in the statistical community is that the normal approximation becomes quite good after only a few initial samples. We recommend a value around fifteen. Chapter 6 provides a sensitivity analysis of this parameter.

2.3.3 Gathering Statistics

Given a current problem solver *PS*, a set of transformations Τ, and a problem *x*, COMPOSER must determine the difference in utility between the current and each of the transformed problem solvers: $\forall \tau \in T$, $U(PS^{\tau}, x) - U(PS, x)$. There are various strategies for obtaining this information. As mentioned in the previous section, the default strategy is to solve the problem with each of the candidate problem solvers. The complexity of this brute-force processing an example is therefore tied to the complexity of each of the $|T|$ problem solvers. In some learning situations brute-force processing may prove too expensive. This can happen, for example, if one or more of the candidate problem solvers is intractable. Furthermore, the brute-force force method is *intrusive* – it requires explicit experimentation with alternative problem solvers. In some learning situations is is desirable to learn passively, through the normal operations of the problem solver. As gathering statistics is COMPOSER's principal expense, it is important to take into account any information that could reduce this cost.

The ideal situation is if we possess a detailed cost model of the problem solver that efficiently derives the ramification of proposed modifications without actually solving the problem (e.g. [Greiner89, Subramanian90]).

7. The distribution must have positive variance and hence finite mean.

With such a model we could simple draw a random training example and then use the model to determine the effect of different transformations. Such models are rarely available. Short of this, it may be possible to extract the necessary statistics to determine the effectiveness of different transformations by observing the normal operations of the current problem solver. In Chapter 3 we describe one such *unobtrusive* implementation. Sometimes it is only possible to extract partial information unobtrusively. Greiner and Jurisica [Greiner92a] propose one method for using such partial information that could be incorporated into COMPOSER, although we have not currently done so.

2.4 COMPUTATIONAL PROPERTIES

We now discuss the computational characteristics of the COMPOSER algorithm. The proofs for all theorems mentioned in this section appear in Appendix B. Many of the algorithm's characteristics will, of course, be domain specific. For example, the number of transformations considered at each step, the number of possible steps, and the improvement in expected utility are all governed by the transformation generator and characteristics of a given application. However there are some general results we can state that will help assess characteristics of the algorithm. The theorems give *average-case* complexity results for the algorithm. All of the discussion assumes the default configuration of the algorithm: COMPOSER operating under Equations 5, 8, and 11, and using the brute-force method for gathering incremental utility statistics.

2.4.1 Sample Complexity

The sample complexity is the number of examples required to perform statistical inference. COMPOSER takes some number of examples at each step of the hill-climbing search. As stated, there is no way to bound the number of steps COMPOSER will take as this is a function of the particular transformation space associated with an application. However we can characterize the expected number of examples taken at a each step in the hill-climbing search in terms of several parameters. This is an useful characterization as improvements in expected utility are only realized after a step terminates.

Recall that α is the allowable statistical error associated with an incremental utility estimate (a bound on the probability that a transformation with negative incremental utility is perceived positive or *vice versa*), β is the allowable error at a step, and T is the set of transformations at a step.

Theorem 1. The expected number of examples required to produce an estimate for a transformation grows linearly with the variance of incremental utility values across different problems. The expected number of examples grows quadratically in one over the absolute value of the transformation's incremental utility. This means first, as the behavior of the transformation is more erratic (sometimes it helps, sometimes it hurts), it takes more examples to produce an estimate. Second, a transformation with very large or very small incremental utility will require fewer examples than one with a more modest performance change.

Theorem 2. The expected number of examples required at a step grows at most quadratically in the number of transformations at that step and grows at most quadratically in $1/\delta$. This means, all other things being equal, there will be an increase in the expected number of examples required at a step as we increase the number of candidates. This is due to the use of Equation 5 which requires greater confidence for each transformation estimate as we increase the size of T. Similarly, the sample complexity will increase polynomially as we require greater statistical confidence.

Remark: The quadratic bounds we show in Theorem 2 are the tightest we have been able to prove. However, the appendix provides an asymptotic argument that the growth is on the order of the natural log of these parameters. This log growth is what is observed empirically for large $1/\delta$.

2.4.2 Run-Time Complexity

The expected run time of the algorithm will depend on the number of examples used by the algorithm and the cost to process each example. Again, these factors will by influenced by the particular application. The number of hill-climbing steps taken by the COMPOSER algorithm will depend on the particular application, and may not be possible to bound in advance. Therefore, we provide results for the complexity of performing a single step. Under the brute-force method for gathering incremental utility statistics, the algorithm actually tries out each transformed problem solver in T over each example. Thus the cost of processing is tied to the complexity of the problem solver.

Theorem 3. Let R be a bound on the resources used by a problem solver on a single run. The expected cost of a step in the hill-climbing search grows linearly in R, at most quadratically in the required confidence, $1/\delta$, and at most cubically in the the number of transformations at that step, T.

Remark: As in Theorem 2, this is the strongest upper bound which we have been able to prove. Using the same asymptotic arguments as in Theorem 2, the complexity is on the order of the log of $1/\delta$ and $n \log n$ in the size of T.

Theorem 1 has some interesting consequence for the performance of the algorithm. A step will terminate when COMPOSER identifies a transformation with positive incremental utility or when it exhausts the set of possible transformations. If there are many transformations with positive incremental utility, COMPOSER will adopt the one that required the fewest examples. Theorem 1 states that the transformation requiring the fewest examples is not necessarily the one with the highest incremental utility, but rather the one with the highest ratio between its variance and the square of its incremental utility. Thus, the COMPOSER algorithm does not necessarily perform steepest-ascent hill-climbing.

The case where every transformations in T has negative incremental utility points to a potential problem. When this occurs the step will not end until all transformations have been rejected.[8] The last transformation to be rejected is expected to be the one with the maximum ratio of its variance to the square of its mean. As the incremental utility of this transformation tends to zero, its sample complexity increases dramatically. A problem occurs if the transformation has zero expected utility. While this may be unlikely, if such a case arises, it is not clear if the step will even terminate. The algorithm will simply exhaust all of its training examples with out making any improvements in expected utility. Depending on the transformation generator associated with the application, under such circumstances it might make sense to terminate the step early and proceed to a different step in the hill-climbing search. We discuss this possibility in Chapter 6.

8. Actually, there is some probability, due to statistical error, that the algorithm will adopt a transformation with negative incremental utility, therefore ending the step prematurely.

Chapter 3

Customization in the PRODIGY Problem Solver

COMPOSER is a general approach for constructing customized problem solvers. Given a application and method for generating atomic transformations, COMPOSER constructs a composite modification in a mathematically rigorous fashion. While the approach is grounded in statistical theory, COMPOSER's effectiveness must be demonstrated over practical customization problems. In this chapter we describe an application of COMPOSER to learning custom search control strategies for the PRODIGY planning system [Minton88b]. PRODIGY is a well studied planning system that has served as the basis for several learning investigat2ons [Etzioni90b, Knoblock89, Minton88b]. This provides an excellent opportunity to compare COMPOSER's approach to other proposed methods for improving PRODIGY's performance.

3.1 THE APPLICATION

This implementation is constructed within the PRODIGY 2.0 architecture which is available from Carnegie Mellon University. PRODIGY is a general purpose means-ends problem solver. Its architecture is based on the STRIPS planner [Fikes71] with a few enhancements. Plans are identified by depth-first search. Search proceeds by recursively applying four control decisions: (1) choosing a node to expand in the current search space, (2) choosing an unachieved goal at that node, (3) choosing an operator that unifies with the goal, and (4) choosing a binding list for the operator. PRODIGY implements a default control method for each of these four control points and these methods may be modified by the introduction of heuristic knowledge called *control rules.* Control rules are described below.

3.1.1 Problem Distributions

We evaluate COMPOSER's ability to customize PRODIGY on three different domain theories. The theories have been previously reported in the learning literature and are summarized in Appendix A. The STRIPS domain was reported in [Minton88b] It is a problem of a robot moving boxes through interconnected rooms with lockable doors. The AB-WORLD world domain was reported in [Etzioni91]. It is a variant of the standard blocksworld domain, designed to highlight deficiencies in the Minton's PRODIGY/EBL approach. The BIN-WORLD domain was introduced in [Gratch91a] and was designed to highlight deficiencies in both PRODIGY/EBL and Etzioni's STATIC approach. This domain is a simple construction domain.

Problem distributions for the STRIPS domain and AB-WORLD domain were constructed using the automatic problem generators provided with the PRODIGY 2.0 package. Problems for the STRIPS domain were generated using the STRIPS random problem generator. Problems for the AB-WORLD domain were generated using the Blocksworld random problem generator. Following the methodology in [Minton88b], the set of problems were biased by filtering out problems that were judged too difficult or too easy. Problems were only selected if they could be solved by the default PRODIGY control strategy in more than 1 CPU second or less than 80 CPU seconds. Problems for the BIN-WORLD were generated in a distribution designed to highlight deficiencies in PRODIGY/EBL and STATIC. This is described in Appendix A.3.

3.1.2 Expected Utility

In the previous evaluations of learning approaches to PRODIGY (e.g. [Etzioni91, Minton88b]), the evaluation criterion was in terms of the cumulative time in CPU seconds to solve a set of problems. Under the decision-theoretic interpretation of the evaluation criterion, this can be captured by a utility function in terms of the time required to solve a problem. In particular, let the utility of the problem solver over a problem be the negative of the CPU time required to solve it. Maximizing expected utility therefore translates into decreasing the average time to required to solve a problem.

3.2 TRANSFORMATION GENERATOR

Minton introduced a technique for generating atomic modifications to the PRODIGY control strategy. The approach used explanation-based learning (EBL) [DeJong86, Mitchell86] to construct atomic control rules. Sets of control rules can be associated with any of the four control points and act to attenuate the default method. The control rules are condition–action statements which alter the way PRODIGY explores the space of possible plans. By default, the planner lists all operators which unify with an unachieved goal and explores these alternatives depth–first. Control rules change the search by discarding or reordering some alternatives. Figure 6 illustrates a control rule learned by the EBL method in the blocksworld domain. The blocksworld has several operators for clearing a block. RULE–1 asserts that in situations where the block is not held, only consider the UNSTACK operator.

RULE–1: IF *current–node* is ?n
current–goal at ?n is (CLEAR ?x)
(NOT (HOLDING ?x)) is true at ?n
THEN choose operator UNSTACK

Figure 6: An example control rule

Not all control rules increase the efficiency of planning. Control rules avoid search in the plan space, however they introduce the cost of matching their preconditions. A rule is harmful when the precondition evaluation cost exceeds the savings. Furthermore, control rules interact in subtle ways. Without a criterion for choosing among possible rule sets, the learning algorithm quickly degrades performance. Minton introduced a heuristic empirical procedure for addressing the utility problem in this context. This procedure attempts to account for the distributional nature of the incremental utility of individual control rules. Minton calls the overall approach of EBL learning and heuristic utility analysis PRODIGY/EBL. PRODIGY/EBL performs quite well on some domains. Unfortunately, it has since been shown to have undesirable properties. Oren Etzioni illustrated how seemingly innocuous changes to a domain theory result in degraded problem solving performance [Etzioni90b]. We showed that this behavior is due to the utility procedure's inability to correctly estimate distribution information and to handle the composability problem (see [Gratch91a, Gratch91b]). COMPOS-

ER's statistically sound utility estimation procedure corrects these problems and should result in a more effective learning algorithm.

In this application, the EBL technique serves as the transformation generator. The EBL component requires training problems to be solved with PRODIGY using some control strategy (in this case a control strategy is a set of control rules for each of the control points). The component analyzes a trace of each solution attempt, and conjectures new control rules. Each of these control rules is an atomic transformation to one control point of the current control strategy. In Minton's system, all control points are given equal importance so our implementation contains no independence or dominance relations between control points.

3.3 IMPLEMENTATION DETAILS

We used COMPOSER to construct a customizing problem solver for the three applications. We call the resulting implementation COMPOSER/PRODIGY. PRODIGY with no control rules acts as the initial problem solver. The implementation attempts to minimize average CPU. Minton's EBL learning element acts as the transformation generator.[9] Thus, COMPOSER/PRODIGY takes the problem solver with the empty set of control rules as PS_{old} and produces a PS_{new} by incrementally adding control rules one at a time, if they have positive incremental utility.

The implementation has some departures for COMPOSER's default control strategy. The resulting architecture is illustrated in Figure 7. We used Equations 7 and 8 to define the bound function: $Bound(\delta, i\,|T|) = \delta$. As stated in Section 2.3.2.1.1, this allows transformations to be added to T at any time in the utility analysis. We used an initial sample size, n_0, of three, smaller than the default.

We exploited a property of control rules to more efficiently gather incremental utility data points. The implementation extracts incremental utility values for all candidate control rules with an unobtrusive procedure of using the trace of a single solution attempt. To accomplish this, we modified the PRODIGY planner to distinguish between adopted and candidate control rules. Adopted control rules are those which have been shown to have positive incremental utility, added into the control strategy of the PRODIGY planner, and therefore effect PRODIGY's behavior. Candidate rules are those in T that have been proposed by the EBL technique, but not yet validated. When solving a problem, candidate rules are checked, their precondition cost and planning recommendations are recorded, but their recommendations are not acted upon. These recommendations would have served to eliminate search some paths explored by the current strategy. After a problem is solved, the annotated trace can be analyzed to identify those search paths which would have been eliminated by candidate control rules. The time spent exploring these avoidable paths indicates the savings which would be provided by the rule. This savings is compared with the recorded precondition match cost, and the difference is reported as the incremental utility of the control rule for that problem.

3.4 EVALUATION

We evaluated COMPOSER/PRODIGY's performance against four other proposed criteria for selecting beneficial control rules: the heuristic utility analysis of PRODIGY/EBL, the non-recursive hypothesis of STATIC,

9. The EBL unit produces two control rule types: rejection rules and preference rules. Minton has noted that preference rules seem to be less effective. We have verified that PRODIGY/EBL actually produces strategies with higher utility if it is prevented from producing preference rules [Gratch91a]. We disabled the learning of preference rules in this implementation because it enabled a more efficient means of gathering incremental utility data points.

Given: $PS_{old},\ TG(\cdot),\ \delta$

[1] $Strat = \emptyset$; $PS := PS_{old} + Strat$; $\mathrm{T} := \emptyset$; $i := 0$; $n := 0$; $\alpha := Bound(\delta, i, |\mathrm{T}|)$;

[2] While more examples do

[3] $n := n + 1$

[4] $\forall \tau \in \mathrm{T}$: Get $\Delta U_n(\tau|PS)$; $\mathrm{T} := \mathrm{T} \cup TG_{EBL}(PS, P_n)$; /* P_n is the nth example problem. */

[5] $$\alpha\text{–}estimates := \left\{ \tau \in \mathrm{T} : n \geq n_0 \text{ and } \frac{S_n^2(\tau|PS)}{\left[\overline{\Delta U_n}(\tau|PS)\right]^2} < \frac{n}{\left[Q(n,\alpha)\right]^2} \right\}$$ /* Collect all trans. that */ /* have reached statistical */ /* significance. */

[6] $\mathrm{T} := \mathrm{T} - \left\{ \tau \in \alpha\text{–}estimates : \overline{\Delta U_n}(\tau|PS) < 0 \right\}$ /* Discard trans. that decrease expeced utility */

[7] If $\exists \tau \in \alpha\text{–}estimates : \overline{\Delta U_n}(\tau|PS) > 0$ Then /* Adopt τ that most increases expected utility */

[8] $Strat := Strat \bigcup \left\{ x \in \alpha\text{–}estimates : \forall y \in \alpha\text{–}estimates \left[\overline{\Delta U_n}(x|PS) > \overline{\Delta U_n}(y|PS) \right] \right\}$

[9] $PS := PS_{old} + Strat$; $\mathrm{T} := \emptyset$; $i := i + 1$; $n := 0$; $\alpha := Bound(\delta, i, |\mathrm{T}|)$

Return: PS

Defaults: $Bound(\delta, i, |\mathrm{T}|) := \delta$, $Q(n, \alpha) := x \text{ where } \int_x^{\infty} \left(1/\sqrt{2\pi}\right) e^{-0.5y^2} dy = \frac{\alpha}{2}$, $n_0 := 3$

Figure 7: COMPOSER/PRODIGY

a combination of these two approaches, and PALO, a statistical approach similar to COMPOSER but based on a more conservative statistical model. Before discussing the experiments we review these techniques.

3.4.1 PRODIGY/EBL'S Utility Analysis

This is the approach developed by Minton for use in PRODIGY/EBL. The technique adopts transformations with a heuristic utility analysis. As control rules are proposed they are added to the current control strategy. The savings afforded by each rule is estimated from a single example (the example problem from which the rule was learned) and this value is credited to the rule each time it applies. Match cost is measured directly from problem traces and averaged across multiple training examples. If the cumulative cost exceeds the cumulative savings, the rule is removed from the current control strategy. The issue of interactions among transformations is not addressed as estimates are gathered as if there were no interactions.

3.4.2 STATIC's Nonrecursive Hypothesis

STATIC utilizes a control rule selection criterion based on Etzioni's structural theory of utility. The criterion is grounded in the *nonrecursive hypothesis*. This states that "EBL is effective when it is able to curtail search via nonrecursive explanations" [Etzioni90a p. 6]. The hypothesis admits several interpretations. The strongest interpretation is that transformations have positive incremental utility, regardless of problem distribution, if they are generated from nonrecursive explanations of planning behavior (i.e. no predicate in a subgoal is derived using another instantiation of the same predicate). A weaker reading is that a composite strategy will improve expected utility if it is constructed from non-recursive elements (admitting that some transformations will have negative incremental utility, but a set of non-recursive transformations will improve performance). The issue of interactions between transformations is also not addressed. STATIC applies this criterion to control rules but the issue is important in macro–operators as well [Letovsky90, Subramanian90].

STATIC has out performed PRODIGY/EBL's on several domains. The nonrecursive hypothesis is cited as the principle reason for this success [Etzioni90b][10]. This claim is difficult to evaluate as these algorithms can generate very different control rules. We wish to focus on the effectiveness of the non-recursive hypothesis, and therefore must remove the complicating factor of a different rule generator. To achieve this goal we constructed the NONREC algorithm, a re-implementation of STATIC's nonrecursive hypothesis within the PRODIGY/EBL framework. NONREC replaces PRODIGY/EBL's empirical utility analysis with a syntactic criterion which only adopts nonrecursive control rules. This acts as a filter, only allowing PRODIGY/EBL to generate non-recursive rules.

3.4.3 A Composite Algorithm

Etzioni has suggested that the strengths of STATIC and PRODIGY/EBL can be combined into a single approach [Etzioni90a]. He proposed a hybrid algorithm which embodies several advancements including a two layered utility criterion. The nonrecursive hypothesis acts as an initial filter, but the remaining nonrecursive control rules are subject to utility analysis and may be later discarded.

We implemented the NONREC–UA algorithm to test this hybrid criterion. As control rules are proposed by PRODIGY/EBL's learning module, they are first filtered on the basis of the nonrecursive hypothesis. The remaining rules undergo utility analysis as in PRODIGY/EBL.

3.4.4 PALO'S Chernoff Bounds

Greiner and Cohen have proposed an approach similar to COMPOSER's [Greiner92b]. The Probably Approximately Locally Optimal (PALO) approach also adopts a hill–climbing technique and evaluates transformations by a statistical method. PALO differs in its stopping rule and and that it incorporates a criterion for when to stop learning. PALO terminates learning when it has (with high probability) identified a near–local maximum in the transformation space. Our evaluation focuses on the different stopping rule which is based on Chernoff bounds.

Chernoff bounds provide a much more conservative model of the discrepancy between a the sample mean and true mean of a distribution. As a result, PALO provides stronger bounds on statistical error at the cost of more examples. This means that if the user specifies an error level of δ, the true error level will never exceed δ, and may in fact be much lower.[11] Our PALO–RI algorithm evaluates this approach. Like COMPOSER,

10. Etzioni and Minton have subsequently suggested that some of the success of STATIC is due to the fact that its global analysis allows for more concise rules [Etzioni92].

11. In addition to the conservative stopping rule, PALO adopts the conservative definition of $Bound(\delta, i, |T|)$ that follows from adopting Equations 5 and 10.

PALO–RI uses a candidate set of rules. In this case the size of the set is fixed before learning begins. A candidate is adopted when the following stopping rule holds:

$$\sum_{i=1}^{n} X_{r,i} \; > \; \Lambda_r \sqrt{2n \ln\left(Cs^2\pi^2/(3\delta)\right)}$$

where $X_{r,i}$ is the incremental utility of rule r on problem i, C is the maximum size of the candidate set, s is one plus the number of rules in the current control strategy, and Λ_r is a is the maximal per problem ΔUTILITY of rule r. The parameter C is the size of the candidate set. We discuss the setting of the various parameters in the next section.

3.4.5 Experiments

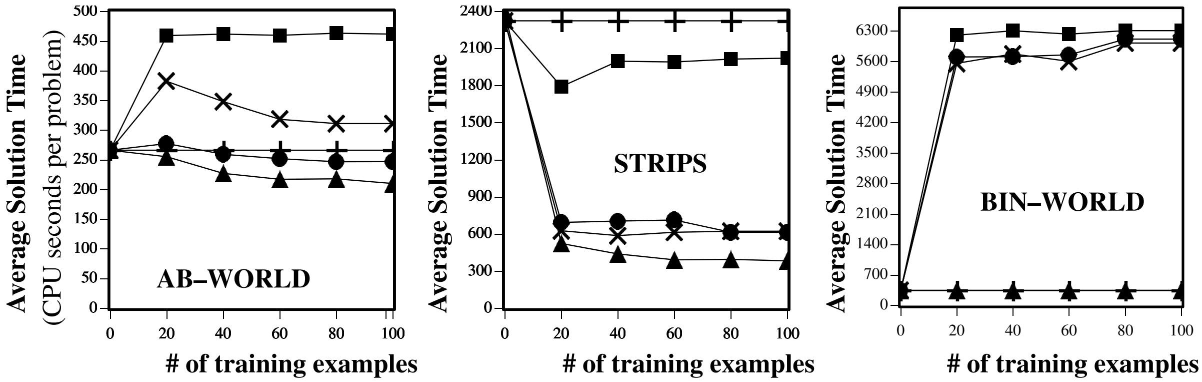
We investigated the STRIPS domain from [Minton88a], the AB-WORLD domain from [Etzioni90a] for which PRODIGY/EBL produced harmful strategies, and the BIN-WORLD domain from [Gratch91b] which yielded detrimental results for both STATIC's and PRODIGY/EBL's learning criteria. Results are summarized in Figure 8. In each domain the algorithms were trained on 100 training examples drawn randomly from a fixed distribution. The problem distributions were constructed using the random problem generator provided with the PRODIGY architecture. The current control strategy was saved after every twenty training examples.[12] The graphs illustrate learning curves where the independent measure is the number of training examples and the dependent measure is execution time in CPU seconds over 100 test problems drawn from the same distribution. This process was repeated eight times, using different but identically distributed training and test sets. Values in Figure 8 represent the average of these eight trials. "Rules Added" indicates the average number of rules learned by the algorithm; "Train Time" is the number of seconds required to process the 100 training examples; "Test Time" is the number of seconds required to generate solutions for the 100 test problems.

COMPOSER and PALO–RI require a error parameter δ which is set at 10%. PALO–RI's behavior is strongly influenced by parameters whose optimal values are difficult to assess. We tried to assign values close to optimal given the information available to us.[13]

During the evaluation, it was apparent that PALO–RI would not adopt any transformations within the 100 training examples. We tried to give the algorithm enough examples to reach quiescence but this proved too expensive. The problem is twofold — first, too many training examples were required; secondly, and as a consequence of the first problem, the candidate set grew large since harmful rules were not discarded as quickly as in COMPOSER/PRODIGY. This increased the cost to solve each training example. To collect statistics on PALO–RI we only performed one instead of eight learning trials. Furthermore, we terminated PALO–RI after the first transformation was adopted or 10,000 examples, whichever came first.

12. PRODIGY/EBL's utility analysis requires an additional settling phase after training. Each control strategy produced by PRODIGY/EBL and NONREC+UA received a settling phase of 20 problems following the methodology outlined in [Minton88a].

13. C was fixed based on the size of the candidate list observed in practice. In the best case a rule can save the entire cost of solving a problem, so for each domain, lambda for each rule was set at the maximum problem solving cost observed in practice. AB-WORLD – C=30 lambda=15; STRIPS – C=20, lambda=100; BIN-WORLD – C=5, lambda=150.



	AB–WORLD			STRIPS			BIN–WORLD		
ALGORITHM	Rules Added	Train Time	Test Time	Rules Added	Train Time	Test Time	Rules Added	Train Time	Test Time
No Learning	—	—	266	—	—	2323	—	—	346
COMPOSER/PRODIGY	1	1667	210	4	4133	382	0	3425	346
PRODIGY/EBL	11	1253	311	20	3775	622	2	6383	6020
NONREC	17	1253	462	25	4012	2021	4	6710	6305
NONREC+UA	9	1259	247	10	3821	614	2	6359	6110

Performance after first rule adopted	PALO–RI		
	Train Exmpls	Train Time	Test Time
AB–WORLD	6069	104,387	182
STRIPS	1182	41,370	1223
BIN–WORLD	10,000+	—	346

Figure 8: Summary of emprical results for the PRODIGY application. Learning curves showing performance as a function of the number of training examples and table of experimental results.

3.5 ANALYSIS

The results illustrate several interesting features. The implementation developed with COMPOSER exceeded the performance of all other approaches in every domain. The learned strategies yielded higher expected utility and were more succinct (contained fewer control rules). In AB-WORLD and STRIPS, COMPOSER/PRODIGY identified beneficial control strategies. In BIN-WORLD the algorithm did not adopt any transformations. It does not appear that any control rule improves performance in this domain. It should be stressed that all algorithms use the same transformation generator. Therefore the results represent differences in approaches to the customization problem rather than in the vocabulary of transformations.

As expected, COMPOSER/PRODIGY and PALO–RI had the highest learning times as they are more thorough in their assessment of incremental utility. The one exception was BIN-WORLD where COMPOSER/PROIDGY quickly discarded a control rule with high match cost which PRODIGY/EBL, NONREC, and NONREC+UA retained. An encouraging result is that COMPOSER/PRODIGY's learning times were not substantially higher than the non–statistical algorithms. PALO–RI's learning times were significantly higher.

The results cast doubts on the efficacy of the nonrecursive hypothesis. NONREC yielded the worst performance on all domains. Even in conjunction with utility analysis the results were mixed — benefit on the AB-WORLD, slightly worse than utility analysis alone in STRIPS, and worse than no–learning in BIN-WORLD. A post–hoc analysis of control strategies indicated that the best rules were nonrecursive, but many nonrecursive rules were also detrimental. In BIN-WORLD, non-recursive rules produced substantial performance degradation. Thus it appears that nonrecursiveness may be an important property but is insufficient to ensure performance improvements. These results are interesting since Etzioni reports that STATIC outperforms PRODIGY/EBL and No Learning in AB-WORLD. The nonrecursive hypothesis cannot completely account for this difference. We attribute the difference to the fact that STATIC and NONREC entertain somewhat different sets of control rules. NONREC was constrained to use the vocabulary which was available to PRODIGY/EBL while STATIC has its own rule generator. This conjecture was recently supported by Minton and Etzioni [Etzioni92].

Finally, although PALO–RI did not improve performance within the 100 training examples, we believe that if it were given sufficient examples it would out perform all other approaches. With extended examples it did exceed COMPOSER/PRODIGY's performance in AB-WORLD. This is because the PALO approach commits to transformations with highest incremental utility while COMPOSER balances incremental utility against variance. Unfortunately the cost of PALO's performance improvement is very high, both in terms of examples and learning time. Thus, while COMPOSER may identify somewhat less beneficial strategies, it achieves much faster convergence.

Chapter 4

Customization in Deep Space Network Scheduling

While the PRODIGY implementation illustrates that COMPOSER compares favorably with other customization approaches, tests on artificial domains have their limitations. COMPOSER's statistical assumptions depend on properties of the distribution of problems. Therefore it is imperative to test COMPOSER on real-world domains and problem distributions. In this chapter we describe an application of COMPOSER to the problem of customizing a scheduler operating in the application of spacecraft communication scheduling. The problem distributions used in this evaluation are based on NASA satellite communication requirements and orbit data.

4.1 THE APPLICATION

The scheduling problem is a complicated real world task that has proved challenging to state-of-the-art scheduling techniques. The problem is to allocate communication requests between earth-orbiting satellites and the three 26-meter antennas at Goldstone, Canberra, and Madrid. These antennas make up part of the Deep Space Network (DSN) that is responsible for communication with earth-orbiting and inter-planetary spacecraft. Each satellite has a set of constraints, called project requirements, that define its communication needs. For example, the Nimbus-7 satellite requires at least four 15-minute communication slots per day, and these slots cannot be greater than five hours apart. Two factors complicate the problem. First, antennas are a limited resource – two satellites cannot communicate with the same antenna at the same time. Second, satellites can only communicate with certain antennas at certain times, depending on their orbits.

Schedules are constructed each week. A weekly scheduling problem is defined by three elements: (1) the set of satellites to be scheduled, (2) the constraints associated with each satellite, and (3) a set of time periods specifying all temporal intervals when a satellite can legally communicate with an antenna. Two time periods conflict if they use the same antenna and overlap in temporal extent. A valid schedule specifies a non-conflicting subset of all possible time periods where each project's requirements are satisfied.

4.1.1 The Scheduling System

LR-26 is a heuristic approach to the scheduling problem [Bell92] developed at the Jet Propulsion Laboratory. It provides a good platform for customization as it can be modified easily to incorporate alternative heuristic strategies. Furthermore, it uses an expert crafted control strategy that provide a challenging base-line to judge COMPOSER's effectiveness. There is also significant motivation to improve the effectiveness of the default control strategy. If LR-26 is chosen to replace the human scheduler, it will serve as one module in a larger interactive system. This system must make many repeated calls to the scheduler to compare variants of each weekly schedule. For this reason, LR-26 must provide solutions in a timely fashion.

Scheduling is formulated as a 0-1 integer programming problem [Taha82]. This is a methodology for finding an assignment to integer variables that maximizes the value of an *objective function*, subject to a set of linear constraints. The objective function characterizes the “value” of the solution. Many constraint satisfaction problems (CSP) are easily cast as integer programming problems [Mackworth92]. In the DSN domain, time periods are treated as 0-1 integer variables (0 if the time period is excluded from the schedule or 1 if it is included), the objective is to maximize the number of time periods in the schedule subject to the project requirements and temporal conflict constraints which are expressed as sets of linear inequalities. Integer programming is NP-hard, and the size of the scheduling problems makes the conventional approach impractical: a typical problem has approximately 650 variables and 1300 constraints. LR-26 embodies a heuristic approach called *lagrangian relaxation* [Fisher81]. Lagrangian relaxation requires identifying a set of constraints that, if removed, make the problem computationally easy. These constraints are “relaxed,” meaning they no longer act as constraints but instead modify the objective function. A relaxed objective function is automatically generated such that satisfying relaxed constraints increases the value of the relaxed solution. The relaxed problem is by definition easy to solve and often finding the highest value relaxed solution solves the original problem. Furthermore, each relaxed constraint has a weight associated with it when it is added to the objective function. By systematically adjusting these weights and re-solving the relaxed problem, a solution to the unrelaxed problem is often efficiently discovered. Even if the unrelaxed problem cannot be solved in this manner, this weight adjustment cycle can move the scheduler closer to a solution, allowing the unrelaxed solution to be discovered with less search. LR-26 relaxes inter-antenna constraints. This representation facilitates an efficient implicit representation of temporal conflict constraints, which make up more than half of all constraints in a typical problem.

LR-26 combines lagrangian relaxation with standard constraint satisfaction search techniques. The scheduler performs depth-first search through a space of partial schedules. A variable is assigned a value of *in* if the associated time period is included in the partial schedule, *out* if it is excluded from the partial schedule. The scheduler constructs a complete schedule by incrementally extending the partial schedule. First it attempts to completely extend the schedule using the lagrangian relaxation method. If the relaxed solution satisfies all constraints it is returned. Otherwise, a set of possible extensions to the partial schedule is created and these are recursive explored. Extensions are created by choosing an unsatisfied constraint, identifying a set of uncommitted variables in the constraint, and assigning possible values to these variables. The set of extensions are placed on a stack to implement the depth-first search. The search continues until a solution is uncovered or a time-bound is reached. Currently the scheduler implements a time-bound of five CPU minutes. Any problem not solved within this bound is deemed unsolvable.

LR-26 can be viewed as a recursive application of four control decisions: (1) decide on a lagrangian weight adjustment scheme, terminating if a viable solution is found; otherwise (2) choose an unsatisfied constraint, (3) determine a set of extensions to the partial schedule that satisfy the constraint, and (4) determine an order to explore these extensions. The scheduler embodies a heuristic method for each of these control operations. These operations provide natural points at which to insert alternative learned methods.

4.1.2 Problem Distribution

To evaluate COMPOSER in this domain, we constructed a distribution of scheduling problems using the requirements and time periods of satellites using the deep space network. Ideally, we would use the identical problem distribution faced by the human experts in this domain. Unfortunately, not all of this information is in electronic form and thus is difficult to present to the LR-26 scheduler. There does, however, exist a large electronic database of information for many of the satellites using the deep space network. We used this database to construct a large body of scheduling problems that are representative of, if not identical to, the type of problems faced by the human schedulers. Problems are generated by randomly choosing combinations of projects from the available data. The requirements and time periods represent the requirements and time periods of actual satellites. The primary difference between these and actual problems lies in the particular combinations of projects that appear in the schedule.

We performed a initial evaluation of the LR-26 scheduler on these generated problems and observed that some problems could not be solved by the scheduler using any of several search control strategies even with large resource bounds. This is consistent with the observation that the scheduling problem is inherently NP-Hard – there will be some problems that cannot be efficiently solved, even with good heuristics. These problems dilute any performance improvement that we might gain through learning. As in the PRODIGY evaluation, we removed this complicating factor by constructing our problem distribution without these unsolved problems. These problems were identified by solving each randomly generated problem multiple times using about twenty different search strategies (the strategies were identified during our pilot investigations). If a problem could not be solved by any of the strategies within the time bound, it was not added to the experimental distribution. For comparative purposes we include a secondary set of experiments that incorporate these unsolved problems. For a complete description of how training examples are generated, see [Gratch93b].

4.1.3 Expected Utility

In the DSN application a chief concern is with the computational efficiency of the scheduler. There is a strong need that the scheduler return quickly on average. This behavioral preference can be expressed by a utility function related to the computational effort required to solve a problem. As the effort to solve a problem increases, the utility of the problem solver on that problem should decrease. For the evaluation, we characterized this preference by making utility the negative of the CPU time required by the scheduler on a problem.

4.2 TRANSFORMATION GENERATOR

LR-26 combines lagrangian relaxation and constraint satisfaction search techniques to increase scheduling efficiency. Nevertheless, scheduling is still quite expensive. While the problems are of sufficient complexity that some search is unavoidable, alternate search control methods can drastically influence the amount of search required, especially as there is substantial repetition in this domain. Many satellites use the antennas for many years, and their project requirements vary little across weeks. As the available time periods are dictated by spacecraft orbits, the space of potential communication slots also contains significant regularity. This suggests that heuristics can be crafted to exploit this regularity to improve performance.

Many heuristics have been suggested to improve scheduling efficiency. Often these heuristics are stated as general principles (e.g. “first instantiate variables that maximally constraint the rest of the search space” [Dechter92]) and there may be many ways to realize them in a particular scheduler and domain. Furthermore, there are almost certainly interactions between methods used at different control points that makes it difficult to construct a good overall strategy. These factors conspire to make manual developing and evaluation of heuristics a tedious, time consuming task that requires significant knowledge about the domain and scheduler.

As a result, only a limited set of alternate heuristics were considered in LR–26's development. We formalized a much larger set for automatic consideration.

As described previously, a there are four basic control decisions in LR–26: decide on a weight adjustment scheme, choose an unsatisfied constraint, determine a set of alternative solutions for the constraint, and determine an ordering on these alternatives. Decisions are resolved by providing ranking functions to order alternative decisions. Methods for these decisions consist of sorting alternatives by some criterion and then choosing the highest ranked alternative. We allowed for both a primary and secondary method to rank candidate constraints, so there are effectively five control points. A control strategy consists of a particular heuristic method for each of the five control points. For LR–26 there are 4 alternative methods for the lagrangian weight adjustment control point, 9 alternative methods for each constraint selection sort (9 for the primary sort and 9 for the secondary sort – excluding the primary sort but including the possibility of no secondary sort), 2 alternative methods for constructing alternative solutions to a constraint, and 4 alternative methods for ordering alternative solutions to a constraint. Exhaustively searching the space of possible control strategies is in general computationally infeasible. As there are roughly $4 \times 9 \times 9 \times 2 \times 4 = 2592$ possible strategies in the LR-26 control strategy search space, to exhaustively search the control strategy space, taking a significant number of examples per strategy (fifty) at a cost of 5 CPU minutes per problem would require approximately 450 CPU days.

Some example methods are as follows. For the weight adjustment, methods perform adjustments at every search node, perform only at root node, never perform adjustments. Methods for selecting constraints involve features of the constraints and constraint graph. Prefer the shortest constraint (e.g. the one that mentions the fewest unbound variables) and prefer the constraint that mentions the variables that temporally conflicts with the most other variables are examples of constraint selection methods (recall variables correspond to time-periods). For constructing alternative solutions, there were 2 methods: force the first variable in the constraint into the schedule for one child and out for the second child; and construct a child for each variable in the constraint, forcing that variable into the schedule. Examples of alternative ordering methods include preferring high conflict variables and preferring low conflict variables[14]. These heuristics can be viewed as variable ordering heuristics from the CSP literature [Dechter92].

COMPOSER requires this search to be structured for hill-climbing search. A simple way to organize the search through this strategy space would be to treat all control points a equal and consider all single method changes to a given control strategy. This was the strategy in our PRODIGY implementation of COMPOSER [Gratch92a]. In the DSN application, we used our knowledge of the scheduler to take advantage of interactions (or lack thereof) between control points to structure the search. The intent of this organization was to reduce the branching factor in the control strategy search and improve the expected utility of locally optimal solutions. This approach led to a transformation generator that implements a layered search through the strategy space. Each control point is assigned to a level. The control strategy space is search by evaluating all

14. This is a prime example of the difficulty of determining good heuristics. Selecting high conflict variable has the benefit of rapidly forcing many time-periods to be in or out of the schedule (e.g. reducing the number of steps to solution). Selecting low conflict intervals advocates adding those time-periods which cost little, and thus may satisfy constraints without causing conflicts [Dechter92].

combinations of methods at a single level, adopting the best combinations, and then moving onto the next level. The organization is shown below:

Level 0: weight adjustment
Level 1: constructing alternatives
Level 2: secondary constraint sort, child sort
Level 3: primary constraint sort

The weight adjustment and alternative construction control points were separated because they are relatively independent from the other control points. While there is clearly some interaction between weight search, alternative construction, and the other control points, a good selection of methods for pricing and alternative construction should perform well across all constraint and child sorts. The primary constraint sort was separated into another level because it was the sole control point that the implementor of LR–26 had spent time experimenting and optimizing. Thus, we believed that it was unlikely the default strategy could be improved upon, and hence relegated to a separate level.

This transformation generator leads to a layered hill-climbing search. It first proposes alternative weight adjustment methods, then alternative construction methods, then combinations of secondary constraint sort and child sort methods, and finally primary constraint sort methods. Each choice is made given the previously adopted methods. Searching the structured space involves evaluating at most $4+2+9\times4+9=51$ strategies.

This leveled search can be viewed as the consequence of asserting certain types of relations between control points. Independence relations indicate cases in which the utility of methods for one control point is roughly independent of the methods used at other control points. Dominance relations indicate that the changes in utility from changing methods for one control point are much larger than the changes in utility for another control point. Finally, inconsistency relations indicate when a method M_1 for control point X is inconsistent with method M_2 for control point Y. This means that any strategy using these methods for these control points need not be considered.

4.3 IMPLEMENTATION ISSUES

We used COMPOSER to construct a customizing problem solver for the DSN applications. The resulting implementation is called COMPOSER/DSN. The expert control strategy serves as PS_{old}. This can be incrementally modified by transformations proposed by the generator. The implementation attempts to minimize average CPU time to produce a scheduling. Unlike the PRODIGY implementation we were unable to exploit the structure of transformations to gather incremental utility data points. The implementation uses the default method of solving each problem m+1 times; once with the current control strategy and once using each of the m transformations in T. The resulting implementation is identical to Figure 4.

4.4 EVALUATION

COMPOSER/DSN should, with high probability, improve the expected utility of the scheduler over the distribution of problems. Two basic claims can be tested empirically: First, COMPOSER/DSN should identify transformations that improve the expert strategy. Second, it should identify these transformations with the confidence predicted by the statistical theory. Besides testing these claims, we are also interested in two secondary questions. How quickly does the technique improve expected utility (e.g., how many examples are required to make statistical inferences?). Also, many problems are unsolvable within five minutes with the expert strategy. Can COMPOSER/DSN improve the number of solvable problems?

For the experiments a learning run consists of 300 randomly selected training examples. The expected utility of all learned strategies is assessed on an independent test set of 1000 test examples. A measurement of learning rate is determined by recording the strategy learned by COMPOSER/DSN after every 20 examples. Thus we can see the result of learning with after twenty examples, after forty examples, etc. To assess statistical error, we perform twenty runs of the algorithm on twenty distinct training sets. The confidence parameter, δ, was set to the standard value of 5%. The initial sample size parameter, n_0, was set to the default value of fifteen.

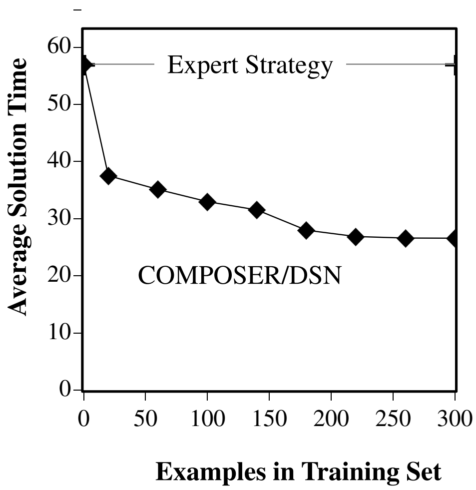
Recall that we purposely excluded several inherently difficult scheduling problems from the problem distribution. These problems, if added to the problem distribution, should make learning more difficult as no strategy is likely to provide a noticeable improvement within the five minute resource bound. Nevertheless, it is important to show that COMPOSER/DSN does not require the elimination of these problems. To test this we created a new distribution by incorporating these problems into the original distribution and repeated the experiments. Results for both sets of experiments are shown in Figure 9. The original distribution is called Distribution 1 while the second is referred to as Distribution 2.

4.5 ANALYSIS

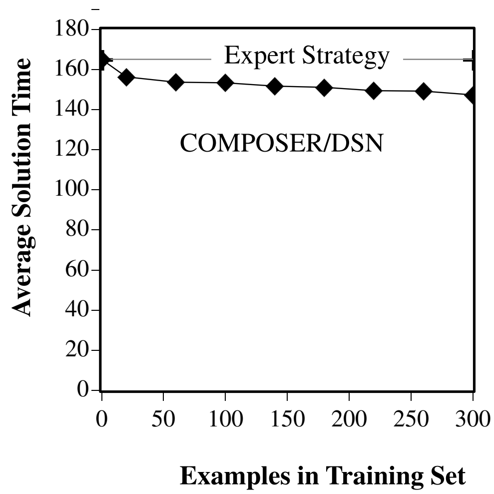
The results support the two primary claims. For Distribution 1, the algorithm learned search control strategies that yielded a significant improvement in performance. It reduced the average time to solve a problem from 57 to 27 seconds (a 53% improvement). There was modest variance in the expected utility of the strategies learned in the twenty trials. The best of these strategies required only 12 seconds on average to solve a problem (an improvement of 78%). The observed statistical accuracy remained well within the theoretically predicted bound: of 93 transformations adopted across the twenty trials, only 3% decreased expected utility. It took an average of 58 examples to adopt each transformation. The human expert strategy was unable to solve 7% of the scheduling problems within the resource bound. One strategy learned by COMPOSER/DSN reduced this number to 3%.

For Distribution 2, learned strategies reduced the average solution time from 165 to 147 seconds (an 11% improvement). The best learned strategies required 160 seconds on average to solve a problem (an improvement of 15%). The observed statistical accuracy did not significantly differ from the theoretically predicted bound: of 107 transformations were adopted across the trials, only 6% decreased expected utility. The introduction of the difficult problems resulted in higher variance in the distribution of incremental utility values, and as predicted by Theorem 1, this results in a higher sample complexity: an average of 108 examples to adopt each transformation. Some improvement was noted even on the "unsolvable" problems. One strategy learned by COMPOSER/DSN increased the number of solvable problems from 51% to 59% (a 16% improvement).

Most of the learned improvement came from improved methods for lagrangian weight adjustment, secondary constraint sorting, and child ordering. The best learned strategy performed no weight adjustment, preferred constraints that contained time periods with high levels of temporal conflicts, and, among child nodes, prefer alternatives with few temporal conflicts. This result suggests that there is no utility in using lagrangian



Distribution 1



Distribution 2

Summary of Results			**Dist. 1**	**Dist. 2**
Avg. Solution Time seconds per prob.	Expert Strategy		57	165
	Learned Strategies	avg. across trials	27	147
		best strategy	12	140
		worst strategy	36	160
Statistical Error		predicted	5%	5%
		observed	3%	6%
Solution Rate % of solvable probs.	Rate Expert Strategy		93%	51%
	Learned Strategies	avg. across trials	95%	54%
		best strategy	97%	59%
		worst strategy	92%	50%

Figure 9: Summary of empirical results from the DSN application. Learning curves showing performance as a function of the number of training examples and table of experimental results. Results are provided for original distribution (Distribution 1) and the distribution including unsolved problems (Distribution 2).

weights to improve the first relaxed solution. One interpretation of the acquired constraint and child methods is that search should proceed by first identifying a highly constrained constraint, and then choose the least constrained way of satisfying it. It is interesting that this is consistent with the general recommendations in the CSP literature [Dechter92].

Chapter 5

Sensitivity Analysis

The behavior of the COMPOSER algorithm is influenced by parameters that can be changed by an application designer. Such parameters are a mixed blessing, particularly when their semantics is difficult to specify. For example, the confidence parameter, δ, has a clear semantics: decreasing δ increases statistical accuracy at the cost of more examples. On the other hand, the effects of initial sample size, n_0, is less clearly defined. It effects the accuracy of statistical approximations, which only indirectly influence the behavior of the algorithm.

In this chapter we investigate the sensitivity of COMPOSER to various parameter settings. Section 2.3 suggested several aspects of the COMPOSER algorithm that can be modified. Ideally we would analyze the effect of all of these recommendations, however this is prohibitively expensive. Instead we focus on two parameters which we find are the most useful to adjust, the confidence parameter and the initial sample size. We would like to show that COMPOSER's behavior changes gracefully and predictably with changes to these parameters. If not, COMPOSER will be inconvenient to use as one may have to expend considerable effort just getting the parameters set correctly. We investigate COMPOSER's performance over the Deep Space Network domain as this affords a test over a realistic distribution of problems.

5.1 EVALUATION

We evaluate the behavior of COMPOSER/DSN as a function of the statistical confidence parameter, δ, and the initial sample size parameter, n_0. Recall that δ is a bound on the probability that COMPOSER will adopt a problem solver with worse expected utility. The initial sample size effects the adequacy of the normal approximation that underlies the Nádas stopping rule. Changing the settings of these parameters will effect COMPOSER's performance. To characterize this effect, we measure several attributes of behavior including statistical error exhibited by the algorithm, the number of examples required to perform statistical inference, and the expected utility of customized problem solvers. From COMPOSER's mathematical framework, we can make several predictions about the effect of these parameters on these three aspects of the algorithm's behavior.

5.1.1 Statistical Error Predictions

Due to the random nature of selecting training examples, there is some probability that the training data will not represent the actual distribution of problems, thereby disrupting COMPOSER's ability to perform statistical inference. The confidence parameter, δ, acts as a bound on this probability. As it is defined by Equation 3, δ is a bound on the probability that the customized problem solver produced by COMPOSER, PS_{new}, is of less expected utility than the original problem solver, PS_{old}:

$$Pr(E_D[U(PS_{new})] < E_D[U(PS_{old)}]) \leq \delta \tag{3}$$

Call δ the *overall bound.* The function *Bound*(δ, *i*, |T|) translates this overall bound into a bound on each incremental utility estimate. This translation is in two steps. First, there is a mapping from the overall bound, δ, to a bound on the error at each step in the hill-climbing search, β_i. Call β_i the *step bound* for the *i*th step. Second, the step bound is mapped to a bound on the error on each incremental utility estimate, α. Call this the *estimate bound.* By default (Equations 5 and 8) $\beta_i = \delta$ and $\alpha = \beta_i/|T|$. Finally, the estimate bound is used by the Nádas stopping rule to compute a sample size for each transformation. These sample sizes will result in incremental utility estimates that satisfy the overall bound of Equation 3, subject to the statistical assumptions outlined in Chapter 2.

These bounds are mathematical predictions and based on statistical assumptions. COMPOSER's actual behavior may be somewhat different. For example, the effectiveness of the normal approximation is based on an assumption of a sufficiently large initial sample size (sufficiently large n_0). The *observed overall error* is the probability that COMPOSER produces a problem solver with less expected utility than the original problem solver, when executed on a specific application. The *observed step error* is the probability that COMPOSER adopts a transformation with negative incremental utility at a step. These observed errors can be estimated by performing multiple experimental runs of the system over an application.

We can make two basic predictions about the relationship between the predicted bounds and the observed error rates with respect to the parameters δ and n_0. First, as we increase the confidence (increase $1/\delta$) we should see a reduction in the observed error. Second, as we decrease the initial sample size (decrease n_0), the normal approximation will become poorer, and the observed error will increase. Our basic claim is that COMPOSER's statistical assumptions are adequate to achieve all requested values of statistical error (the observed error should remain below δ for all settings). However, we expect for very small settings of n_0, the normal approximation will be sufficiently bad to disrupt this property.

5.1.2 Sample Complexity Predictions

At each step in the hill-climbing search, COMPOSER requires some number of examples, either to determine the next transformation to apply or to recognize that no transformation produces an improvement. The *sample complexity* is the number of examples required to complete one of these steps. Theorem 2 outlines the basic relationship between the confidence parameter and this number. Based on this theorem we can predict that the number of examples will grow at most quadratically in $1/\delta$. In that theorem we also observe that asymptotically the sample complexity grows with the log of $1/\delta$. It is more difficult to test the log observation over an application, however, as this approximation is only valid for large values of $1/\delta$, which in turn result in prohibitively high sample complexities.

The initial sample size, n_0, will also impact the sample complexity. For a given confidence level, there exists some sample size that guarantees that the estimate bound is satisfied, meaning if COMPOSER based its incremental utility estimates on this number of examples, it would guarantee the error in its estimates remained below the bound of α. The Nádas stopping rule approximates this sample size with its statistical technique by first taking n_0 examples, and then sequentially taking a new examples and testing the rule. If the required number of example is less than n_0, COMPOSER will take too many examples. Thus, for very large n_0, we the sample complexity should grow linearly with n_0.

Even when the sample size is less than the required sample size, we expect an increasing trend in sample complexity with increases in n_0. This is because small values of n_0 can result in a poor normal approximation, which in turn can mislead the system into having greater confidence in its estimates than is justified. Sometimes this mistaken confidence is so great as to have the Nádas stopping rule terminate prematurely, thereby lowering the average sample complexity.

5.1.3 Average Improvement Predictions

Each run of the COMPOSER algorithm produces a problem solver with some expected utility (recall that expected utility is the average utility of the problem solver with respect to the application's distribution of problems). The improvement resulting from a run is the difference in expected utility between the original and customized problem solver. Because of the random nature of the algorithm, if COMPOSER is run multiple times, each time may result in a different problem solver and, therefore, a different improvement. For this reason it is useful to consider the average improvement across multiple runs.

We expect that the average improvement of the customized problem solver to increase as we increase the confidence factor. This is because a customized problem solver is constructed by applying a sequence of transformations. Increasing the confidence decreases the likelihood that a member of that sequence will have negative incremental utility.

We also expect that the average improvement to decrease as we decrease n_0. This is because for small values of n_0, the normal approximation will be poor, resulting in an increase in the observed overall error.

5.1.4 Experiments

As in Chapter 4, a single learning trial consists of training COMPOSER on 300 randomly selected training examples. Because of the statistical nature of COMPOSER, multiple trials are required to produce meaningful results. The evaluation consists of a set of experiments for each setting of δ and n_0. The reported results of an experiment are averaged over of eighty learning trials.

For the sensitivity analysis we compared several values for δ and n_0. To be consistent with statistical learning theory evaluations, we evaluated the the effect of the confidence parameter by considering linear changes to $1/\delta$. The analysis compares seven different values $1/\delta$ and six different values for n_0, resulting in 7×6 experiments. We chose values of $1/\delta = 10, 30, 50, 70, 90, 110, 130$ corresponding to δ values of 0.1, 0.0333, 0.02, 0.01429, 0.0111, 0.0091, and 0.0077 respectively. For n_0 we chose values of 3, 7, 11, 15, 19, and 23. For evaluation of observed statistical error we tracked both the observed overall error and the observed step error.

Results are displayed in graphs where several values of $1/\delta$ are compared for a single value of n_0. The results of each set are displayed in three graphs, one for each aspect of behavior. A reference curve is provided with each graph to help compare results across experiments. For the statistical error the reference curve represents the predicted overall bound. The sample complexity graphs illustrate a linear reference curve and the average improvement graphs have a constant reference curve.

In these experiments we drew incremental utility values from a large cache of values saved from the previous experiment, rather that resolving problems from scratch. This enabled us to efficiently perform large number of trials in this analysis. Results are summarized in Figures 10 and 11.

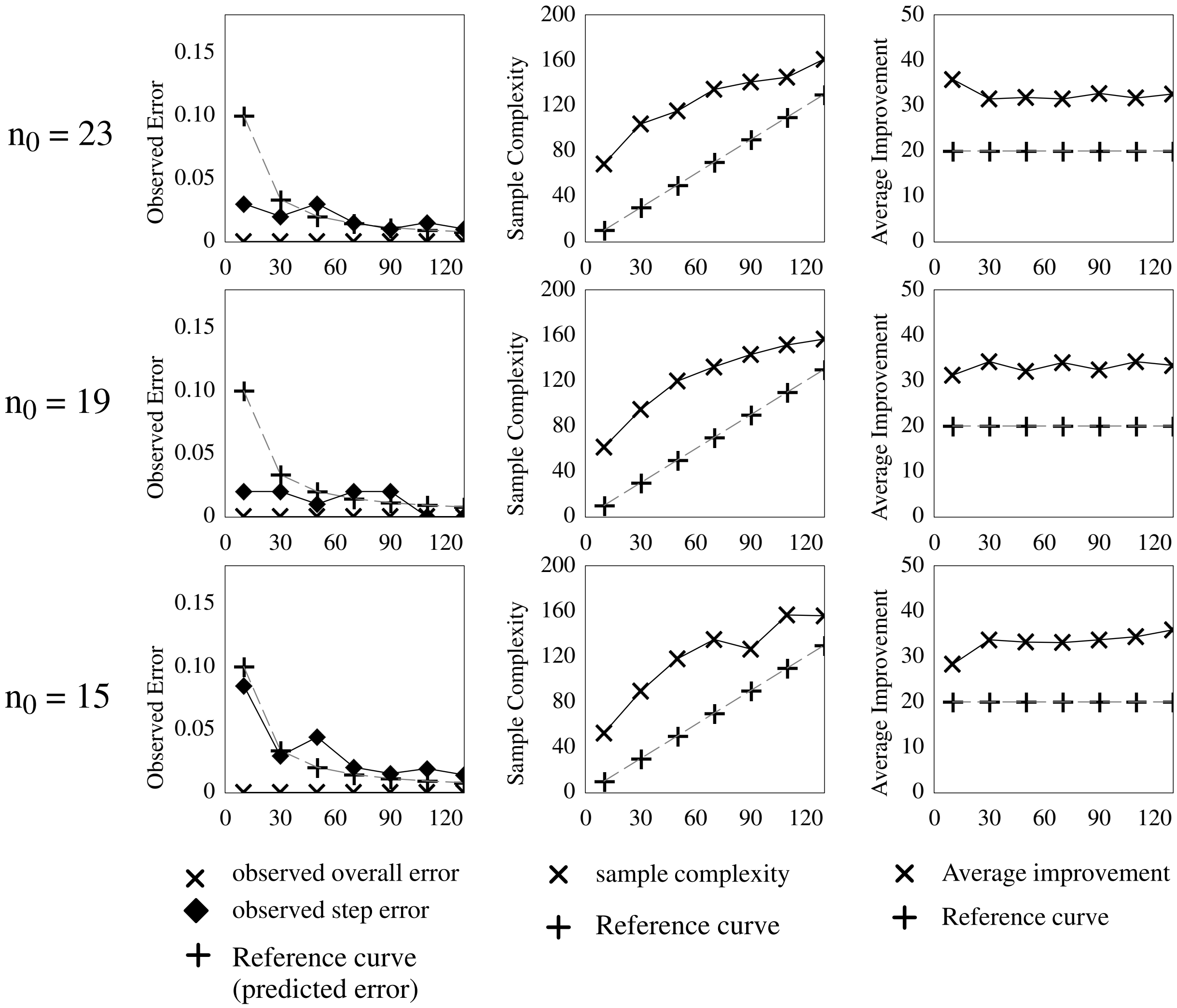


Figure 10: Empirical results of sensitivity analysis. Results for n_0 = 23, 19, and 15. Each row indicates a different initial sample size. Each column shows a different attribute of performance. Dotted curves indicate reference curves.

5.1.5 Observed Statistical Error Results

The results confirm the two basic predictions. First, the observed overall error decreased as we increased the requested confidence. Second, the overall error increased as we decreased the initial sample size. Surprisingly, this second effect only occurred for the smallest setting of n_0. A very encouraging, and also surprising, result was that the observed overall error remained well below the overall bound for all settings of $1/\delta$ and n_0. This last result suggests that the modeling assumptions, meaning the settings for *Bound*(δ, i, $|T|$), $Q(n, \alpha)$ and the entire range of values for n_0, hold for all tested settings of δ.

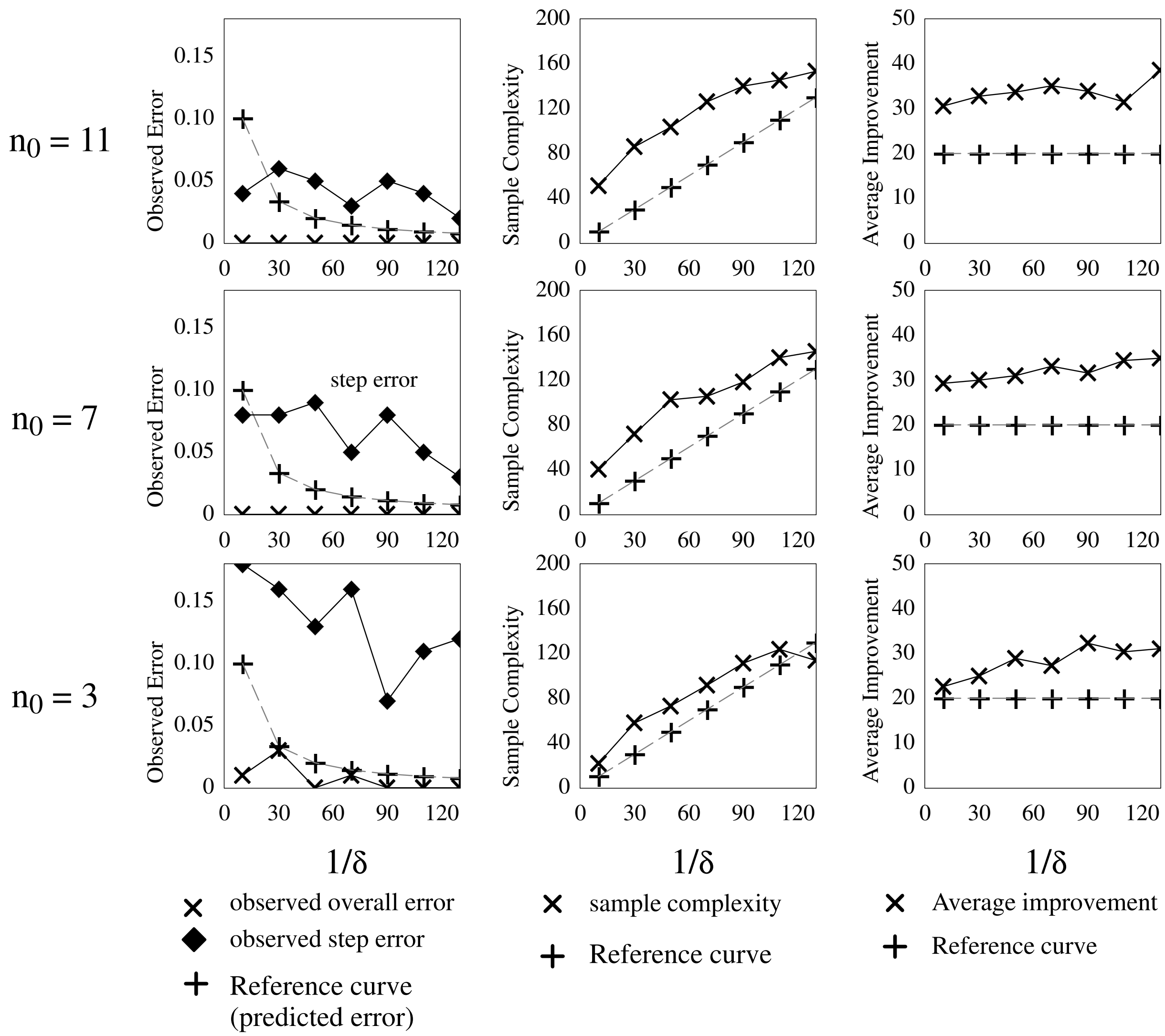


Figure 11: Empirical results of sensitivity analysis (continued). Results for $n_0 = 3$, 7, and 11.

Tracking observed overall error and observed step error separately allows us to tease apart how *Bound*(δ, *i*, |T|) and $Q(n, \alpha)$ contribute to the overall successful performance. Under the default settings, the step error, β_i is set to the total error, δ (Equation 8). The results show that the step error was not necessarily below the requested value of δ. The observed step error closely followed the bound for large values of n_0, but was considerably higher than the bound for low values of n_0. Comparing these results with the observation of low overall error suggests two conclusions First, the decision to bound the step error by δ (Equation 8) is probably too conservative – even when there is higher than expected step error the observed overall error remains low. Second, the performance of the normal approximation embedded in the default $Q(n, \alpha)$ (Equation 11) is sensitive to the initial sample size, and performs poorly for small values of n_0. The observation that the step error was not significantly below β_i, even for large n_0, suggests that the other assumption embodied in *Bound*(δ, i, |T|), Equation 5, is not overly conservative. The results also show an increasingly erratic behavior of the plots with smaller n_0 which can be traced to the higher variance that results from basing estimates on fewer initial examples.

Another interesting trend is the apparent interaction between n_0 and $1/\delta$. First, when we insisted on *high* confidence, a small initial sample size led to *more* than the predicted error. This should have been expected as a small sample size will result in $Q(n, \alpha)$ providing a coarse approximation of the true error, and insisting on high confidence requires that the algorithm make fine distinctions about statistical error. Second, when we ask for *low* confidence, a large initial sample size led to *less* than the predicted error. This can be explained by the observation that statistical error is dependent on the size of the final sample, and a large initial sample size forces COMPOSER to take more examples than might be necessary to achieve the required level of accuracy.

5.1.6 Sample Complexity Results

The second column of Figures 10 and 11 illustrates the average number of examples required to adopt a transformation. The results support the predictions of Theorem 2. Growth was considerably below the quadratic bound, in fact it was sub-linear. The growth seems larger than the log factor discussed in Theorem 2, however, that is an asymptotic result and does not necessarily hold for the relatively small values of $1/\delta$ used in the evaluation.

As predicted, the average number of examples also grew as we increased the initial sample size. The growth is difficult to see from the results as the graphs are in terms of $1/\delta$. Figure 12 displays the growth as a function of n_0. The figure shows that the sample complexity grew roughly linearly in n_0, confirming expectations.

5.1.7 Average Improvement Results

The underlying goal of COMPOSER is to produce problem solvers with high expected utility. Thus the most important consideration in the sensitivity analysis is how parameter settings effected this aspect of behavior. The results in terms of average improvement are very encouraging. We predicted a trend of decreased improvement, both as we decrease $1/\delta$ and n_0, however, the results indicate almost no effect from changing the parameters on the average improvement produced by COMPOSER (there was a slight trend consistent with the predictions).

5.2 ANALYSIS

The sensitivity analysis produced two very encouraging results. First was the relative invariance in the performance of customized problem solvers across the different parameter settings. Except for a noticeable degradation for the smallest initial sample size, the average expected utility of customized problem solvers was quite good. Second, the observed overall error was well below the requested value across the different parameter settings. Although these results are based a single application and some caution must be taken in extrapolating them, they do indicate that parameter tuning may not be a significant problem when constructing customized problem solvers. The results do indicate that the confidence parameter has a significant effect on the average sample size. But again these results were encouraging, showing that the actual growth was significantly less than the quadratic bound we provide in Theorem 2.

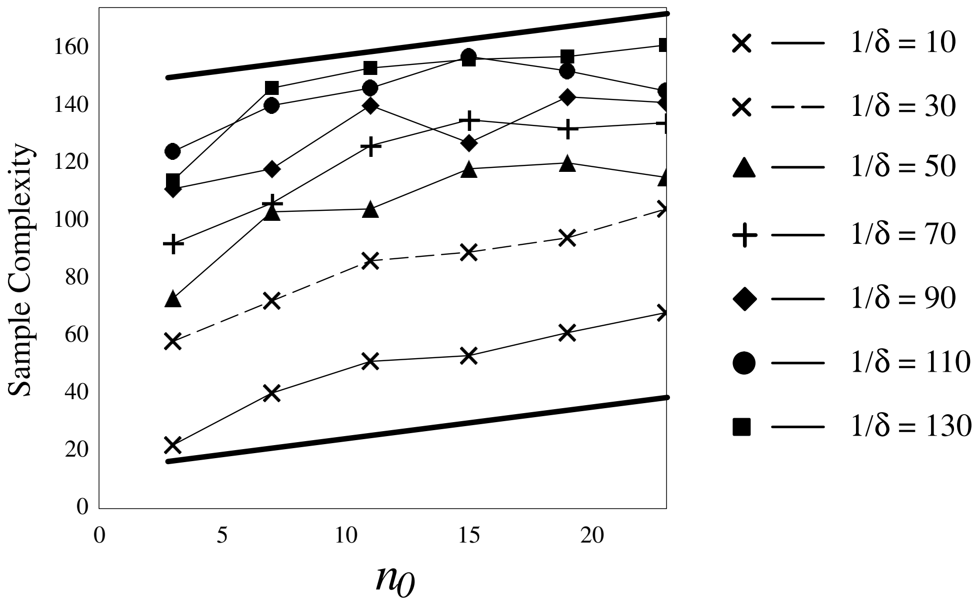


Figure 12: Sample complexity results as a function of n_0. The graph is bounded above and below by a line of unit slope, which is the predicted increase as a function of n_0.

The results highlight two areas of caution. First, the normal approximation significantly degraded when the initial sample size was small. Although this degradation was relatively graceful, the acceptable values for a given application will depend on properties of the original distribution, such as its asymmetry (see Appendix C for some observations on this effect). Second, the results indicated an interaction between n_0 and δ. These parameters overlap somewhat in function. Providing a high initial sample size resulted in higher observed confidence. However, this only appears an issue when we ask for a high initial sample size and low confidence. In practice, it is hardly a problem to receive more confidence than requested.

The results suggest some basic rules of thumb for applying composer to a particular application. Higher confidence requires a larger initial sample size. An rule we have derived from our experience is to set n_0 to one quarter of $1/\delta$. Larger initial sample sizes do not seem to significantly increase the sample complexity, therefore if one is concerned about statistical error, they should use larger initial sample sizes before increasing the confidence factor.

Chapter 6

Limitations and Future Work

COMPOSER provides a solution to the customization problem and has proven effective in both of two rigorous evaluations. These successes are encouraging, however it is important to realize that COMPOSER embodies many design commitments that restrict its generality. In this chapter we discuss these limitation and possible extensions to the approach. We first characterize some conditions on when COMPOSER will lead to successful results. We then consider limitations and extensions to three key aspects of the COMPOSER algorithm: organizing the search through the space of modifications, estimating utility of transformations, and gathering statistics.

6.1 APPLICABILITY CONDITIONS

COMPOSER embodies numerous restrictions in achieving the goal of efficient customization. As a result, the technique will not apply equally well to every situation. We can summarize four basic conditions that, if satisfied by an application, should lead to successful results.

1. Well Structured Transformation Space

A modified problem solver is constructed by composing some body of atomic modifications. In general, the space of possible composite modifications will be so large as to make exhaustive search intractable. COMPOSER requires a transformation generator that structures this space into a sequence search steps, with relatively few transformations at each step.

Clearly, COMPOSER's performance is tied to the transformations it is given. If COMPOSER is to be effective, there must exist good methods for the control points that make up a strategy. Because of the nature of hill-climbing, even if a good strategy exists, there is no guarantee that COMPOSER will find it. One may have to consider carefully how to organize the transformation space. When it is available, knowledge such as dominance and independence can help structure the transformation space, improving learning efficiency and mitigating the effects of local maxima.

2. Large Supply of Representative Training Problems

COMPOSER's statistical approach assumes that the pattern of tasks can be represented by a fixed problem distribution. To estimate this distribution, the algorithm must be provided with a relatively large body of training problems that are representative of this distribution.

An problem arises when the distribution cycles through a set of different patterns, or if the pattern shifts slowly over time. Shifts in the distribution, cyclic or otherwise, violate the fixed distribution assumption. However there are approaches for partially overcoming these difficulties. With cyclic patterns, it may suffice to average over the cycle. One can draw problems throughout the extent of the cycle and then randomly shuffle them. Customizing to this shuffled distribution will result in a problem solver that does not take advantage of the shifts, but is at least customized to do well on average. With slowly shifting distributions one can take windows of problems, train on then, and periodically re-customize the problem solver as the distribution shifts. Tracking and taking advantage of predicted shifts is an important area of future research.

3. Low Problem Solving Cost

By default, COMPOSER extracts incremental utility statistics by solving each training example using various transformations. This is only feasible if problems can be solved with a sufficiently low cost in resources. This is perhaps the strongest limitation of the technique. It is not feasible to use COMPOSER to customize, for example, an average-case exponential-time problem solver. The technique is quite appropriate, however, if the problem solver is initially tractable, but highly inefficient. Potential ways of relaxing this reliance on problem solving are discussed below.

4. Domain Regularity

The motivation behind customization is that there is application-specific information to exploit. There must be restrictions in the domain of discourse or regularity in the pattern of tasks, such that a customized problem solver can outperform a general purpose technique. If the application requires solving a sequence of provably intractable problems, there is probably no good custom technique.

6.2 ORGANIZING SEARCH

One of the key issues in customization is successfully navigating through the vast space of possible composite modifications. COMPOSER's strongest restriction, hill-climbing, is in service of the goal of efficient search. In many ways this restrictions is too strong. When there are strong interactions between transformations, COMPOSER may find poor local-maxima, or not find any solutions at all. In other ways the restriction is too weak. It says nothing about how many transformations will be considered at each step.

It is vitally important to focus the search to consider the most promising alternatives first. COMPOSER is inherently generate and test. Performance can be improved by making generation as intelligent as possible. One of the important distinctions between the PRODIGY implementation and the Deep Space Network implementation, is that while the former had a much larger space of possible modifications (the power set of all syntactically legal control rules), the PRODIGY implementation searched this space much more efficiently. This is because a powerful learning element was used to propose only a small set of possible transformations at each step. The explanation-based learning component carefully analyzes each problem solving trace for search inefficiencies, and only proposes transformations for observed deficiencies. Where applicable, heuristics like Etzioni's non-recursive hypothesis can help filter out unpromising transformations. We have not yet developed such powerful techniques for proposing transformations in the scheduling domain. COMPOSER/DSN investigates all legal transformations at a step, without limiting its active hypothesis based on the scheduler's performance. Improving this performance is an area of current research in the DSN application. We are currently investigating the use of inductive learning techniques to suggest a limited set of modifications based on observations of scheduling behavior over multiple problems.

The Deep Space Network implementation used dominance and independence relations between control points to help organize the search space, both to address the problem of local maxima and to increase search efficiency. Such relations need further study to assess their effectiveness. While we believe that such information was an important factor in COMPOSER's success in the DSN domain, we must perform further studies to assess its true impact. If it can be demonstrated that such information plays a crucial role, the next logical step is to consider techniques for automatically deducing these relations through an analysis of a problem solver's structure.

An interesting issue arises when transformations are continuous rather than discrete, for instance, when the problem solver has a real-valued parameter the influences its behavior. As an example, consider A* search where heuristic distance is a linear combination of two attributes. Changing the coefficients in the combination will change the heuristic measure, and presumably the behavior of the search. Such a learning problem is not well suited to COMPOSER's hill-climbing. To encode it, one would have to discretize the combination into several distinct values, and treat each as a distinct transformation. A better approach would be to attempt to approximate the functional relationship, and set the parameter accordingly. This suggest a much different control structure than simple hill-climbing search. Future work will consider such alternatives.

Finally, we are currently investigating how to apply notions from work on bounded rationality to help control the search [Doyle90, Horvitz89]. COMPOSER is directed towards identifying transformations with high incremental utility. The utility function determines the value of transformations, and thus the direction of this search. The efficiency of search is ensured by imposing some restrictive bias on how exploration proceeds. The actual search behavior of the algorithm arises from the interaction of the utility function and these biases. We would like to develop a more principled method for characterizing the value of investigating transformations which directly relates their incremental utility and the cost to achieve this improvement. For some preliminary results see [Gratch93a].

6.3 ESTIMATING INCREMENTAL UTILITY

The search through the transformation space relies on the ability to accurately estimate the incremental utility of alternative transformations. This is made difficult by the distributional nature of incremental utility. Typically the exact shape of the distribution, or even its general form are unavailable and must be estimated by applying training data to some statistical model. While the number of examples required to form these estimates is only log of the required confidence, reducing this cost is a significant practical concern. Another chief limitation is that the accuracy of estimates is ultimately tied to the appropriateness of the statistical model. In the case of COMPOSER, appropriateness is tied to an initial sample size parameter which may have to be adjusted for a particular domain. Reliance on such parameters is unfortunate, but the only obvious statistical alternative seems to be to use extremely weak statistical models, like chernoff bounds, which result in substantially higher sample complexities. An important area of future work is the investigation of alternative stopping rules which lie between the Nádas technique and chernoff bounds.

One inefficiency in how COMPOSER gathers statistics is that it treats each transformation as in independent entity, even though their there is often a relationship between transformations that would allow a more efficient use of information. As an extreme example, if two identical transformations are provided to COMPOSER, the algorithm will maintain twice the statistics necessary. Sometimes it may be possible to develop a single statistical model from which one can derive the incremental utility of multiple transformations. This is the approach taken by [Subramanian92] and [Laird92]. This approach is not always possible – our current experience in the Deep Space Network domain suggests that little information can be gained about transformations in this domain without trying them out – but it is an important area of future research.

Heuristics and prior information can be used to take the place of or bias statistical estimates. Syntactic measures like the *operationality criteria* of Mitchell, Keller, and Kedar-Cabelli [Mitchell86] can be seen as approximate binary measures of incremental utility. Unfortunately, syntactic measures have difficulty capturing the distributional nature of incremental utility. Instead, we would like to investigate the use of such measures as a bias on the estimation process. For example, COMPOSER could be modified to require less statistical confidence when transformations satisfy certain syntactic measures of utility, thereby allowing estimates to be based on fewer examples. In a related issue, we observed in the PRODIGY application that many of the same control rules considered in one hill-climbing step are also considered in subsequent hill-climbing steps. Currently COMPOSER discards all information across steps as each step is conditional on a different control strategy. However, information gained from a previous step is often useful, and could be used as a bias on future estimation.

6.4 GATHERING STATISTICS

The need for efficient search and estimation arises from the fact that it can be quite expensive to gather incremental utility statistics. The default method requires solving problems to obtain incremental utility values. COMPOSER is limited to cases where it is feasible to solve problems problems with the original and intermediate problem solvers. While this dependance on problem solving is a serious limitation, in some applications it can be overcome or mitigated. For example, in the PRODIGY domain we were able to take advantage of properties of the transformations to process examples more efficiently. This example is just one instance of a general observations that that some transformation vocabularies may be easier to implement within the COMPOSER framework than others. Perhaps the issue can be resolved by identifying alternative means to gather utility values.

An important area of future work is the possibility of basing incremental utility estimates on weaker and cheaper-to-obtain information. For example, Greiner and Jurisica [Greiner92a] propose one method for evaluating several transformations from a single solution attempt by maintaining upper and lower bounds on the utility of the novel search paths. Other authors have suggested that it may be possible to gain useful information about currently intractable problems by first learning from simpler problems (e.g. [Natarajan89]) or by observing a teacher (e.g. [Tadepalli91]).

6.5 SUMMARY

The notion of decision-theoretic customization provides a general framework for viewing techniques that modify problem solving behavior. However, to address pragmatic concerns like computational efficiency, an algorithm must adopt multiple commitments that limit its generality. COMPOSER can be viewed as one set of such simplifying commitments, and thus represents just one point in a large space of possible solutions to the customization problem.

Throughout this thesis we have viewed the overall customization problem as composed of essentially three basic and roughly independent problems. First, there is a problem of searching the space of possible composite modifications. Second, there is the issue of of obtaining estimates of local properties of transformations across the pattern of task, in our case estimating incremental utility. Finally, there is an issue of gathering information, such as incremental utility values, to perform estimation. COMPOSER can be seen as embodying commitments to arrive at reasonable solutions for each of these subproblems. By decomposing the problem in this way, it is possible to consider approaches like COMPOSER as not just a single algorithm, but as a collection of methods, each of which can be tested individually. It is our hope then that future research in this area will not simply result in the development of large techniques like COMPOSER, or PRODIGY, etc., but by the development of smaller, well understood, methods that may be combined to produce a complete customization algorithm. For a development of this notion, see [Gratch92b].

Chapter 7

Conclusion

"A strict accordance between practical work and a corresponding theory is never possible and yet all our life is based on constant practical applications of inapplicable theories."

– Jerzy Neyman

We conclude by first restating the main arguments of the thesis and then by summarizing the contributions of this work.

7.1 SUMMARY OF MAJOR POINTS

Chapter 1 reviewed how past and current approaches exploit domain constraints to improve problem solving behavior. Some techniques focus on the structure on the domain of discourse, while others utilize the pattern of tasks. Generally, approaches improve the computational efficiency of problem solving, although we illustrated how many other aspects of problem solving performance are important for specific applications. Many terms have been applied to the problem these adaptive approaches are intended to solve (e.g. speed–up learning, knowledge compilation, the utility problem), however these terms are often inconsistently applied and, we believe, do not capture the full generality of what these approaches attempt to accomplish. Our definition of the customization problem was motivated to provide a clear and general characterization of the goal of these varied approaches.

We introduced a characterization of an application in terms of three components: (1) an evaluation criterion, (2) a domain of discourse, and (3) a pattern of tasks. The evaluation criterion captures what aspects of problem solving behavior are important in a given application – some applications require solution generation efficiency, others require solution execution efficiency. The domain of discourse and pattern of tasks capture how the problem solver is to be used. We defined the customization problem as the problem of adapting a problem solver given this characterization of an application. The process of customization modifies a problem solver to maximize its evaluation over the application, subject to the domain of discourse and pattern of tasks.

Chapter 2 illustrated that by assuming the pattern of task is derived from a random selection from a fixed probability distribution, application's evaluation criterion can be represented as the expected utility of the problem solver over the pattern of tasks. COMPOSER uses this characterization to guide the customization process.

Customization is cast as a search through some space of possible modified problem solvers for one with high expected utility. Modifications are adopted only after they are statistically evaluated, to an arbitrary level of confidence, using training examples drawn according to the pattern of tasks.

COMPOSER can be combined with a problem solver to produce an adaptive problem solving system. This was illustrated both for PRODIGY, a general purpose planing system, and LR-26, a JPL scheduling system. These implementations served as the basis of two extensive evaluations of the composer technique. In the first evaluation the approach compared favorable to PRODIGY/EBL, a state-of-the-art learning technique for improving planning performance. In the second evaluation, COMPOSER improved the performance of a scheduler in the real-world application of satellite communication.

Chapter 6 highlighted that COMPOSER is in no sense the final solution to the customization problem. COMPOSER embodies many assumptions and commitments that restrict its power and generality. As such, it represents just one point in a vast space of possible solutions. One might suspect there is no final solution to the customization problem; any technique will possess some restrictions. In any event, our hope is that COMPOSER will prove a useful approach for developing effective adaptive problem solvers.

7.2 CONTRIBUTIONS

This thesis has argued it is desirable, and possible, to construct general problem solving techniques that automatically adapt to the characteristics of a specific application. Adaptive problem solving is a means of reconciling two seemingly contradictory needs. On the one hand, general purpose techniques can ease much of the burden of developing a application and satisfy the oft argued need for declarative and modular knowledge representation. On the other hand, general purpose approaches are ill-suited to the specialized demands of individual applications. General approaches have proven successful, only after a tedious cycle of manual experimentation and modification. Adaptive techniques promise to reduce the burden of this modification process and, thereby, take a long step toward reconciling the conflicting needs of generality and efficiency.

Ours is not the first work to advocate adaptive problem solving. This thesis builds upon a considerable tradition in the machine learning community. However, it can be safely argued that the science of adaptive problem solving is still very much in its infancy. Techniques are heuristic, evaluation empirical, and even the basic goals are subject to frequent debate. We have attempted, in small measure, to improve on the situation, to clarify the goals as they are currently understood, to provide a formal framework for discussing the issues, and to introduce a rigorous learning approach.

This thesis develops a formal characterization of the customization problem, tailoring a problem solver to a specific application. Our definition of the customization problem connects work on adaptive problem solving to the rich field of decision theory. This has been a fertile connection, giving rise to COMPOSER. COMPOSER is a statistically rigorous algorithm built upon the decision-theoretic foundation. It transforms a general problem solving technique into one specialized to a specific application. COMPOSER is still a heuristic approach, but by casting it within a statistically sound framework we are able to clearly articulate the assumptions which underly the technique and predict their consequences. Most importantly, since these assumptions are stated explicitly, they can be subjected to empirical investigations. We have done so extensively throughout this thesis.

Appendix A
Domain Specifications

A.1 STRIPS DOMAIN

The STRIPS domain is simple domain from [Minton88b] where it is referred to as the extended STRIPS robot domain. The domain consists of interconnected rooms containing a robot, boxes and keys. Rooms are connected by doors which may be open, closed, or closed and locked. Boxes may be pushable or carriable or neither. Objects may be next to each other. Goals consist of achieving some conjunctive state in this world.

A.1.1 Domain Theory

```
(PICKUP–OBJ (<o1>)
 (preconds (and        (arm–empty)
                       (next–to robot <o1>)
                       (carriable <o1>)))
 (effects (  (del (arm–empty))
             (del (next–to <o1> <*30>))
             (del (next–to <*31> <o1>))
             (add (holding <o1>)))))


(PUTDOWN (<o2>)
 (preconds (holding <o2>))
 (effects (    (del (holding <*35>))
               (add (next–to robot <o2>))
               (add (arm–empty)))))


(PUTDOWN–NEXT–TO (<o3> <other–ob> <o3–rm>)
 (preconds (and        (holding <o3>)
                       (is–object <other–ob>)
                       (inroom <other–ob> <o3–rm>)
                       (inroom <o3> <o3–rm>)
                       (next–to robot <other–ob>)))
 (effects (  (del (holding <*35>))
             (add (next–to <o3> <other–ob>))
             (add (next–to robot <o3>))
             (add (next–to <other–ob> <o3>))
             (add (arm–empty)))))
```

```
(PUSH–TO–DR (<b1> <d1> <r1>)
 (preconds (and        (is–door <d1>)
                       (dr–to–rm <d1> <r1>)
                       (inroom <b1> <r1>)
                       (next–to robot <b1>)
                       (pushable <b1>)))
 (effects (  (del (next–to robot <*3>))
             (del (next–to <b1> <5>))
             (del (next–to <*13> <b1>))
             (add (next–to <b1> <d1>))
             (add (next–to robot <b1>)))))


(PUSH–THRU–DR (<b–x> <d–x> <r–x> <r–y>)
 (preconds (and        (is–room <r–x>)
                       (dr–to–rm <d–x> <r–x>)
                       (is–door <d–x>)
                       (dr–open <d–x>)
                       (next–to <b–x> <d–x>)
                       (next–to robot <b–x>)
                       (pushable <b–x>)
                       (connects <d–x> <r–x> <r–y>)
                       (inroom <b–x> <r–y>)))
 (effects (  (del (next–to robot <*1>))
             (del (next–to <b–x> <*12>))
              (del (next–to <*7> <b–x>))
             (del (inroom robot <*21>))
             (del (inroom <b–x> <*22>))
             (add (inroom robot <r–x>))
             (add (inroom <b–x> <r–x>))
             (add (next–to robot <b–x>)))))
```

```
(GO–THRU–DR (<ddx> <rrx> <rry>)
 (preconds (and        (arm–empty)
                       (is–room <rrx>)
                       (dr–to–rm <ddx> <rrx>)
                       (is–door <ddx>)
                       (dr–open <ddx>)
                       (next–to robot <ddx>)
                       (connects <ddx> <rrx> <rry>)
                       (inroom robot <rry>)))
 (effects (  (del (next–to robot <*19>))
             (del (inroom robot <*20>))
             (add (inroom robot <rrx>)))))

(CARRY–THRU–DR (<b–zz> <d–zz> <r–zz> <r–ww>)
 (preconds (and        (is–room <r–zz>)
                       (dr–to–rm <d–zz> <r–zz>)
                       (is–door <d–zz>)
                       (dr–open <d–zz>)
                       (is–object <b–zz>)
                       (holding <b–zz>)
                       (connects <d–zz> <r–zz> <r–ww>)
                       (inroom <b–zz> <r–ww>)
                       (inroom robot <r–ww>)
                       (next–to robot <d–zz>)))
 (effects(   (del (next–to robot <*48>))
             (del (inroom robot <*41>))
             (del (inroom <b–zz> <*42>))
             (add (inroom robot <r–zz>))
             (add (inroom <b–zz> <r–zz>)))))

(PUSH–BOX (<ba> <bb> <ra>)
 (preconds (and        (is–object <ba>)
                        (is–object <bb>)
                       (inroom <bb> <ra>)
                       (inroom <ba> <ra>)
                       (pushable <ba>)
                       (next–to robot <ba>)))
 (effects (  (del (next–to robot <*14>))
             (del (next–to <ba> <*5>))
             (del (next–to <*6> <ba>))
             (add (next–to robot <ba>))
             (add (next–to robot <bb>))
             (add (next–to <ba> <bb>))
             (add (next–to <bb> <ba>)))))

(GOTO–OBJ (<b> <rm>)
 (preconds (and        (is–object <b>)
                       (inroom <b> <rm>)
                       (inroom robot <rm>)))
 (effects (  (add (next–to robot <b>))
             (del (next–to robot <*109>)))))

(GOTO–DR (<dx> <rx>)
 (preconds (and        (is–door <dx>)
                       (dr–to–rm <dx> <rx>)
                       (inroom robot <rx>)))
 (effects (  (del (next–to robot <*18>))
             (add (next–to robot <dx>)))))

(OPEN (<door>)
 (preconds (and        (is–door <door>)
                       (unlocked <door>)
                       (next–to robot <door>)
                       (dr–closed <door>)))
 (effects (  (del (dr–closed <door>))
             (add (dr–open <door>)))))

(CLOSE (<door1>)
 (preconds (and        (is–door <door1>)
                       (next–to robot <door1>)
                       (dr–open <door1>)))
 (effects (  (del (dr–open <door1>))
             (add (dr–closed <door1>)))))

(LOCK (<door2> <k1> <rm–b>)
 (preconds (and        (is–door <door2>)
                       (is–key <door2> <k1>)
                       (holding <k1>)
                       (dr–to–rm <door2> <rm–b>)
                       (inroom <k1> <rm–b>)
                       (next–to robot <door2>)
                       (dr–closed <door2>)
                       (unlocked <door2>)))
 (effects (  (del (unlocked <door2>))
             (add (locked <door2>)))))

(UNLOCK (<door3> <k2> <rm–a>)
 (preconds (and        (is–door <door3>)
                       (is–key <door3> <k2>)
                       (holding <k2>)
                       (dr–to–rm <door3> <rm–a>)
                       (inroom <k2> <rm–a>)
                       (inroom robot <rm–a>)
                       (next–to robot <door3>)
                       (locked <door3>)))
 (effects (  (del (locked <door3>))
             (add (unlocked <door3>)))))
```

A.1.2 Initial Control Rules for STRIPS Domain

As in [Minton88b], in this domain, PRODIGY is provided with an initial set of search control rules. This set consists of the following rules:

```
(ONLY–PUSH–BOX
        (lhs (and  (current–node <node>)
                   (current–goal <node> (next–to robot <x>))
                   (candidate–op <node> PUSH–BOX)))
          (rhs (reject operator PUSH–BOX)))

(ONLY–CARRY–BOX–THRU–DOOR
        (lhs (and  (current–node <node>)
                   (current–goal <node> (inroom robot <x>))
                   (candidate–op <node> CARRY–THRU–DR)))
        (rhs (reject operator CARRY–THRU–DR)))

(ONLY–PUSH–BOX–THRU–DOOR1
        (lhs (and  (current–node <node>)
                   (current–goal <node> (inroom robot <x>))
                   (candidate–op <node> PUSH–THRU–DR)))
        (rhs (reject operator PUSH–THRU–DR)))

(ONLY–PUSH–BOX–THRU–DOOR2
        (lhs (and (current–node <node>)
                   (current–goal <node> (next–to <y> <x>))
                   (candidate–op <node> PUSH–THRU–DR)))
        (rhs (reject operator PUSH–THRU–DR)))

(ONLY–PUTDOWN–NEXT–TO–1
        (lhs (and  (current–node <node>)
                   (current–goal <node> (arm–empty))
                   (candidate–op <node> PUTDOWN–NEXT–TO)))
        (rhs (reject operator PUTDOWN–NEXT–TO)))

(ONLY–PUTDOWN–NEXT–TO–2
        (lhs (and  (current–node <node>)
                   (current–goal <node> (inroom <xx> <yy>))
                   (candidate–op <node> PUTDOWN–NEXT–TO)))
        (rhs (reject operator PUTDOWN–NEXT–TO)))
```

A.2 AB–WORLD DOMAIN

This domain was constructed by Oren Etzioni [Etzioni90a]. It is a slight variant of Minton's Blocksworld domain. A single macro–operator is added that allows PRODIGY to grasp a block that is second from the top of a tower.

```
(GRAB–FROM–BLOCK (params (<u> <t> <r>))
 (preconds (and       (object <u>)
                      (object <t>)
                      (object <r>)
                      (clear <t>)
                      (on <t> <u>)
                      (on <u> <r>)
                      (arm–empty)))
 (effects (  (del (on <t> <u>))
             (del (on <u> <r>))
             (del (arm–empty))
             (add (on <t> <r>))
             (add (holding <u>)))))
```

```
(PICK–UP (params (<ob1>))
 (preconds (and       (object <ob1>)
                      (clear <ob1>)
                      (on–table <ob1>)
                 (arm–empty)))
 (effects (  (del (on–table <ob1>))
             (del (clear <ob1>))
             (del (arm–empty))
             (add (holding <ob1>)))))
```

```
(PUT–DOWN (params (<ob>))
 (preconds (and       (object <ob>)
                      (holding <ob>)))
 (effects (  (del (holding <ob>))
             (add (clear <ob>))
             (add (arm–empty))
             (add (on–table <ob>)))))
```

```
(STACK (params (<sob> <sunderob>))
 (preconds (and       (object <sob>)
                      (object <sunderob>)
                      (clear <sunderob>)
                      (holding <sob>)))
 (effects (  (del (holding <sob>))
             (del (clear <sunderob>))
             (add (arm–empty))
             (add (clear <sob>))
             (add (on <sob> <sunderob>)))))
```

```
(UNSTACK (params (<us–ob> <underob>))
 (preconds (and       (object <us–ob>)
                      (object <underob>)
                      (on <us–ob> <underob>)
                      (clear <us–ob>)
                      (arm–empty)))
 (effects (  (del (on <us–ob> <underob>))
             (del (clear <us–ob>))
             (del (arm–empty))
             (add (holding <us–ob>))
             (add (clear <underob>)))))
```

A.3 BIN-WORLD DOMAIN

This domain was constructed by us in [Gratch91b] to highlight deficiencies in PRODIGY/EBL's utility analysis and Etzioni's non-recursive hypothesis. The domain is for a robot assembly task where the goal is to construct a composite part from a set of components. All the components for a particular part are stored in a bin. If all the components in the bin are free of defects, the part may be assembled. Otherwise another bin must be searched. The INSPECT–BIN operator determines if a given bin is suitable for assembly. The ASSEMBLE–COMPONENTS operator constructs the part.

A.3.1 Domain Theory

```
(ASSEMBLE–COMPONENTS (<BIN>)
  (preconds (exists (<BIN>) (parts–bin <BIN>) (defect–free–components <BIN>)))
  (effects ((add (TOP)))))

(INSPECT–BIN (<BIN>)
  (preconds (forall (<MEMBER>) (in–bin <MEMBER> <BIN>)
                    (GOOD <BIN> <MEMBER>)))
  (effects ((add (defect–free–components <BIN>)))))
```

A.3.2 Problem Distribution

A problem distribution is defined by enumerating a set of problem classes and assigning probabilities to each class. A set of problems is created by randomly constructing problems according to the distribution. The experiment is based on a uniform distribution over two problem classes. This means that each class has an equal chance of participating in a problem solving attempt. The first class contains problems with fifty bins of two components each. Forty-nine bins contain a defective component. One bin contains no defects. The bins are ordered such that by default the defect-free bin is checked last. The second class contains problems with two bins of two hundred components each. One bin is defect–free. The other bin contains a defective component and the components are ordered such that by default the defective component is observed last. Again, the bins are ordered such that by default the defect–free bin is checked last.

The rational behind this problem distribution is to construct a distribution with high variance. PRODIGY/EBL bases its utility estimates on a single example. Using a high variance ensures that estimating from a single example will result in an inaccurate representation of the true incremental utility of any learned control rule.

A.4 DEEP SPACE NETWORK SCHEDULING

To evaluate LR-26 we constructed a large set of scheduling problem instances. We obtained project requirements and time period information for the following eight projects:

26-METER SUB-NET MAINTENANCE (DSS) – All three antennas receive regularly scheduled maintenance periods. Projects cannot access the antennas during this period.

ASTRO-D (ASTD) – An x-ray astronomy satellite. It is a scientific Earth orbiting satellite of the Institute of Space and Astronautical Science of the Ministry of Education, Science, and Culture of Japan. Its mission is to provide powerful capabilities for the study of high energy astrophysical phenomena. Launch date: February 1, 1993. Projected DSN support: 2 years prime, 2 years extended.

INTERNATIONAL SOLAR TERRESTRIAL PHYSICS GEOTAIL MISSION (GTL) – Geotail will study the solar wind by observing the plasma sphere, plasma sheet, and ring currents in the Earth’s Geomagnetic tail region. Geotail data should help to provide a more complete understanding of how the solar processes affect the Earth’s environment. Launch date: July 14, 1992. Projected DSN support: 3 years.

NIMBUS-7 (NIM7) – The Nimbus program provides for the research and development of a series of large automated spacecraft for the flight test of advanced instruments for pollution, oceanographic, and meteorological applications. Launch date: October 24, 1978. Projected DSN support: 15 years plus 11 years extended mission.

SAMPEX (SMPX) – GSFC developed the SMEX program to provide frequent flight opportunities for highly focused and inexpensive space science missions. SAMPEX is the first mission of the SMEX program. Its primary scientific objectives are to measure the elemental and isotopic composition of solar energetic particles, anomalous cosmic rays, and galactic cosmic rays over the energy range from approximately 1 to several hundred MeV per nucleon. Launch date: June 1992. Projected DSN support: 3 years plus 3 years extended mission.

SOLAR-A (SOLA) – The YOHKOH spacecraft mission objectives are to investigate high energy phenomena of the sun using X-ray telescopes and spectrometers during the maximum activity period of the solar cycle. Experiments are being supported by various universities and laboratories in Japan, England, and the United States. Launch date: August 29, 1991. Projected DSN support: 3 years plus 2 years extended mission.

SUMBILLIMETER WAVE ASTRONOMY SATELLITE (SWAS) – A follow-on mission designed to study molecular clouds in the galactic plane, providing a mini and full survey of the clouds, leading towards the development of maps. Launch date: June 1995. Projected DSN support: 3 years minimum.

TOTAL OZONE MAPPING SPECTROMETER – EARTH PROBE (TOMS) – The primary science objectives of the TOMS-EP mission are to supplement and continue the ongoing measurements of Earth's atmospheric ozone currently being performed by USSR-US Meteor-3/TOMS cooperative mission and the Nimbus-7 mission. Launch date: December 1993. Projected DSN support: 3 years plus 2 years extended mission.

A weekly schedule is defined by a set of projects, and for each project, its requirements and a set of communication time periods. Project requirements were obtained directly from the project managers and from the Deep Space Network Mission Support Requirements (JPL D-0787, Rev. AI). Communication time periods were obtained through the Operations Mission Planner (OMP-26). OMP-26 is a scheduling tool being developed at JPL (see JPL D-10092). OMP-26 takes the raw time period data maintained by Network Operations Control Center and produces the ".bat" files used by the LR-26 scheduler. We derived time period data from the following six weeks (designated wk/yr): 17/92, 18/92, 32/92, 24/95, 24/96, 24/97. These were the only time period files available at the time of the experiments.

Random scheduling problems are constructed by systematically choosing subsets of seven projects from the list of eight, and for each project, choosing a set of time periods from the one of the six time period data files. This procedure generates a large body of possible scheduling problems. For an experimental run, all generated problems are randomly sorted and then drawn in order without replacement.

Appendix B

Theorems

Lemma 1: Under the default specification (Equation 11), $Q(n, \alpha)$ is the $\alpha/2$th quantile of the standard normal distribution:

$$Q(n,\alpha) = x \;\; such\ that \;\; \int_{x}^{\infty} \left(1/\sqrt{2\pi}\right) e^{-0.5y^2} dy = \frac{\alpha}{2}$$

$Q(n, \alpha)$ is bounded by a linear function in $1/\alpha$.

Proof:

Let $Q(n, \alpha) = q$, Let $\Phi(x) = \left(1/\sqrt{2\pi}\right) e^{-0.5x^2}$

Let Z be a standard normal random variable, meaning the distribution of Z is $\Phi(z)$.

We first re-derive the result known as Markov's inequality

$$E\left[|Z|\right] = \int_{-\infty}^{\infty} |Z|\Phi(z)dz \geq \int_{|Z| \geq q} |Z|\Phi(z)dz \geq q \int_{|Z| \geq q} \Phi(z)dz = q \times P\left[|Z| \geq q\right]$$

$$\text{Therefore} \quad P\left[|Z| \geq q\right] \leq \frac{1}{q} E\left[|Z|\right] \qquad \text{(Markov's inequality)}$$

The area under $\Phi(z)$ to the right of q is by definition $\alpha/2$. Similarly, as $\Phi(z)$ is a symmetric function, the area to the left of $-q$ is $\alpha/2$. Thus the probability that Z is greater than q or less than $-q$ is α. This is simply the shaded area in Figure 13.

Namely: $P\left[|Z| \geq q\right] = \alpha$

From Markov's inequality

$$P\left[|Z| \geq q\right] \leq \frac{1}{q} E\left[|Z|\right]$$

So $\alpha \leq \frac{1}{q} E\left[|Z|\right]$

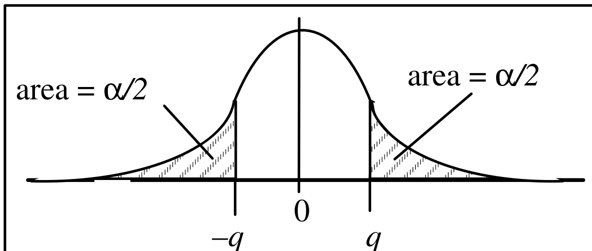


Figure 13: Distribution of Z

It can be shown $E\left[|Z|\right]$ is finite. Specifically $E\left[|Z|\right] = \sqrt{\frac{2}{\pi}}$

Therefore $q \leq \frac{E\left[|Z|\right]}{\alpha} = \sqrt{\frac{2}{\pi}} \cdot \frac{1}{\alpha}$

q.e.d.

Observation 1: Under the default specification (Equation 11), $Q(n, \alpha)$ is the ath quantile of the standard normal distribution:

$$Q(n,\alpha) = x \quad \text{where} \quad \int_x^\infty \frac{1}{\sqrt{2\pi}} e^{-\frac{1}{2}y^2} dy = \frac{\alpha}{2}$$

Asymptotically, $Q(n, \alpha)$ is of order root of log of $1/\alpha$:

Let $Q(n, \alpha) = q$, so $\frac{\alpha}{2} = \int_q^\infty \frac{1}{\sqrt{2\pi}} e^{-\frac{1}{2}y^2} dy$

When q is large, the integral can be approximated with its asymptotic expansion. We use Equation 7.1.23 in the Handbook of Mathematical Functions [Abramowitz65 p. 298].

$$\frac{\alpha}{2} \backsim \frac{e^{-q^2}}{q\sqrt{\pi}}\left[1 + \sum_{m=1}^{\infty}(-1)^m \frac{1 \cdot 3 \ldots (2m-1)}{(2q^2)^m}\right]$$

As q becomes large, the summation term rapidly tends to zero. Therefore we can approximate the relationship with the simpler expression:

$$\frac{\alpha}{2} = \frac{e^{-q^2}}{q\sqrt{\pi}}$$

And

$$\frac{1}{\alpha} = \frac{1}{2} e^{q^2} q \sqrt{\pi}$$

And

$$\ln(1/\alpha) = q^2 + \ln q - \ln 2$$

As q becomes large, the q^2 term rapidly dominates. Therefore we get the result:

$$q \approx O\left(\sqrt{\ln(1/\alpha)}\right)$$

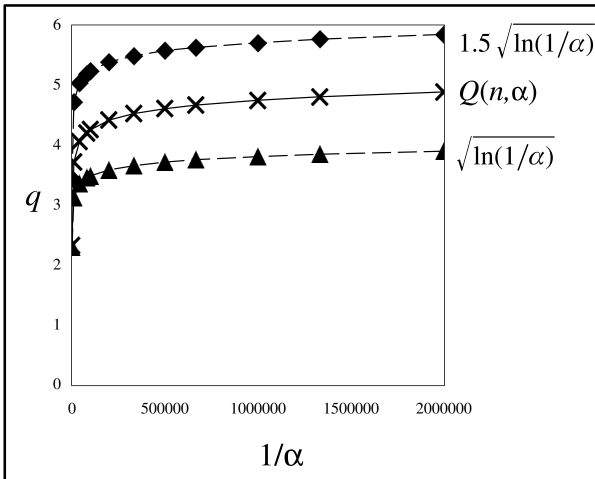


Figure 14: Comparison $Q(n,\alpha)$ to the approximation $\sqrt{\ln(1/\alpha)}$

Numeric simulations show that this is a reasonable approximation (see Figure 14). This means that the bound in Lemma 1 is too conservative. The following theorems use the bound in Lemma 1 to bound the computational complexity of the approach. However, the root-log approximation provides a more reasonable characterization in practice.

Theorem 1: Let ST be the stopping time associated with the Nádas stopping rule for a given transformation. The expected stopping time is governed by the following relationship:

$$E[ST] \approx Q(n,\alpha)^2 \frac{\sigma^2}{\mu^2}$$

meaning the stopping time is approximately equal to $Q(n,\alpha)$ times the actual variance of the distribution associated with the transformation, divided by the square of the actual incremental utility of the transformation, where σ^2 is the actual variance and μ is the actual incremental utility.

Proof:

Theorems 11, 12, and 13 of [Nádas69] state that:

$$E[ST] \approx \min_{n \geq 1}\left(n : \frac{\sigma^2}{\mu^2} \leq \frac{n}{Q(n,\alpha)^2} \right)$$

By definition, σ^2, μ, and $Q(n, \alpha)^2$ are always positive and $ST \geq n_0$. Setting the inequality to an equality and solving for n we get the result.

Corollary: The stopping time associated with a transformation grows linearly in σ^2 and quadratically in the absolute value of $1/\mu$. If the default definition of $Q(n, \alpha)$ is used (Equation 11), it grows no more than quadratically in $1/\alpha$ (from Lemma 1).

Theorem 2: Let ST^* be the stopping time associated with a step in COMPOSER's hill-climbing search. Use the default value for $Bound(\delta, i, |T|) = \delta/|T|$ (Equations 5 and 8), where δ is the error level and T is the set of transformations at that step. Then $E[ST^*]$ grows quadratically in $1/\delta$ and quadratically in $|T|$:

$$E[ST^*] \leq c\left(\frac{|T|}{\delta}\right)^2$$

Ignoring differences in the means and variances in the elements of T as it changes.

Proof:

This follows directly from Lemma 1 and Theorem 1. This states that as we increase the size of the candidate set at a step, or as we increase $1/\delta$, we should expect a quadratic increase in the stopping time at a step.

Remark. Observation 1 indicates this bound is too conservative. In practice the growth is closer to the log of $|T|/\delta$. The constant c is a factor related to the mean and variance of the last transformation adopted at the step. If there are transformations with positive μ in T (positive incremental utility), we expect the one with minimum σ^2/μ^2 to be adopted and end the step. If no transformation in T has positive μ, the step will continue until all elements of T are rejected. In this case c is the maximum σ^2/μ^2 from T.

Theorem 3: Let R be a bound on the cost of solving a problem. Using the default settings for COMPOSER, the expected time required to perform one step in the hill-climbing search is cubic in the number of transformations and quadratic in the required confidence, and linear in R:

$$\textit{Cost of a step is} \quad O\left(R\frac{|T|^3}{\delta^2}\right)$$

Proof:

From Theorem 2 the number of samples required at a step is $O\left(\frac{|T|^2}{\delta^2}\right)$

Using the default means for gathering incremental utility statistics requires solving each sample $|T| + 1$ times. Each time can cost at most R. *q.e.d.*

Remark: Again, this bound is too conservative. Using Observation 1, the cost observed in practice should be approximately:

$$O\left[R \cdot |T| \cdot \ln\left(\frac{|T|}{\delta}\right)\right]$$

Appendix C
COMPOSER Properties

This appendix relates some miscellaneous observations relevant to the performance of COMPOSER.

C.1 PROPERTIES OF *Bound*(δ, i, |T|)

The *Bound* function relates the bound on the probability of adopting a customized problem solver with worse expected utility to a bound on the accuracy of incremental utility estimates. By default, the allowed statistical error at a step, β_i, is set to the overall acceptable error of δ. This embodies a policy that taking several steps forward is worth an occasional step backwards, mean that across a transformation sequence there may be more than probability δ transformations with negative incremental utility. However, we investigated two alternative bounding functions, reproduced here:

$$GLOBAL-BOUND/LIMITED-STEPS : \beta_i := \delta/k \tag{8}$$

$$GLOBAL-BOUND/UNLIMITED-STEPS : \beta_i := \frac{6\delta}{i^2\pi^2} \tag{9}$$

These functions act as conservative bounds, ensuring that the total probability of *any* transformation in the adopted sequence having negative incremental utility remains below δ. In the case were we know there will be exactly *k* steps the hill-climbing search can take, this is accomplished simply by setting the error allowed at a step to δ/k. In the case where the number of steps cannot be bounded, we need an infinite sequence of values such that the sum of any finite set of values remains below δ, a property satisfied by Equation 9. Equation 9 has two drawbacks. First, since the number of steps will always be a finite number, the sum of β_i's will always be less than δ. Thus Equation 9 "wastes" some probability. Second, the number of examples required for each successive transformation increases as each successive β_i decreases. This punishes later steps.

What we relate here is the surprising observation that, if many steps are taken, the total number of examples required by the two approaches differs by a factor of two.

Let *n* be the number of steps taken in the search. From Observation 1 we know that the stopping times associated with a transformation grow roughly with the log of the confidence. We also know that it grows linearly with the variance and quadratically in one over the true incremental utility of the transformation. Assume that the number of transformations at each step is constant and therefore factored out. The total number of examples, *N*, is the sum of the examples taken at each step. Each equation leads to a different number. Call these N_{Eq8} and N_{Eq9} for Equations 8 and 9 respectively:

$$N_{Eq8} = \sum_{i=1}^{n} \frac{S_i^2}{2U_i^2} \ln\frac{n}{\delta} \qquad\qquad N_{Eq9} = \sum_{i=1}^{n} \frac{S_i^2}{2U_i^2} \ln\left(\frac{i^2\pi^2}{6\delta}\right)$$

To compare these two techniques we must make some assumptions. One is to assume the variance is equivalent for each transformation, $\forall i : S^{2^i} = S^2$. A more difficult issue is how to model the incremental utility at each step, U_i. This is the utility of the transformation which will be adopted at each stage. We expect the transformation with the highest utility to be adopted first so U_i can reasonably be considered the transformation with highest utility at the ith stage. This value will tend to change across different transformations however. Thus we might imagine a function $U(i)$ i=1..n which states how the average utility of transformations changes across the transformation sequence to be adopted. For lack of a better name we will call this an *acceleration function.* Experience with composer suggests that this function is monotonically decreasing as i increases. However, for our first cut at analysis we will assume that $U(i)$ equals some constant c. We compare the ratio of N_{Eq9} to N_{Eq8}.

$$N_{Eq8} = \sum_{i=1}^{n} \frac{S^2}{2U(i)^2} \ln \frac{n}{\delta} \qquad N_{Eq9} = \sum_{i=1}^{n} \frac{S^2}{2U(i)^2} \ln\left(\frac{i^2\pi^2}{6\delta}\right)$$

$$N_{Eq8} = n\frac{S^2}{2c^2}\left[\ln n - \ln \delta\right] \qquad N_{Eq9} = \sum_{i=1}^{n} \frac{S^2}{2c^2}\left[2\ln \pi - \ln 6 - \ln \delta + 2\ln i\right]$$

$$N_{Eq9} = \frac{S^2}{2c^2}\left[2n\ln \pi - n\ln 6 - n\ln \delta + 2\ln n!\right]$$

$$\frac{N_{Eq9}}{N_{Eq8}} = \frac{\frac{S^2}{2c^2}\left[2n\ln \pi - n\ln 6 - n\ln \delta + 2\ln n!\right]}{\frac{S^2}{2c^2}\left[n\ln n - n\ln \delta\right]}$$

and

$$\frac{N_{Eq9}}{N_{Eq8}} = \frac{2n\ln \pi - n\ln 6 - n\ln \delta + 2\ln n!}{n\ln n - n\ln \delta}$$

let $c_1 = 2\ln\pi - \ln 6 = 0.4977$, $c_2 = -\ln\delta$

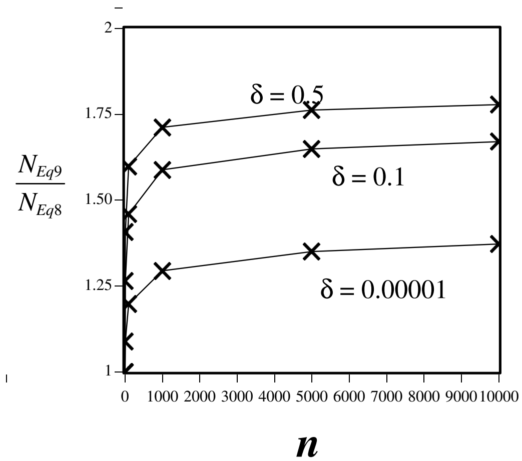
$$\frac{N_{Eq9}}{N_{Eq8}} = \frac{n(c_1 + c_2) + 2\ln n!}{n(c_2 + \ln n)}$$

For large values of n, $n!$ can be approximated by $\sqrt{2\pi n}\left(\frac{n}{e}\right)^n$

This is called *Stirling's approximation*. Using this approximation:

$$\frac{N_{Eq9}}{N_{Eq8}} = \frac{n(c_1 + c_2) + \ln 2\pi + \ln n - n + 2n\ln n}{nc_2 + n\ln n}$$

The ratio is a function of two variables, n and δ. If we consider the partial derivative of this ratio with respect to n (hold δ constant and vary n) we see that the ratio is dominated by the $n \ln n$ factors and in the limit the ratio converges to two. If we look at the partial derivative with respect to δ, the ratio c_3/c_2 dominates and converges quickly to one as δ tends toward zero. Below the ratio is plotted for some values of n and δ:



The preceding analysis indicates that, for large *n*, Equation 9 is worse than Equation 8 by a factor of two if we assume a uniform utility for each transformation in the sequence. Through simulations we have observed that N_{Eq9} can actually be smaller than N_{Eq8} if the average incremental utility tends to rise increase as hill-climbing proceeds. However, as we have noted in our two COMPOSER applications, the acceleration function tends to decrease as transformations are adopted. A reasonable model of our data uses an acceleration function of the form $U(i) = C/i$.

For either equation, $N = \sum_{i=1}^{n} \frac{S^2}{2U(i)^2} \ln \frac{1}{\beta_i}$ so $N = \sum_{i=1}^{n} \frac{S^2 i^2}{\left(\sqrt{2}\,C\right)^2} \ln \frac{1}{\beta_i}$

Letting $K = \frac{S^2}{\left(\sqrt{2}\,C\right)^2}$ we get $N = K \sum_{i=1}^{n} i^2 \ln \frac{1}{\beta_i}$

Substituting in Equations 8 and 9:

$$N_{Eq8} = K\left[c_3 \sum_{i=1}^{n} i^2 + 2 \sum_{i=1}^{n} i^2 \ln i \right] \qquad N_{Eq9} = K \sum_{i=1}^{n} i^2 [2 \ln \pi - \ln 6 - \ln \delta + 2 \ln i]$$

Let $c_1 = 2\ln\pi - \ln 6 = 0.4977$, $c_2 = -\ln\delta$, $c_3 = c_1 + c_2$ and note $\sum_{i=1}^{n} i^2 = \frac{1}{3}n^3 + \frac{1}{3}n^2 + \frac{1}{6}n = Q(n)$

so

$$\frac{N_{Eq9}}{N_{Eq8}} = \frac{Q(n)c_3 + 2\sum_{i=1}^{n} i^2 \ln i}{Q(n)c_2 + Q(n)\ln n}$$

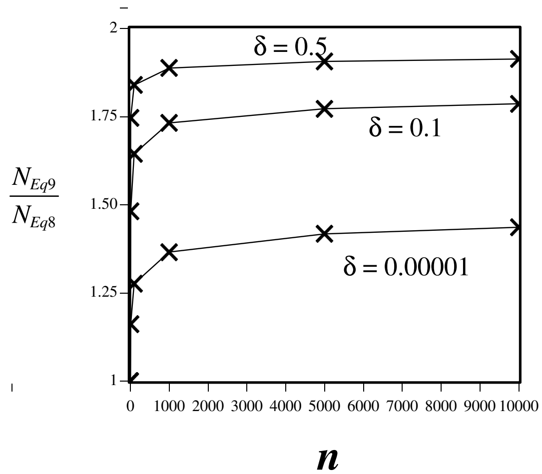
For large *n* we can approximate the summation with an integral, so

$$\frac{N_{Eq9}}{N_{Eq8}} = \frac{Q(n)c_3 + 2\int_0^n x^2 \ln x\,dx}{Q(n)c_2 + Q(n)\ln n}$$

Integrating by parts using $u=\ln x$ and $dv=x^2dx$ we get $\int_0^n x^2 \ln x\,dx = \frac{1}{3}n^3\left(\ln n - \frac{1}{3}\right)$

$$\frac{N_{Eq9}}{N_{Eq8}} = \frac{Q(n)c_3 + \frac{2}{3}n^3 \ln n - \frac{2}{9}n^3}{Q(n)c_2 + Q(n)\ln n}$$

If we look at the partial derivative with respect to n the ratio is dominated by the $n^3 \ln n$ terms which again converge to two. Like the previous ratio, the partial derivative with respect to δ limits to one as delta tends to zero. Some plots follow for values of n and δ:



The preceding analysis indicates that if a large number of hill-climbing steps are expected, there is probably little to be gained in carefully allocating the error across steps. Even though Equation 9 has the undesirable property that it "wastes" statistical error and punishes later steps, this does not translate into a significant increase in the number of examples required. Perhaps a more interesting consequence of the analysis is the rapid growth in examples required, using these conservative methods for determining step error. With the acceleration function of $U(i) = C/i$, the total number of examples grows cubically with the number of steps. While this is still polynomial, and therefore tractable in the traditional sense, it is a pragmatic concern.[15]

C.2 PROPERTIES OF $Q(n, \alpha)$

The function $Q(n, \alpha)$ models the discrepancy between the estimated mean of a distribution, based on a fixed sample of data points, and the true mean of the distribution. COMPOSER relies on the Central Limit Theorem, which states that difference between the sample mean and the true mean of a distribution is (approximately) normally distributed. We restate the theorem here.

Central Limit Theorem: If $\overline{X}$ is the mean of a random sample $X_1, X_2, ..., X_n$ of size n from a distribution with a finite mean μ and finite positive variance σ^2, then the distribution of

$$W = \frac{\overline{X} - \mu}{\sigma / \sqrt{n}} = \frac{\sum_{i=1}^{n} X_i - n\mu}{\sqrt{n}\,\sigma}$$

is normal with zero mean and unit variance, standard normal, in the limit as n approaches infinity.

The Central Limit Theorem is useful for approximating the distribution of W when n is "sufficiently large." Generally, if n is greater than about 25, the approximation will be quite good. Furthermore, if the distribution is continuous, symmetric, and unimodal, values as small as 4 or 5 will be adequate.

15. We thank Russell Greiner for his assistance in both of these derivations. Dale Schuurmans has noted that replacing Equation 9 with Rissanen's sequence, $\beta_i = \ln c{\times}i \log^* i / \delta$ where c=2.865064, leads to an asymptotic ratio of one.

Typically the variance of the underlying distribution is unknown. In this case the sample variance can be used when computing *W*. This is the technique used in COMPOSER. Using the sample mean also results in a distribution that in the limit approaches a standard normal distribution. However this approximation typically requires somewhat larger *n* to attain a "sufficiently large" sample. For smaller samples, the discrepancy between the sample and true means is better approximated using the student *t* distribution. The *t* distribution converges quickly to the standard normal distribution as show in Figure 15. In this figure, the dotted line shows the standard normal distribution. The curves are almost indistinguishable when *W* is based on a sample of size ten.

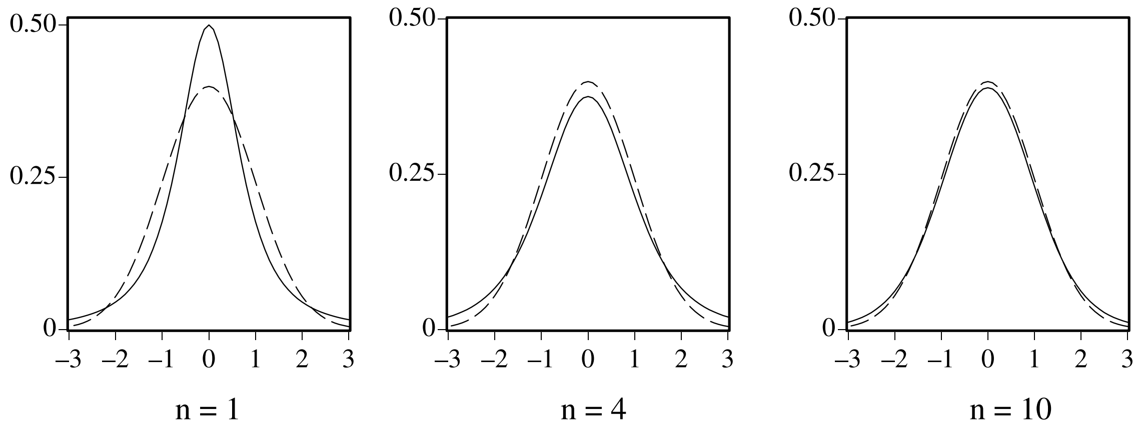


Figure 15: Comparison of standard normal and student t distribution. The dotted line illustrates the standard normal distribution. The *t* distribution is a function of sample size. The graphs compare the distributions where the sample is of size one, four, and ten.

The accuracy of the normal approximation depends on the properties of the original distribution. As stated above, the distribution of the sample mean quickly converges to a normal distribution when the original distribution is continuous, symmetric and unimodal. If the original distribution is normal, the approximation is exact. Figure 16 shows how well the standard normal approximates a sample derived from a uniform distribution (normal shown in dashes). The approximation is quite close after a "sufficiently large" sample size of only four.

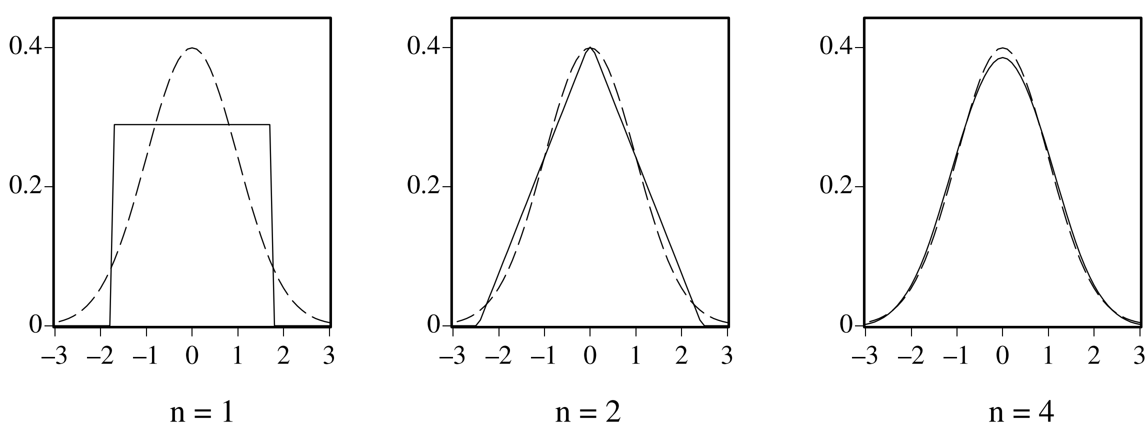


Figure 16: Comparison of the normal approximation of a uniform distribution. The approximation is very close after only four data points. Original distribution is uniform in the interval [0..1]. Graphs show normalized distributions.

Figure 17 shows the normal approximation where the original distribution is chi-square, a highly non-symmetric distribution. In this case the "sufficiently large" sample is considerably larger than with the uniform distribution. Even after fifty examples there is still noticeable discrepancy.

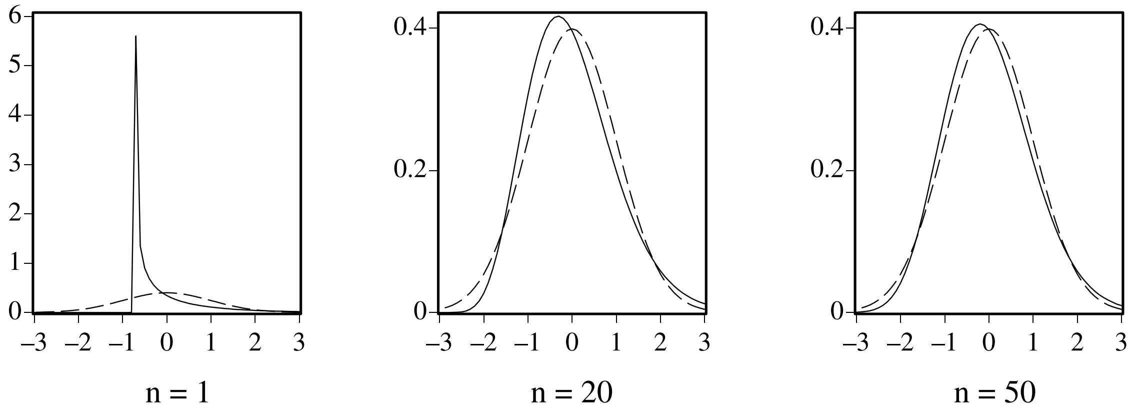


Figure 17: Comparison of the normal approximation of a chi-square distribution.
In this case the sample mean takes significantly more examples to converge to a normal.

The lesson from the chi-square distribution is that some care must be taken in making the normal approximation. If it is known in advance that the distribution is slow to converge to a normal, it may be possible to choose a more appropriate model, such as a chi-square. Fortunately for us, and the general enterprise of statistics, such problematic distributions appear relatively rare. Such normal approximations are ubiquitous in scientific tests of experimental significance.

BIBLIOGRAPHY

[Abramowitz65] M. Abramowitz and I. A. Stegun, *Handbook of Mathematical Functions*, Dover Publications, inc., New York, 1965.

[Agre87] P. Agre and D. Chapman, "Pengi: An Implementation of a Theory of Activity," *Proceedings of the National Conference on Artificial Intelligence*, Seattle, WA, July 1987, pp. 268–272.

[Bell92] C. E. Bell, "Scheduling Deep Space Network Data Transmissions: A Lagrangian Relaxation Approach," *Internal JPL report*, July 1992.

[Berger80] J. O. Berger, *Statistical Decision Theory and Bayesian Analysis*, Springer Verlag, 1980.

[Borgida89] A. Borgida and D. W. Etherington, "Hierarchical Knowledge Bases and Efficient Disjunctive Reasoning," *Proceedings of the First International Conference on Principles of Knowledge Representation and Reasoning*, Toronto, Ontario, Canada, May 1989, pp. 33–43.

[Bylander92] T. Bylander, "Complexity Results for Extended Planning," *Proceedings of the First International Conference on Artificial Intelligence Planning Systems*, College Park, Maryland, June 1992, pp. 20–27.

[Chapman87] D. Chapman, "Planning for Conjunctive Goals," *Artificial Intelligence 32*, 3 (1987), pp. 333–378.

[Cheeseman89] P. Cheeseman, B. Kanefsky and W. M. Taylor, "Where the Really Hard Problems Are," *Proceedings of the Twelfth International Joint Conference on Artificial Intelligence*, Sidney, Australia, August 1989, pp. 163–169.

[DeJong86] G. F. DeJong and R. J. Mooney, "Explanation–Based Learning: An Alternative View," *Machine Learning 1*, 2 (April 1986), pp. 145–176. (Also appears as Technical Report UILU–ENG–86–2208, AI Research Group, Coordinated Science Laboratory, University of Illinois at Urbana–Champaign.)

[Dean88] T. Dean and M. Boddy, "An Analysis of Time–Dependent Planning," *Proceedings of The Seventh National Conference on Artificial Intelligence*, Saint Paul, MN, August 1988, pp. 49–54.

[Dechter87] R. Dechter and J. Pearl, "Network–Based Heuristics for Constraint–Satisfaction Problems," *Artificial Intelligence 34*, 1 (December 1987), pp. 1–38.

[Dechter92] R. Dechter, "Constraint Networks,"in *Encyclopedia of Artificial Intelligence*, Stuart C Shapiro (ed.), John Wiley and Sons, Inc., 1992.

[Doyle90] J. Doyle, "Rationality and its Roles in Reasoning (extended version)," *Proceedings of the National Conference on Artificial Intelligence*, Boston, MA, 1990, pp. 1093–1100.

[Duncker45] K. Duncker, "On Problem Solving," *Psychological Monographs 58*, 270 (1945),.

[Eorl92] K. Eorl, D. S. Nau and V. S. Subrahmanian, "On the Complexity of Domain–Independent Planning," *Proceedings of the National Conference on Artificial Intelligence*, San Jose, CA, July 1992, pp. 381–386.

[Etzioni90a] O. Etzioni, "A Structural Theory of Search Control," Ph.D. Thesis, Department of Computer Science, Carnegie–Mellon University, Pittsburgh, PA, In preparation, 1990.

[Etzioni90b] O. Etzioni, "Why Prodigy/EBL Works," *Proceedings of the National Conference on Artificial Intelligence*, Boston, MA, August 1990, pp. 916–922.

[Etzioni91] O. Etzioni, "STATIC a Problem–Space Compiler for PRODIGY," *Proceedings of the National Conference on Artificial Intelligence*, Anaheim, CA, July 1991, pp. 533–540.

[Etzioni92] O. Etzioni and S. Minton, "Why EBL Produces Overly–Specific Knowledge: a Critique of the PRODIGY Approaches," *Proceedings of the Ninth International Conference on Machine Learning*, Aberdeen, Scotland, July 1992, pp. 137–143.

[Fikes71] R. E. Fikes and N. J. Nilsson, "STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving," *Artificial Intelligence 2*, 3/4 (1971), pp. 189–208.

[Fisher81] M. Fisher, "The Lagrangian Relaxation Method for Solving Integer Programming Problems," *Management Science 27*, 1 (1981), pp. 1–18.

[Freuder82] E. C. Freuder, "A Sufficient Condition for Backtrack–Free Search," *J. Association for Computing Machinery 29*, 1 (January 1982), pp. 24–32.

[Goldberg82] A. Goldberg, P. W. Purdom and C. A. Brown, "Average time analysis of simplified Davis–Putnam procedures," *Information Process. Lett. 15*, (1982), pp. 72–75.

[Govindarajulu81] Z. Govindarajulu, *The Sequential Statistical Analysis*, American Sciences Press, INC., Columbus, OH, 1981.

[Gratch91a] J. Gratch and G. DeJong, "A Hybrid Approach to Guaranteed Effective Control Strategies," *Proceedings of the Eighth International Workshop on Machine Learning*, Evanston, IL, June 1991.

[Gratch91b] J. M. Gratch and G. F. DeJong, "On comparing operationality and utility," Technical Report UIUCDCS–R–91–1713, Department of Computer Science, University of Illinois, Urbana, IL, 1991. (Submitted to Machine Learning Journal)

[Gratch92a] J. Gratch and G. DeJong, "COMPOSER: A Probabilistic Solution to the Utility Problem in Speed–up Learning," *Proceedings of the National Conference on Artificial Intelligence*, San Jose, CA, July 1992, pp. 235–240.

[Gratch92b] J. Gratch and G. DeJong, "A Framework of Simplifications in Learning to Plan," *First International Conference on Artificial Intelligence Planning Systems*, College Park, MD, 1992, pp. 78–87.

[Gratch93a] J. Gratch and G. DeJong, "Rational Learning: A Principled Approach to Balancing Learning and Action," Technical Report UIUCDCS–R–93–1801, Department of Computer Science, University of Illinois, Urbana, IL, April 1993.

[Gratch93b] J. Gratch and S. Chien, "Learning Search Control Knowledge for the Deep Space Network Scheduling Problem: Extended Report and Guide to Software," Technical Report UIUCDCS–R–93–1789, Department of Computer Science, University of Illinois, Urbana, IL, January 1993.

[Greiner89] R. Greiner and J. Likuski, “Incorporating Redundant Learned Rules: A Preliminary Formal Analysis of EBL,” *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, Detroit, MI, August 1989, pp. 744–749.

[Greiner92a] R. Greiner and I. Jurisica, “A Statistical Approach to Solving the EBL Utility Problem,” *Proceedings of the National Conference on Artificial Intelligence*, San Jose, CA, July 1992, pp. 241–248.

[Greiner92b] R. Greiner and W. W. Cohen, “Probabilistic Hill–Climbing,” *Proceedings of Computational Learning Theory and ’Natural’ Learning Systems*, 1992. ((to appear))

[Harmellen88] F. Harmellen and A. Bundy, “Explanation–Based Generalisation = Partial Evaluation,” *Artificial Intelligence 36*, (1988), pp. 401–412.

[Horvitz89] E. J. Horvitz, G. F. Cooper and D. E. Heckerman, “Reflection and Action Under Scarce Resources: Theoretical Principles and Empirical Study,” *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, Detroit, MI, August 1989, pp. 1121–1127.

[Knoblock89] C. Knoblock, “Learning Hierarchies of Abstraction Spaces,” *Proceedings of the Sixth International Workshop on Machine Learning*, Ithaca, NY, 1989, pp. 241–245.

[Korf87] R. E. Korf, “Planning as Search: A Quantitative Approach,” *Artificial Intelligence 33*, (1987), pp. 65–88.

[Laird86] J. E. Laird, P. S. Rosenbloom and A. Newell, *Universal Subgoaling and Chunking: The Automatic Generation and Learning of Goal Hierarchies*, Kluwer Academic Publishers, Hingham, MA, 1986.

[Laird92] P. Laird, “Dynamic Optimization,” *Proceedings of the Ninth International Conference on Machine Learning*, Aberdeen, Scotland, July 1992, pp. 263–272.

[Letovsky90] S. Letovsky, “Operationality Criteria for Recursive Predicates,” *Proceedings of the National Conference on Artificial Intelligence*, Boston, MA, August 1990, pp. 936–941.

[Lorenz78] K. Lorenz, *Behand the Mirror: a search for a natural history of human knowledge*, FirstHarvest/HBJ, New York, NY, 1978.

[Luchins50] A. S. Luchins and E. H. Luchins, “New Experimental Attempts at Preventing Mechanization in Problem Solving,” *The Journal of General Psychology 42*, (1950), pp. 279–297.

[Mackworth92] A. K. Mackworth, “Constraint Satisfaction,”in *Encyclopedia of Artificial Intelligence*, Stuart C Shapiro (ed.), John Wiley and Sons, Inc., 1992.

[Maes90] P. Maes and R. A. Brooks, “Learning to Coordinate Behaviors,” *Proceedings of the National Conference on Artificial Intelligence*, Boston, MA, August 1990, pp. 796–802.

[Medin89] D. L. Medin and B. H. Ross, “The Specific Character of Abstract Thought: Categorization, Problem Solving, and Induction,”in *Advances in the Psychology of Human Intelligence*, R. Sternberg (ed.), Lawrence Erlbaum and Associates, Hillsdale, NJ, 1989.

[Melhorn84] K. Melhorn, *Data Structures and Algorithms 1: Sorting and Searching*, Springer Verlag, 1984.

[Miller92] D. P. Miller, R. S. Desai, E. Gat, R. Ivlev and J. Loch, “Reactive Navigation through Rough Terrain: Experimental Results,” *Proceedings of the National Conference on Artificial Intelligence*, San Jose, CA, July 1992, pp. 823–828.

[Minton88a] S. N. Minton, “Learning Effective Search Control Knowledge: An Explanation–Based Approach,” Ph.D. Thesis, Department of Computer Science, Carnegie–Mellon University, Pittsburgh, PA, March 1988. (Also appears as CMU–CS–88–133)

[Minton88b] S. Minton,in *Learning Search Control Knowledge: An Explanation–Based Approach*, Kluwer Academic Publishers, Norwell, MA, 1988.

[Mitchell86] T. M. Mitchell, R. Keller and S. Kedar–Cabelli, "Explanation–Based Generalization: A Unifying View," *Machine Learning 1*, 1 (January 1986), pp. 47–80.

[Mitchell92] D. Mitchell, B. Selman and H. Levesque, "Hard and Easy Distributions of SAT Problems," *Proceedings of the National Conference on Artificial Intelligence*, San Jose, CA, July 1992, pp. 459–465.

[Mostow81] J. Mostow, "Mechanical Transformation of Task Heuristics into Operational Procedures," Ph.D. Thesis, Department of Computer Science, CMU, Pittsburgh, PA, 1981.

[Nádas69] A. Nádas, "An extension of a theorem of Chow and Robbins on sequential confidence intervals for the mean," *The Annals of Mathematical Statistics 40*, 2 (1969), pp. 667–671.

[Natarajan89] B. K. Natarajan, "On Learning from Exercises," *Proceedings of the Second Annual Workshop on Computational Learning Theory*, Santa Cruz, CA, JULY 1989, pp. 72–87.

[Pérez92] M. A. Pérez and O. Etzioni, "DYNAMIC: a new rule for trining problems in EBL," *Proceedings of the Ninth International Conference on Machine Learning*, Aberdeen, Scotland, July 1992, pp. 367–372.

[Russell89] S. Russell and E. Wefald, "Principles of Metareasoning," *Proceedings of the First International Conference on Principles of Knowledge Representation and Reasoning*, Toronto, Ontario, Canada, May 1989, pp. 400–411.

[Schoenfeld82] A. H. Schoenfeld and D. Herrmann, "Problem Perception and Knowledge Structure in Expert and Novice Mathematical Problem Solvers," *Journal of Experimental Psychology: Learning, Memory, and Cognition 8*, 5 (1982), pp. 484–494.

[Schoppers92] M. Schoppers, "Building Plans to Monitor and Expoint Open–Loop and Closed–Loop Dynamics," *Proceedings of the First International Conference on Artificial Intelligence Planning Systems*, College Park, Maryland, June 1992, pp. 204–213.

[Schwuttke92] U. M. Schwuttke and L. Gasser, "Real–time Metareasoning with Dynamic Trade–off Evaluation," *Proceedings of the National Conference on Artificial Intelligence*, San Jose, CA, July 1992, pp. 500–506.

[Subramanian90] D. Subramanian and R. Feldman, "The Utility of EBL in Recursive Domain Theories," *Proceedings of the National Conference on Artificial Intelligence*, Boston, MA, August 1990, pp. 942–949.

[Subramanian92] D. Subramanian and S. Hunter, "Measuring Utility and the Design of Provably Good EBL Algorithms," *Proceedings of the Ninth International Conference on Machine Learning*, Aberdeen, Scotland, July 1992, pp. 426–435.

[Tadepalli91] P. Tadepalli, "Learning with Inscrutable Theories," *Proceedings of the Eighth International Workshop on Machine Learning*, Evanston, IL, June 1991, pp. 544–548.

[Taha82] H. A. Taha, *Operations Research, an Introduction*, Macmillan Publishing Co., INC., 1982.

[Wellman92] M. P. Wellman and J. Doyle, "Modular Utility Representation for Decision–Theoretic Planning," *Proceedings of the First International Conference on Artificial Intelligence Planning Systems*, College Park, Maryland, June 1992, pp. 236–242.