ELSEVIER

Artificial Intelligence 88 (1996) 101-142

Artificial Intelligence

A statistical approach to adaptive problem solving

Jonathan Gratch [a,*], Gerald DeJong [b,1]

[a] *Information Sciences Institute, University of Southern California, 4676 Admirality Way, Marina del Rey, CA 90292, USA*

[b] *Beckman Institute, University of Illinois, 405 N. Mathews, Urbana, IL 61801, USA*

Received May 1994; revised February 1995

Abstract

Domain independent general purpose problem solving techniques are desirable from the standpoints of software engineering and human computer interaction. They employ declarative and modular knowledge representations and present a constant homogeneous interface to the user, untainted by the peculiarities of the specific domain of interest. Unfortunately, this very insulation from domain details often precludes effective problem solving behavior. General approaches have proven successful in complex real-world situations only after a tedious cycle of manual experimentation and modification. Machine learning offers the prospect of automating this adaptation cycle, reducing the burden of domain specific tuning and reconciling the conflicting needs of generality and efficacy. A principal impediment to adaptive techniques is the utility problem: even if the acquired information is accurate and is helpful in isolated cases, it may degrade overall problem solving performance under difficult to predict circumstances. We develop a formal characterization of the utility problem and introduce COMPOSER, a statistically rigorous learning approach which avoids the utility problem. COMPOSER has been successfully applied to learning heuristics for planning and scheduling systems. This article includes theoretical results and an extensive empirical evaluation. The approach is shown to outperform significantly several other leading approaches to the utility problem.

1. Introduction

There is a wide gulf between *general* approaches and *effective* approaches to problem solving. Practical success has come from custom techniques like expert systems, reactive systems [52, 66], or other application specific techniques that require extensive human

* Corresponding author. E-mail: gratch@isi.edu.

[1] E-mail: dejong@cs.uiuc.edu.

PII S0004-3702(96)00011-2

investment to complete. AI researchers have also developed domain independent algorithms such as nonlinear planning and constraint satisfaction algorithms. Unfortunately, when general approaches show success, it is usually only after extensive domain specific adjustments. The resulting systems, while derived from a general technique, bear more resemblance to the custom approaches.

Adaptive problem solving is a potential means for circumventing this generality/performance tradeoff in repetitive problem solving situations. We want our problem solvers to work well for the problems they actually encounter and we care not at all about performance on other hypothetical problems. Worst-case behavior is largely irrelevant to real-world problem solving. Pragmatically, by sacrificing good behavior on unseen or unlikely problems, a system's overall performance may be enhanced. In fact, machine learning techniques have successfully demonstrated the capacity to enhance problem solving performance, although in limited contexts [18, 46, 57, 65]. Nonetheless, adaptive problem solving is still far from realized in any general sense.

The principal impediment to adaptive problem solving is characterizing when an automatically hypothesized adaptation actually results in improved problem solving performance. Steve Minton introduced to machine learning the term *utility problem* to refer to this difficulty of insuring performance improvements [53]. Minton originally discussed the problem in the context of improving the average problem solving speed via search control heuristics called control rules. While there has been considerable progress on this issue [15, 40, 48, 53], the proposed methods are often ad hoc, poorly understood, and can fail to improve performance, or worse, actually degrade problem solving performance under certain circumstances.

Adaptivity can be an effective method for improving problem solving performance in that real-world problems are often constrained in ways that only become obvious through experience. On one hand, the domain specification may implicitly embed constraints that are difficult to deduce *a priori*, as in the blocksworld domain where a block can never be atop itself, though this constraint is not explicitly represented. On the other hand, the distribution of tasks embeds many constraints that can only be induced from experience. For example, a particular blocksworld application may never contain towers of height greater than three.

Exploiting these regularities can lead to clear performance improvements. Of course, the problem solver cannot look into the future to anticipate particular problems. However, it can generalize from characterizations of past problems. Most distributions exhibit peculiarities that may be exploited once detected. Conversely, even worst-case intractable algorithms may perform well under certain distributions. For example, Goldberg suggests that naturally occurring satisfiability problems are frequently solved in $O(n^2)$ time [22]. Recent work has focused on characterizing these easy distributions [5, 56] or devising techniques that can exploit specific distribution information when it is available [3]. Research into self-organizing systems [51, pp. 252–285] and dynamic optimization [45] exploit the fact that learning the expected distribution of tasks can allow the construction of a problem solver with substantially better expected performance.

In the next section, we provide a formal characterization of the utility problem in decision theoretic terms. This introduces the notion of expected utility as a metric of problem solving performance and casts the learning as a search through a space of

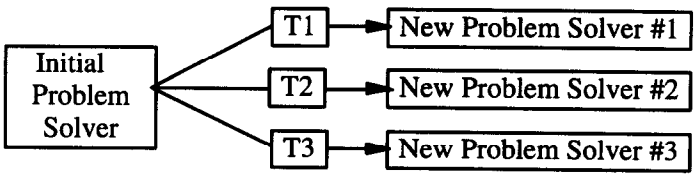


Fig. 1. Learning transforms an initial problem solver into a new one. To accomplish this the learner must choose one of a set of possible transformations.

problem solving transformations for a problem solver with high expected utility. Section 3 describes the COMPOSER algorithm, a probabilistic approach to avoiding the utility problem. COMPOSER performs a probabilistic search through the transformation space and incorporates several techniques to ensure the efficiency of this process. Section 4 describes an extensive evaluation of the approach in the context of learning control rules for a domain independent problem solver. The experiments indicate that COMPOSER compares favorably with existing approaches to the utility problem. Section 5 provides an average case analysis of the algorithm's complexity. COMPOSER's run time is shown to be polynomial in the number of transformations considered and in the statistical confidence required. Finally, we discuss some limitations and present conclusions.

2. A formal characterization of the utility problem

Before a rigorous learning algorithm can be constructed, we must explicitly characterize what the learning system is attempting to achieve. In this section, we introduce a formal characterization of a general class of learning problem solvers. This formalism makes precise and explicit intuitive and often unstated notions of what a learning system should do, and it provides a structure for formal analysis, enabling us to make definitive and precise statements.

Abstractly, a learning algorithm operates on an initial problem solver, transforming it into a different problem solver, where the effect of this change is assessed relative to some evaluation criterion and a pattern of tasks associated with the intended application. This is illustrated in Fig. 1 where a set of hypothesized *transformations* defines a set of potential problem solvers. The learning system must decide which hypothesized transformations to adopt, where the outcome of this decision is a new problem solver. To characterize "good" outcomes, we use the decision theoretic notion of *expected utility* [2] as a common framework for characterizing this decision problem. Doyle has argued for the merits of decision theory as a standard for evaluating artificial intelligence systems [13] and it has seen increasing acceptance, both in artificial intelligence at large [41, 64, 67, 71] and machine learning in particular [30, 35, 45, 69].

2.1. Expected utility

Decision theory relies on the observation that preferences over different outcomes can, under some natural assumptions, be characterized by a real-valued *utility function*.

Outcome A is preferred to outcome B iff the utility of A is greater than the utility of B.

In an uncertain world, a decision may not always produce the same outcome. Even the best decision policy may do very well one time and poorly the next. The correct decision policy under uncertainty maximizes *expected* utility: a decision is characterized by a set of outcomes and a probability distribution over this set, and the expected utility of a decision is the utility of all possible outcomes weighted by their probability of occurrence.

For learning, the characteristics of the intended application determine what outcomes (transformed problem solvers) are preferred. With decision theory, we represent these preferences with a utility function, allowing learning to be cast as the decision problem of choosing a transformation that increases expected utility. However, we require the application to obey the following two restrictions.

Fixed distribution assumption. The pattern of tasks in the problem solving environment must be characterizable as a random selection of tasks according to a fixed (but unknown) probability distribution over the domain of discourse. This restriction states that there is some probability of occurrence associated with each task that is independent of the tasks 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 limitations that we discuss further in Section 6.

Expected utility assumption. The evaluation criterion must be expressed by a *utility function*. This is a function from a problem solver and problem to a numeric value. The function must be chosen such that problem solver A is preferred over problem solver B if and only if the expected utility of A is greater than the expected utility of B. Decision theory posits the *expected utility hypothesis*: that there exists such a utility function for any consistent set of preferences (see [11, Ch. 7]). The utility function must also be computable by the learning system. For example, if the application requires an efficient problem solver, the utility function could be the CPU cost to solve a problem (or to determine it cannot be solved). This can be computed by actually attempting to solve the problem and measuring the time. The expected utility of a problem solver would then be its average problem solving time for the given fixed problem distribution. If the problem solver is semi-decidable, one could proceed by imposing a resource bound and assigning some suitable utility value to the conclusion "I don't know" (see [34]).

It may not be immediately clear how to represent preferences in terms of a single utility measure. In many real-world situations, comparisons between actions are made on the basis of several performance attributes. In planning problems, we may care not only about how fast a plan is created, but also about other attributes such as its execution cost or robustness. A large literature in the decision sciences is devoted to how to translate these "multi-attribute" problems into a single utility function (see, for example, [63]).

With the fixed distribution and expected utility assumptions we can characterize the value of a problem solver by its expected utility. Formally, let PS denote a problem solver, D denote the set of possible problems expressible in the domain, $\Pr_D(x)$ be the probability of occurrence for problem $x \in D$, and $U(PS, x)$ be the utility of PS

on problem x. The expected utility of PS with respect to the distribution D, written $E_D[U(PS)]$, is defined as:

$$E_D[U(PS)] = \begin{cases} \int_D U(PS,x)\Pr_D(x)\mathrm{d}x & (D \text{ continuous}), \\ \sum_{x \in D} U(PS,x)\Pr_D(x) & (D \text{ discrete}). \end{cases} \quad (1)$$

2.2. Composite transformations

Next we must formalize the effects of learning. A learning algorithm maps some initial problem solver PS_{old} into a new problem solver PS_{new}. We introduce the notion of a *composite transformation* to denote the structural changes performed to a problem solver in the course of learning. A composite transformation is whatever is required to transform PS_{old} into PS_{new} and it may be built from several component structural changes. For example, the PRODIGY/EBL system [53] builds a composite transformation from a set of learned control rules. A given learning algorithm has the potential to produce a variety of composite transformations depending on the initial problem solver and the distribution of observed problems. This corresponds to the notion of a hypothesis space in classification learning and we characterize it as a set of possible composite transformations. Alternatively, this set can be thought of as the set of all problem solvers reachable by the learning algorithm. Note that this set will be quite large (possibly infinite) for any nontrivial learning approach.

A composite transformation maps PS_{old} into some PS_{new}. The value of this composite transformation can be measured by the difference in expected utility between PS_{new} and PS_{old}. This provides a metric for assessing the performance of a learning algorithm.

2.3. Optimality versus improvement

Expected utility defines a total preference ordering over a set of composite transformations associated with a learning approach. The ideal learning algorithm would choose the composite transformation in this set with the highest expected utility. Such an algorithm can be considered optimal with respect to the set of entertained composite transformations. One might consider optimality as part of the requirement for avoiding the utility problem—that is, a learning algorithm should not only avoid lowering expected utility, but it should avoid sub-optimal improvements as well.

Unfortunately, optimality is an extremely expensive requirement. For many machine learning algorithms, it is computationally intractable to identify the optimal transformation. For example Greiner shows the inherent difficulties when transformations are constructed from macro-operators [36]. We have chosen, therefore, not to insist on optimality and instead we adopt a weaker requirement. In our analysis, when a learning algorithm adopts a composite transformation it must increase expected utility, but it need not be the optimal choice.

In fact, just improving the problem solver may be beyond the capabilities of a learning algorithm. If all of the composite transformations lead to a *decrease* in expected utility, a

learning algorithm should ignore all of the transformations, and leave the initial problem solver unchanged.

2.4. Unknown information

For a given learning problem there is some true but unknown expected utility associated with each possible transformed problem solver: both the utility of a transformed problem solver on any given problem, and the probability distribution over the space of possible problems are typically unknown. A learning system can only estimate this information through training examples. For example, a learning system might estimate the value of a transformation by solving some randomly selected problems with the original and transformed problem solvers. Increasing the number of examples would improve the estimate, but it may still differ from the true expected utility due to sampling error.

Unknown information means a learning algorithm cannot guarantee that a composite transformation increases expected utility. There is always the possibility that a transformation is estimated to increase expected utility when in fact it does not. Nevertheless, in characterizing the utility problem we would like to explicitly quantify the chance that learning does not improve performance. Therefore, we adopt a probabilistic requirement. An algorithm which solves the utility problem may adopt composite transformations with negative expected utility as long as this event occurs with probability less than some pre-specified amount, which may be arbitrarily close to zero.

2.5. The utility problem

A learning algorithm exhibits the utility problem when it lowers the expected utility of the initial problem solver. More precisely, given: (1) an application described by a utility function, a set of problems, and access to problems drawn according to a fixed probability distribution, (2) an initial problem solver for this application, PS_{old}, (3) a confidence parameter, $1 - \delta$, a learning algorithm exhibits the utility problem whenever: (a) it produces some transformed problem solver $PS_{\text{new}} \neq PS_{\text{old}}$ and (b) with probability greater than δ, PS_{new} has lower expected utility than PS_{old}, with respect to the fixed probability distribution. Under this definition, if an adaptive problem solver adopts some composite transformation, it must, with high probability, improve performance. We say a learning algorithm *solves* the utility problem if it satisfies this requirement.

Admittedly, solving the utility problem, thus defined, is a weak requirement on a learning algorithm. For example, one trivial way to solve the utility problem is to avoid learning anything. What the requirement does provide, however, is a measure of confidence in whatever changes a learning algorithm may make. That is, if a learning algorithm modifies a problem solver, while at the same time avoiding the utility problem, we may have confidence that those modifications will improve problem solving performance on problems drawn according to the same probability distribution. Ideally, one would like to strengthen this minimal requirement with additional restrictions on when it is acceptable not to learn. While a general statement of these requirements is outside the scope of this article, the subsequent section discusses some stronger requirements satisfied by our COMPOSER system.

3. A solution to the utility problem

The preceding framework not only defines the goal of a learning technique, it suggests a natural solution to the utility problem. The effectiveness of a transformation can be judged in terms of its expected utility, which can be estimated to an arbitrary level of confidence using statistical procedures. Beyond solving the utility problem, our COMPOSER system, which embodies this basic solution, can be shown to improve performance with high probability given that certain requirements of the domain and transformations are satisfied. However there are many difficulties that must be resolved before this basic solution can be realized in an efficient and practical algorithm. This section outlines our strategies for addressing these difficulties and ends with a presentation of the algorithm.

3.1. Incremental learning

The most significant difficulty arises in how to efficiently investigate the vast set of possible composite transformations. Frequently there is some internal structure to transformations that can be exploited. In most learning techniques, a composite transformation consists of many individual *atomic transformations* (later we refer to atomic modifications simply as *transformations*). For example, SOAR constructs a new problem solver from individual chunks [46]. PRODIGY/EBL builds a learned control strategy from individual control rules [53]. In such systems,composite transformations may share many individual components.

Instead of making a global decision among all possible composite transformations, an *incremental* learning system builds up a composite transformation by making many local decisions. The *composability problem* is the name we give to the problem of identifying an effective composite transformation given that it must be constructed from multiple atomic transformations. This is analogous to the planning problem. A planner does not solve a goal by searching through the set of all complete plans. Rather it uses operators to make incremental progress. In COMPOSER, we view atomic transformations as *learning* operators. Just as individual planning operators can be flexibly combined to solve a variety of goals, individual atomic transformations can be combined to address the particular combination of problem solving inefficiencies. In fact, most learning for planning techniques can be viewed from this perspective, although they are not generally described in such terms.

3.1.1. Operationality criteria: independent measures

The composability problem constrains the acceptable local measures of atomic transformation quality. A measure must ensure that locally beneficial transformations combine into a beneficial composite. One solution is to develop *independent measures*. These are local measures that assess the quality of an atomic transformation independently of the other transformations that appear in the final composite transformation. With such a measure, transformations can be separately evaluated on sampled problems and easily combined. Mitchell et al.'s operationality criteria [54] and Etzioni's non-recursive hypothesis [14] are attempts to provide such a measure. Incidentally, these are both in-

dependent of the problem distribution as well, obviating the need for statistical validation of the transformations.

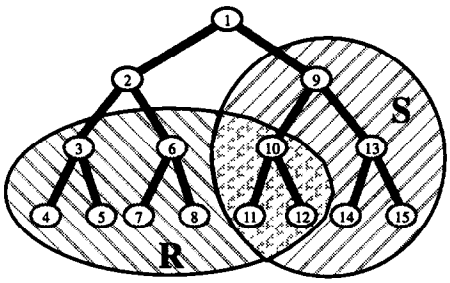
Unfortunately, independent measures are in general not possible. Atomic transformations potentially interact with each other in difficult to predict ways. Such interactions have long been recognized as a source of difficulty in planning, but have been overlooked in learning, often with unfortunate consequences (as illustrated in Section 4). Simple syntactic independent measures like operationality and the non-recursive hypothesis, while useful heuristics, cannot provide even weak guarantees against detrimental results.

As an intuitive illustration, consider the problem of selecting a satisfactory sequence of dishes at a Dim Sum restaurant. One seasoned patron advises "Avoid things that exude a faint hazelnut-like odor". Another who is equally trustworthy says "Don't eat things prepared with Xanthin leaves". Each rule alone may improve our chance for an enjoyable meal while together they diminish it. This is because the rules interact. Suppose there is one particularly evil dish, a large green blob of dough that smells of hazelnut and is typically served in a Xanthin leaf sauce. Indeed it may be this single offensive entree that our two friends were warning us about, each in his own way. Each rule avoids this dish but also eliminates other delicious items whose only sin is to have a single superficial feature in common with the horrible one. Furthermore, we must pay the overhead of evaluating both rules: we must persuade the waiter to allow us to sniff each dish as it comes by and we must explain in broken Cantonese that we wish to avoid Xanthin leaves when, in point of fact, we have no idea what a Xanthin leaf is. Depending on our sensitivity to awkward social situations we might be better off to simply forget both rules and risk a taste of the green doughy blob.

Here, two rules avoid the same penalty. As there is no added benefit in eliminating it twice, the utility of the two together is not equivalent to the sum of their improvements in isolation. A more subtle interaction involves a rule that has an evaluation cost which is sensitive to the context in which it is evaluated. A second rule that significantly alters the problem solving procedure may substantially change the average evaluation cost of the first rule.

To make this more formal, we focus on the problem of building a composite transformation where the atomic transformations are search control rules, as in PRODIGY/EBL [53]. Let utility be the negative of the time to solve a problem. Control rules can reduce the time to solve a problem by eliminating search, but they introduce an evaluation overhead: the control rule's preconditions must be matched at each node in the search space to see if a portion of the search tree can be pruned. In isolation, a control rule increases utility if the benefits of less search exceed this evaluation cost. Thus, search time saved minus evaluation time is a candidate for an independent evaluation criterion. However, since control rules interact, the improvement of multiple control rules is not the sum of the improvements of the rules in isolation.

Let us quantify this. Consider the interaction between the control rules illustrated in Fig. 2. This shows a hypothetical search space of fifteen nodes which is exhaustively searched by the initial problem solver to conclude that there are no solution nodes under node 1. Suppose r and s are two heuristics that prune the nodes in sets R and S respectively. $|R|$ is the number of nodes trimmed by r. $|S|$ is similarly defined. When



R–S = nodes saved only by r
S–R = nodes saved only by s
R∩S = nodes saved by both r and s

M_r = Average match cost of r
M_s = Average match cost of s
g = Average cost to expand a node

Fig. 2. Example of interacting heuristics.

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. Let the average evaluation cost of r be M_r, the average cost of s be M_s, and the average time to expand a node be g.

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:

$$\begin{aligned} \textit{Residue} &= U(\{r, s\}, p) - [U(\{r\}, p) + U(\{s\}, p)] \\ &= |R - S| \cdot M_r - |R \cap S| \cdot g. \end{aligned}$$

This residue measures the interaction between atomic transformations. The transformations combine synergistically if this value is positive. For example, one control rule may prune sub-trees over which another control rule tends to be expensive to evaluate. When using the first rule, the average match cost of the second decreases. If the residue is negative, they engage in a harmful interaction. For example, there maybe a large overlap in the search they avoid. The key point is that two control rules with positive utility in isolation can potentially combine to yield a strategy worse than neither. Such interactions would seem to preclude effective independent criteria for determining the benefit of atomic transformations.

3.1.2. *Incremental utility: a context sensitive measure*

While we cannot develop a general independent measure of atomic transformation quality, we can develop a *context sensitive* measure that accounts for the context of other atomic transformations participating in the composite transformation. In particular, we can view a composite transformation as a sequence of intermediate problem solvers: $PS_{\text{old}}, PS_1, PS_2, \ldots, PS_{\text{new}}$, each with some expected utility. We adopt a context sensitive measure of utility that states how much a given atomic transformation improves expected utility if appended to an existing sequence of transformations.

Let PS denote a problem solver and let $PS^\tau = \textit{Apply}(\tau, PS)$ denote the problem solver that results from applying an atomic transformation τ to PS. We define the *incremental utility* of τ to be the difference between the expected utility of PS^τ 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. We can state this formally as:

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

or equivalently:

$$\Delta U_D(\tau|PS) = \begin{cases} \displaystyle\int_D [U(PS^\tau, x) - U(PS, x)]\Pr_D(x)\mathrm{d}x & (D \text{ continuous}), \\ \displaystyle\sum_{x\in D} [U(PS^\tau, x) - U(PS, x)]\Pr_D(x) & (D \text{ discrete}). \end{cases} \tag{2}$$

The change in expected utility provided by a composite transformation is equivalent to the sum of the incremental utilities of each transformation:

$$\Delta U_D(\tau_0|PS_{\text{old}}) + \Delta U_D(\tau_1|PS_{\text{old}}^{\tau_0}) + \Delta U_D(\tau_1|PS_{\text{old}}^{\tau_0,\tau_1}) + \cdots.$$

Our definition of incremental utility clarifies two important properties of transformations. First the effect of a transformation on utility 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 indicates that in general we cannot identify a globally maximal composite transformation without considering a potentially explosive number of conditional utility values.

3.2. COMPOSER

COMPOSER is a statistical approach that provably solves the utility problem. Given a learning element that provides atomic transformations, if COMPOSER adopts a composite transformation, it is guaranteed (with pre-specified confidence) to improve expected utility. Additionally, we can show that, given sufficient examples, COMPOSER will adopt a sequence of improving transformations with high probability, given the existence of such a sequence.

COMPOSER requires as input a transformable initial problem solver, a learning element with certain characteristics (specified below), a utility function, and a source of training problems drawn randomly from the problem distribution. We next describe COMPOSER's method for searching the set of composite transformations. We then define the constraints on the method for proposing atomic transformations. Finally we describe how the system achieves its statistical guarantee.

[2] In other learning algorithms incremental utility is simply referred to as utility. We feel the additional terminology helps to highlight the difference between the utility of a problem solver, which is of interest to the user, and the utility of a transformation which is only of interest to the learning algorithm.

3.2.1. Overview

Because they do not compose linearly, it is typically intractable to determine the best sequence of transformations. However, we want as great an improvement as possible. COMPOSER uses incremental utility in conjunction with a greedy hill-climbing procedure to explore the set of possible composite transformations. COMPOSER begins its search 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. One shortcoming is it cannot exploit positive interactions. Solutions, therefore, may be local optima.

3.2.2. Transformation generator

COMPOSER requires a source of transformations for each step in the search. This is abstractly formalized as a function we call a *transformation generator*. This is a function $TG : PS \times X \rightarrow \{\tau_1, \ldots, \tau_k\}$ that maps a problem solver and an optional set 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, SOAR can be viewed as a transformation generator takes the current problem solver and the single training problem that produced an impasse, and generates a set of chunks. Etzioni's STATIC system [14] can be seen as a transformation generator that takes the original problem solver and no training examples, and generates a set of control rules.

3.2.3. Statistical inference

COMPOSER must estimate incremental utility from training data and assess the accuracy of these estimates. There are several philosophical stances for reasoning about these statistical issues. The computational learning community has favored worst-case statistical models (also called *non-parametric* techniques). These are quite useful to make theoretical statements but are too inefficient for most practical uses. *Parametric* techniques provide a more practical alternative. In these, the distribution of utility values is assumed to be a function of a set of predefined parameters with unknown values. Inference reduces to estimating these unknown values from the data. Bayesian models are a popular parametric alternative to worst-case models but they require the specification of prior knowledge about the probable performance of the alternative transformations. We prefer to avoid dependence on prior information, and so have adopted so-called frequentist statistical models which have the efficiency of Bayesian approaches without the need for the specification of prior information.[3] When we must balance between absolutely insuring error does not exceed a bound and achieving reasonable efficiency,

[3] There is controversy between Bayesians and frequentists as to which approach is more appropriate; in many cases it can be shown that Bayesian approaches with non-informative priors are equivalent to frequentist approaches. We do not take a stand on this issue. One might well replace our frequentist model with a Bayesian one without changing the principal theoretical contributions of this work. The straightforward mapping of COMPOSER into a Bayesian framework is to map the probability statement of Eq. (3) into a 0–1 loss function [2, p. 63]. Under this interpretation, δ becomes a bound on the expected loss of a decision.

we err to the side of efficiency, as long there are good arguments that the error bounds are not exceeded in practice. In Appendix A, we describe how to adjust this tradeoff when necessary.

COMPOSER must ensure that the *overall* error remains below some threshold δ. Formally:

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

To achieve this, COMPOSER must account for three sources of error. Each time COMPOSER identifies a new transformation to adopt, it compares a set of estimates, one for each transformation considered at that step. The first source of error is associated with these estimates (there is some probability that a transformation has negative incremental utility, even though it appears positive). Second, these individual errors combine into a somewhat larger probability of error for the overall decision of what to adopt at a given step. Third, the final problem solver, PS_{new}, is produced as a result of several steps, and the error across all of these steps must be accounted for as well.

We define the function $Bound(\delta, |T|)$, which specifies the acceptable error for a utility estimate as a function of the overall error, δ, and the size of the set of transformations, T, at a given step in the hill-climbing search: [4]

$$Bound(\delta, |T|) = \frac{\delta}{|T|}.$$

$Bound(\delta, |T|)$ avoids the utility problem by bounding the first sense of error (the error of each individual incremental utility estimate) in such a way that as they combine into the second and third sense, the overall error remains below the total acceptable bound of δ.

Given this definition it remains to construct a statistical procedure that estimates incremental utility of a transformation to the specified error bound. This involves two issues. First we must determine how estimates are generated from training problems. Second we must decide how many training problems are needed to attain sufficiently accurate estimates.

We can estimate incremental utility by randomly drawing problems according to the distribution D and, for each transformation under consideration, averaging the resulting incremental utility values. Call $\overline{\Delta U_n}(\tau|PS)$ the estimated incremental utility from n training problems. Given a current problem solver, PS, a set of transformations, and a problem x, COMPOSER must determine the difference in utility between the current and each of the transformed problem solvers: $\forall \tau \in T, U(PS^{\tau}, x) - U(PS, x)$. 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 $PS^{\tau_1}, PS^{\tau_2}, \ldots, PS^{\tau_m}$. Processing each training example involves $m+1$ problem solving attempts. The complexity of processing an example is therefore tied to

[4] This definition embodies a compromise between bounding statistical error and example efficiency which works well in practice. The rational behind the compromise is discussed in Appendix A, where we illustrate some other possible definitions.

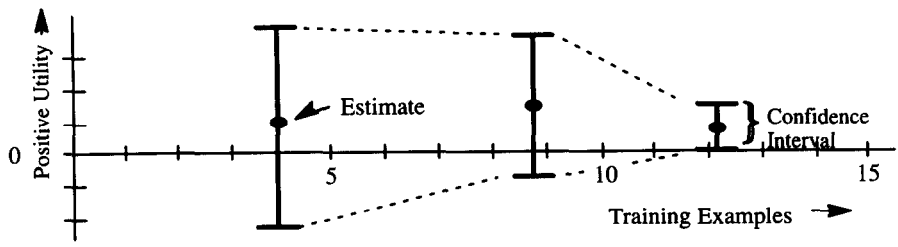


Fig. 3. Sizing the confidence interval. We would like to take enough examples such that the confidence interval lies entirely above or below the axis.

the complexity of each of the $m+1$ problem solvers. We call this *brute-force* processing and it is the default method used by COMPOSER. We show in Appendix A that for many applications there are other more efficient methods for obtaining the incremental utility values.

We can determine a suitable sample size by considering how the accuracy of estimates improves as we increase the number of examples used in the computation. Specifically, COMPOSER must determine a sufficiently large sample size such that it can bound the probability that incremental utility is estimated to be positive when in fact it is negative and vice versa. We would also prefer to use as few examples as possible. COMPOSER relies on a *sequential* statistical technique to determine how many examples are sufficient to make this inference [23]. 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 tends to be significantly less than that required by fixed sample techniques.

We determine the sample size using a stopping rule proposed by Nádas [60]. The intuition behind the stopping rule is quite simple. Given a sample of values and a confidence level α we can construct a confidence interval containing the true incremental utility with probability $1-\alpha$. In other words, there is only probability α that the true incremental utility lies outside the confidence interval. With more examples, the width of the interval tends to shrink.[5] The stopping rule determines how many examples are needed for the interval to shrink to the point of being entirely above or below zero incremental utility, as illustrated in Fig. 3. At this point, we can state, with probability $1-\alpha$, that the transformation has positive (negative) incremental utility if the estimate is positive (negative).

After each example is processed the Nádas stopping rule is evaluated. Sampling terminates when the rule evaluates to true. When this occurs we can state that the given transformation will speed up (slow down) *PS* if its estimated incremental utility is positive (negative) with confidence $1-\alpha$. Examples are taken until the following equation holds:

[5] The interval size is actually a random function of the data. It will tend to shrink with more examples, but not monotonically. It will grow, temporarily, if an outlying data point is encountered.

$$n \geqslant n_0 \quad \text{and} \quad \frac{S_n^2(\tau|PS)}{\left[\overline{\Delta U}_n(\tau|PS)\right]^2} \leqslant \frac{n}{Q(\alpha)^2} \tag{4}$$

where n is the number of examples taken so far, $\overline{\Delta U}_n(\tau|PS)$ is the transformation's average improvement, $S_n^2(\tau|PS)$ 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 $Q(\alpha)$ is a parametric function that models the discrepancy between the true and the estimated incremental utility:

$$Q(\alpha) := x \quad \text{such that} \quad \int_x^\infty \left(\frac{1}{\sqrt{2\pi}}\right) e^{-0.5y^2} dy = \frac{\alpha}{2}.$$

The function $Q(\alpha)$ makes the parametric statistical assumption that the estimated incremental utility is normally distributed about the true incremental utility. The reasonableness of this parametric model is justified by an important theorem in statistics, the Central Limit Theorem [38, p. 192]. This theorem states that regardless of the distribution of a random variable, the average of these values will tend to be normally distributed about the true mean of the distribution. This theorem demonstrates that even if the underlying distribution is non-normal, it can be accurately approximated as normal for the purposes of estimating expected utility. Although the Central Limit Theorem strictly holds only as the sample size tends to infinity, extensive experience in real-world problems has demonstrated the effectiveness of this "normal approximation" even with small sample sizes (see [39, Section 5.3]).

The choice of minimum sample size, n_0, relates to the definition of $Q(\alpha)$ and is discussed in Appendix A. By default we use a sample size of fifteen which has worked well in our empirical investigations.

3.2.4. The COMPOSER algorithm

COMPOSER is illustrated in Fig. 4. After each problem solving attempt, COMPOSER updates its statistics and evaluates the stopping rule for each candidate transformation. If the stopping rules are satisfied for one or more transformations with *positive* incremental utility, COMPOSER adopts the transformation with highest incremental utility, and invokes the transformation generator to obtain a new set of candidate transformations. If instead, stopping rules are satisfied for transformations with *negative* incremental utility, these are eliminated from further consideration (note that eliminating a candidate does not affect the current problem solver, so the statistics associated with the remaining candidates are unaffected). This cycle repeats until the training set is exhausted. Each time a transformation is adopted the expected utility of the resulting problem solver is higher than its predecessor, giving COMPOSER an *anytime behavior* [7].

4. Utility problem in the PRODIGY problem solver

This section describes one of two extensive evaluations we have performed with the COMPOSER system. We describe an application of COMPOSER to learning search

COMPOSER($PS_{old}, TG(\cdot), \delta$, examples, n_0)

1. $PS := PS_{old}$; $T := TG(PS)$; $n := 0$; $i := 0$; $\alpha := Bound(\delta, |T|)$;
2. **While** $T \neq \emptyset$ and $i <$ |examples| do
   /* Hill-climb as long as there is data and possible transformations */
3. **Repeat** /* Until enough data taken to identify next step */
4. $n := n + 1$; $i := i + 1$; step-taken := FALSE;
5. $\forall \tau \in T$: Get $\Delta U_i(\tau|PS)$
   /* Observe incremental utility values for ith Problem */
6. *significant* := $\{\tau \in T$: $n \geqslant n_0$ and
   $(S_n^2(\tau|PS))/[\overline{\Delta U}_n(\tau|PS)]^2 < n/[Q(\alpha)]^2\}$
   /* Collect all transformations that have reached statistical significance */
7. **If** $\exists \tau \in$ *significant*: $\overline{\Delta U}_n(\tau|PS) > 0$ **Then**
   /* Adopt τ that most increases expected utility */
8. $PS = Apply(x \in$ *significant*:
   $\forall y \in$ *significant* $[\overline{\Delta U}_n(x|PS) > \overline{\Delta U}_n(y|PS)], PS)$
9. $T := TG(PS)$; $n := 0$; $\alpha := Bound(\delta, |T|)$; step-taken := TRUE;
10. **Else** $T := T - \{\tau \in$ *significant*: $\overline{\Delta U}_n(\tau|PS) < 0\}$
    /* Discard transformations that lower expected utility */
11. **Until** step-taken or $T = \emptyset$ or $i =$ |examples|;
12. **Return** *PS*

$Bound(\delta, |T|) := \delta/|T|$, $Q(\alpha) := x$ where $\int_x^\infty (1/\sqrt{2\pi})e^{-0.5y^2}dy = \alpha/2$

Fig. 4. The COMPOSER algorithm.

control strategies for the PRODIGY planning system [53]. PRODIGY is a well studied planning system that has served as the basis for several learning investigations [15,43, 53] and has attained the status of a benchmark for learning systems. COMPOSER has also been successfully applied to the problem of learning heuristic control strategies for a NASA scheduling domain, which is described elsewhere [26].

A main goal of this evaluation is to contrast COMPOSER's approach with several other methods for addressing the utility problem, including methods developed explicitly for the PRODIGY system as well as a more general statistical approach similar to COMPOSER. We would like to compare the theoretical basis for these alternatives rather than implementation details. Therefore, we take pains to provide a fair comparison by minimizing the differences between the systems. Each method is re-implemented within the context of the PRODIGY problem solving system and each method is constrained to use PRODIGY's explanation based learning approach (discussed below) as the source of learned transformations.

4.1. 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 based on the STRIPS planner [18] 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 (where a node contains a conjunction of goals, some of which may already be achieved),
(2) choosing an unachieved goal at that node,
(3) choosing an operator that possibly achieves the goal, and
(4) choosing a binding list for the operator.

PRODIGY implements a default control method for each of these decisions and these methods may be modified by the introduction of heuristic knowledge called *control rules*. Control rules are described in Section 6.2.

4.1.1. Problem distributions

We evaluate COMPOSER's ability to identify effective modifications to PRODIGY on three different domain theories. The STRIPS domain was reported in [53]. It is a problem of a robot moving boxes through interconnected rooms with lockable doors. The AB-WORLD domain was reported in [16]. It is a variant of the standard blocksworld domain, designed to highlight deficiencies in Minton's PRODIGY/EBL approach. The BIN-WORLD domain was introduced in [30] 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 are constructed with the problem generators provided with PRODIGY 2.0. Following the methodology in [53], the set of problems was biased by filtering out problems that were judged too difficult or too easy; problems were excluded if the default PRODIGY control strategy required less than 1 CPU second or more than 100 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 B. A more detailed description of the experiments appears in [24].

4.1.2. Expected utility

We follow the established PRODIGY methodology of measuring problem solving performance by the cumulative time in CPU seconds to solve a set of problems (e.g. [16, 53]). Under the decision theoretic interpretation of the evaluation criterion, this is captured as a utility function based on the time required to solve a problem. In particular, we let the utility of the problem solver over a problem be the negative of the CPU time required to solve the problem. Maximizing expected utility therefore translates into "improving" the average time required to solve a problem.

4.2. Transformation generator

Minton introduced a technique for generating atomic modifications of the PRODIGY control strategy. The approach uses explanation based learning (EBL) [12, 54] to construct atomic search control rules based on traces of problem solving behavior and a theory of the problem solver. Sets of control rules can be associated with any of the four control points. The search control rules are condition–action statements which alter the way PRODIGY explores the space of possible plans.

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

Fig. 5. An example of a control rule.

PRODIGY/EBL generates rejection and selection control rules which are guaranteed sound in that they do not prune valid solutions from the search tree.[6] In particular, to acquire a *rejection* control rule, PRODIGY must show that a particular decision cannot even in principle lead to a successful outcome. PRODIGY performs a full tree search of all viable options. If no success node is encountered in this exhaustive search, PRODIGY re-expresses the conditions that describe the failure in a way that can be tested at the node itself. Under these conditions, there is no point in conducting the search since it is doomed to fail. The rejection rule prunes this decision for future searches. Likewise, a *selection* control rule can be acquired only if all ways but one of resolving the decision lead to failures. If resources are exhausted before a full search has been completed, then no rules can be learned from this portion of the tree. This is a limitation of the approach but fortunately, even in recursive domains, there are usually enough fully explored sub-trees to learn useful control rules. An example of an operator selection rule is shown in Fig. 5. Under the conditions of its antecedent, all operators except UNSTACK necessarily lead to failures.

While control rules are sound, they may not increase the efficiency of planning. All control rules avoid search in the plan space, but 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. Unfortunately, while it performs quite well on some domains, PRODIGY/EBL has since been shown to have undesirable properties. Etzioni illustrated how seemingly innocuous changes to a domain theory result in degraded problem solving performance [15]. 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 [29]). COMPOSER's statistically sound utility estimation procedure corrects these problems and should result in a more effective learning algorithm.

[6] The EBL unit used in PRODIGY/EBL can produce three control rule types: rejection, selection, and preference rules. Preference rules, in fact, can lead to incorrect action selection. 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 [29]. We disabled the learning of preference rules in this implementation because it enabled a more efficient means of gathering incremental utility data points (see Appendix A).

For this evaluation, the EBL component of PRODIGY serves as the transformation generator. The EBL component analyzes a trace of each solution attempt and conjectures new control rules. Each of these control rules serves as an atomic transformation to the current search control strategy. To be more consistent with its use in the PRODIGY system, our actual use of this transformation generator differs somewhat from its normal usage in COMPOSER (Fig. 4). Rather than forcing transformations to be conjectured all at once (as in lines 1 and 3 of the algorithm), transformations are potentially added to the set T after each problem solving event. Whenever a transformation is adopted (line 8 of the algorithm), all previously conjectured transformations are carried forward to the new transformation set, though their statistics are discarded.

4.3. COMPOSER implementation details

We used COMPOSER to construct an adaptive problem solver for the three applications. We call the resulting implementation COMPOSER/PRODIGY. PRODIGY with no control rules acts as the initial problem solver. Minton's EBL learning element acts as the transformation generator. 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 with positive incremental utility.

Several properties of this application allowed us to tailor COMPOSER to achieve greater statistical efficiency. 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 and have been added into the control strategy of the PRODIGY planner. Candidate rules are those that have been proposed by the EBL technique, but not yet validated. When solving a problem, candidate rules are checked, their precondition cost and the search paths they would eliminate are recorded, but the search paths are not actually eliminated. 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. [7]

4.4. Evaluation

We evaluated COMPOSER/PRODIGY's performance against four other proposed criteria for addressing the utility problem:

[7] Appendix A describes how for some applications the *Bound* function can be modified to improve the efficiency of utility analysis. For this implementation we used Eqs. (A.2) and (A.3) of the appendix to define a variant of the default *Bound*. As stated in Section A.1.1, this allows transformations to be added to T at any time in the utility analysis. Due to the relatively low variability of control rules, we used a minimum $n_0 = 3$.

(1) the heuristic utility analysis of PRODIGY/EBL [53],
(2) the non-recursive hypothesis of STATIC [14],
(3) a hybrid of PRODIGY/EBL and STATIC suggested by Etzioni [14] to overcome limitations of the two systems, and
(4) PALO [35], a statistical approach similar to COMPOSER but based on a more conservative statistical model.

Before discussing the experiments we review these techniques. For the evaluations we tried to minimize differences between the systems. All systems are implemented within the PRODIGY problem solving framework and use the same transformation generator.

4.4.1. PRODIGY/EBL's utility analysis

This technique, developed by Minton for use in PRODIGY/EBL, 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—estimates are gathered as if there were no interactions.

4.4.2. STATIC's non-recursive hypothesis

STATIC utilizes a control rule selection criterion based on Etzioni's structural theory of utility. The criterion is grounded in the *non-recursive hypothesis*. This states that "EBL is effective when it is able to curtail search via non-recursive explanations" [14, 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 non-recursive 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 on average). 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 [47,68].

For Etzioni, the explanation of a control rule is recursive if any proposition in its proof tree contains another proposition with the same predicate name. For example, suppose that executing the PICKUP operator has no chance of leading to a solution in some blocksworld problem. PRODIGY would acquire a rejection control rule stating that the PICKUP operator should be pruned from consideration in any future situation that matches the conditions of this one. The precise conditions are constructed by generalizing why PICKUP necessarily fails in this example. Now suppose that the example's explanation includes a requirement of "CLEAR(A)" which is in turn supported through some inferences by "CLEAR(B)". This explanation is recursive. According to a strong

reading of the non-recursive hypothesis, the associated control rule could not improve the overall utility.

STATIC outperforms PRODIGY/EBL's on several domains. The non-recursive hypothesis is cited as a principal reason for this success [15].[8] This claim is difficult to evaluate as the two algorithms use different rule generators. Different vocabularies are also employed to construct their respective 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 non-recursive hypothesis within the PRODIGY/EBL framework. NONREC replaces PRODIGY/EBL's empirical utility analysis with a syntactic criterion which only adopts non-recursive control rules. This acts as a filter, only allowing PRODIGY/EBL to generate non-recursive rules. All rules that satisfy the non-recursive criterion are incorporated into the final problem solver.

4.4.3. A composite algorithm

Etzioni suggests that the strengths of STATIC and PRODIGY/EBL can be combined into a single approach [14]. He proposed a hybrid algorithm which embodies several advancements including a two layered utility criterion. The non-recursive hypothesis acts as an initial filter, but the remaining non-recursive 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 non-recursive hypothesis. The remaining rules undergo utility analysis as in PRODIGY/EBL.

4.4.4. PALO's Chernoff bounds

Greiner and Cohen have proposed an approach similar to COMPOSER's [33]. 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 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 the sample mean and true mean of a distribution. As a result, PALO provides stronger bounds on statistical error but 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.[9] Our PALO-RI algorithm evaluates this approach. Like COMPOSER, 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:

[8] 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 [17].

[9] In addition to the conservative stopping rule, PALO adopts the conservative definition of *Bound* that follows from adopting Eqs. (A.1) and (A.5), whereas COMPOSER adopts Eqs. (A.1) and (A.3).

$$n \cdot \overline{\Delta U}_n(\tau|PS) > \Lambda_\tau \sqrt{2n \ln \left(\frac{T_{\max}(h+1)^2 \pi^2}{3\delta} \right)}$$

where $\overline{\Delta U}_n(\tau|PS)$ is the estimated incremental utility of transformation τ, $T_{\max}$ is the size of the largest possible candidate set, h is the number of transformations added to the current transformation sequence (the number of steps taken in the hill-climbing search), and Λ_τ is the size of the range of incremental utility values for a given transformation:

$$\Lambda_\tau = \max_i \{\Delta U_i(\tau|PS)\} - \min_i \{\Delta U_i(\tau|PS)\}.$$

A disadvantage of the technique is that its sample complexity is strongly effected by the setting of Λ_τ, a parameter whose true value is typically unavailable in advance of learning (as its depends on the maximum and minimum performance improvement possible by a given transformation). To use this method, one must be able to bound the range of incremental utility values as tightly as possible without underestimating the true range. In the context of PRODIGY, one can set an upper bound by noting that all problems are restricted to be solvable within a resource bound of 100 CPU seconds—the most a transformation could help (or hurt) is 100 seconds. This seemed too conservative for our purposes. Rather, we bound Λ_τ by the maximum time PRODIGY actually requires to solve problems from the training set before learning. Using a tighter bound would require some detailed knowledge of how much transformations are expected to help; knowledge that was not provided to the other learning systems and thus seems unfair to provide to PALO-RI.

PALO-RI uses the same method as COMPOSER/PRODIGY to obtain incremental utility statistics. We discuss the setting of the system's various parameters in the next section.

One advantage of PALO not included in PALO-RI, is that it incorporates an additional test that terminates sampling if the incremental utility of all transformations is recognized to fall below a pre-specified threshold. In contrast, COMPOSER may expend considerable data when the best transformation leads to a negligible improvement in performance (see Section 5). Methods which incorporate this additional "don't care" parameter are referred to as *indifference zone* methods [1]. Our more recent work incorporates an indifference zone into a COMPOSER-like method [25]. That paper also includes a detailed discussion of the tradeoffs imposed by such approaches.

4.4.5. Experimental procedure

We investigated the STRIPS domain from [53], the AB-WORLD domain from [14] for which PRODIGY/EBL produced harmful strategies, and the BIN-WORLD domain from [30] which yielded detrimental results for both STATIC's and PRODIGY/EBL's learning criteria. Results are summarized in Fig. 6. In each domain, the algorithms we The problem distributions were constructed using the random problem generator provided with the PRODIGY architecture. The current control rule set was saved after

every twenty training examples.[10] The independent measure for the experiments is the number of training examples and the dependent measure is the execution time in CPU seconds over 100 test problems drawn from the same distribution. This process was repeated eight times using distinct training and test sets constructed from the same problem generator. All results reported are the average of these eight trials (rounded to the nearest whole number). Fig. 6 shows the comparison graphs along with the number of rules learned by the algorithm, the number of seconds required to process the 100 training examples, and the number of seconds required to generate solutions for the 100 test problems. COMPOSER and PALO-RI require an error parameter δ which is set at 10% for the experimental runs. 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.[11]

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, the candidate set grew large since harmful rules were not discarded as quickly as in COMPOSER/PRODIGY. 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.

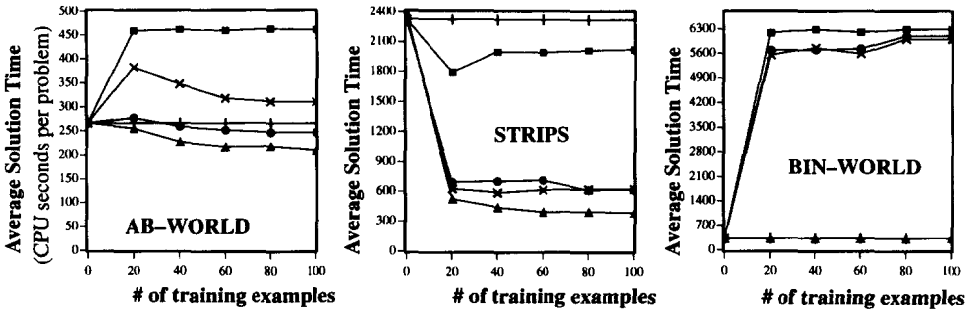
4.5. Analysis

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 (containing fewer control rules). In AB-WORLD and STRIPS, COMPOSER/PRODIGY identified beneficial control strategies. In BIN-WORLD, the algorithm did not adopt any transformations. In fact, 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 utility problem rather than differences in the vocabulary of transformations.

We expected COMPOSER/PRODIGY to have higher learning times than PRODIGY/EBL or NONREC due to its more rigorous in their assessment of incremental utility. Surprisingly, the learning times were not much higher than the non-statistical approaches and COMPOSER/PRODIGY actually learned more quickly on BIN-WORLD where it quickly discarded a control rule with high match cost which PRODIGY/EBL, NONREC,

[10] 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 [53].

[11] 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: $C = 5$, $\lambda = 150$.



ALGORITHM	AB–WORLD			STRIPS			BIN–WORLD		
	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

Fig. 6. Results for the PRODIGY application. Learning curves show performance as a function of the number of training examples.

and NONREC+UA retained. As expected, PALO-RI's learning times were significantly higher than the other systems.

The results cast doubt on the efficacy of the non-recursive hypothesis. NONREC yielded the worst performance on all domains. In conjunction with utility analysis, the results were mixed—performing adequately in AB-WORLD but 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 non-recursive, but many non-recursive rules were also detrimental. In BIN-WORLD, non-recursive rules produced substantial performance degradation. Thus, while non-recursiveness may be an important property, alone it 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 non-recursive hypothesis cannot completely account for this difference. We attribute the remaining difference to the fact that STATIC and NONREC entertain somewhat different sets of control rules. In our experiments, we constrained NONREC to use the rule vocabulary which was available to PRODIGY/EBL while in Etzioni's experiments STATIC entertained a somewhat different space of rules. This conjecture was recently supported by Minton and Etzioni [17].

Finally, although PALO-RI did not improve performance within the 100 training examples, if given sufficient examples it would likely outperform the other approaches. This is because the large sample sizes required by PALO allow utility estimates to converge closely to their true values before any transformation is selected. Thus, the transformations with the higher incremental utility tend to be selected first. (In fact, the initial selection of better control rules allowed PALO-RI to exceed COMPOSER/PRODIGY's performance in AB-WORLD given extended examples.) In contrast, COMPOSER assesses both incremental utility and the variance of utility values when determining when a transformation has reached significance. This results in COMPOSER recognizing low variance transformations more quickly. Unfortunately the cost of PALO's performance improvement is very high, both in terms of examples and learning time. While COMPOSER may identify somewhat less beneficial strategies, it achieves much faster convergence.

5. Complexity results

We now turn to an analysis of the COMPOSER algorithm. Given our emphasis on the practical aspects of the algorithm, our analysis will not consider worst-case behavior. Rather, we provide *average case* complexity results for the algorithm for both *sample complexity*, the number of examples required to make a statistical inference, and run time complexity of the algorithm. We focus on the number of examples and the amount of work required to perform a single step in the hill-climbing search. Obviously, the total complexity will depend on the number of hill-climbing steps taken, but this is domain specific.

The number of examples taken at a hill-climbing step also depends on domain specific factors, but these can be related to complexity in meaningful ways. For example, the amount of work at a step depends on the number of candidate transformations at that step and we can specify the exact function relationship between work and the number of transformations. With such knowledge, a user of COMPOSER can assess how best to organize the transformation generator for a specific learning problem.

5.1. Properties of the Nádas stopping rule

The properties of COMPOSER follow from the properties of its method for statistical inference. Therefore, we first consider the characteristics of the Nádas stopping rule. Given some transformation τ, and an error level α, this stopping rule determines how many examples are sufficient to show that the incremental utility of τ is positive (negative) with probability $1 - \alpha$. The characteristics of this stopping rule have been proven

by Nádas in [60]. The proofs are technical but we will restate the results and give an intuitive explanation of why they hold.

COMPOSER takes examples until the following inequality holds (Eq. (4)):

$$n \geqslant n_0 \quad \text{and} \quad \frac{S_n^2}{\left[\overline{\Delta U_n}\right]^2} \leqslant \frac{n}{Q(\alpha)^2}$$

where n is the number of examples taken so far, $\overline{\Delta U_n}$ is the transformation's average improvement, S_n^2 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 a minimum sample size, and $Q(\alpha)$ is the function that models the discrepancy between the true and estimated incremental utility:

$$Q(\alpha) = x \quad \text{such that} \int_x^\infty \left(\frac{1}{\sqrt{2\pi}}\right) e^{-0.5y^2} dy = \frac{\alpha}{2}.$$

For a given sequence of training examples the stopping rule will be satisfied after some number of examples, called the *stopping time* in the sequential statistical literature. (The term "stopping time" is somewhat misleading as it refers to the number of examples, not the temporal duration of the procedure. We retain the term, however, to be consistent with the statistical literature.) The stopping time, ST, is a random variable. The sample complexity of the stopping rule is characterized by the expected value of the stopping time: $E[ST]$. This is the average number of examples required to make a decision. From the results of Nádas we can derive the following relationship for the expected stopping time.

Theorem 1. *Let ST be the stopping time associated with the Nádas stopping rule for a given transformation. Let α be the requested error level, σ^2 be the actual variance of the distribution associated with the transformation and μ be the actual incremental utility. Then*:

(1) *For small $1/\alpha$ the expected stopping time is governed by the following relationship*:

$$E[ST] \leqslant \frac{2}{\alpha^2 \pi} \cdot \frac{\sigma^2}{\mu^2}.$$

(2) *For large $1/\alpha$ the expected stopping time is governed by the following relationship*:

$$E[ST] \approx \ln(1/\alpha) \frac{\sigma^2}{\mu^2}.$$

This result states that the stopping time is determined by the error level parameter, which is under control of the user, and two fixed but unknown constants, σ^2 and μ, which are properties of the inference problem. The average stopping time associated with a particular transformation is bounded by a quadratic in $1/\alpha$ (or to log of $1/\alpha$

for large $1/\alpha$), linearly with the variance of the transformation, and quadratically with the inverse of its mean. This makes intuitive sense: the greater the required confidence, the more difficult it is to bound the mean, the greater the variance in incremental utility values, the more difficult it is to bound its mean, and the closer the incremental utility is to zero, the more difficult it is to show that the transformation is better (worse) than the default strategy.

Proof. A non-closed form equation for the stopping time is derived by Nádas in [60]. The theorem follows from this proof and other results (see [24, Appendix B]). Although the complete proof is too lengthy to include here, it is easy to provide intuition on why a result like Theorem 1 should hold. The Central Limit Theorem states that the normalized difference between the true incremental utility and the sample incremental utility will be (approximately) normally distributed. Using this observation it is easy to compute a confidence interval around the mean, given a sample of n observations. Any introductory statistics book shows (with a suitable mapping of notations) that:

$$\Pr\left[\overline{\Delta U}_n - Q(\alpha)\sqrt{\frac{S_n^2}{n}} \leqslant \mu \leqslant \overline{\Delta U}_n + Q(\alpha)\sqrt{\frac{S_n^2}{n}}\right] = 1 - \alpha$$

or in other words, with probability $1-\alpha$, the true incremental utility of a transformation, μ, lies within the interval $\overline{\Delta U}_n \pm Q(\alpha)\sqrt{S_n^2/n}$, where S_n^2 is the variance of our n samples. The Nádas stopping rule is designed to take examples until the size of this confidence interval is twice the size of the unknown mean, μ. Formally:

$$Q(\alpha)\sqrt{\frac{S_n^2}{n}} = \mu.$$

Solving this relationship for n we get the following relationship:

$$n = Q(\alpha)^2 \frac{S_n^2}{\mu^2}.$$

This shows that n is the number of examples that will produce a confidence interval of the appropriate size. Or stated differently, n should be the stopping time, which is the result of Theorem 1.

It then remains to show how $Q(\alpha)$ grows as a function of $1/\alpha$. That $Q(\alpha$ is bounded by a linear function follows from *Markov's inequality*. More precisely, this shows $Q(\alpha) \leqslant \sqrt{2/\pi} \times 1/\alpha$. For large $1/\alpha$, $Q(\alpha)$ converges to about $\sqrt{\ln(1/\alpha)}$. This can be obtained using the asymptotic expansion of the standard normal distribution. Both derivations are given in [24]. □

5.2. Sample complexity

The sample complexity is the number of examples required to perform statistical inference (which is equivalent to the largest stopping time at a step). 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 each step in the hill-climbing search in terms of several parameters.

Within a particular hill-climbing step there is some set of transformations T. Recall that α is the allowable statistical error associated with an incremental utility estimate for each transformation in T. The value α is a bound on the probability that a transformation with negative incremental utility is perceived as positive or *vice versa*. Let the value δ be total error bound. The error α is related to δ by the *Bound* function and by default, $\alpha = \delta/|T|$.

As validation proceeds within the step, COMPOSER consumes training examples, dynamically computing estimates for transformations in T. One of three cases must arise: (1) all transformations are shown to have negative incremental utility and are discarded; (2) some transformation with positive incremental utility is identified and adopted; or (3) the training set is exhausted. In the first case, the expected number of examples is equivalent to the maximum stopping time of the transformations in T; in the second, the expected number of examples is equivalent to the stopping time of the adopted transformation. The following theorem describes the relationship that governs the sample complexity, given that sufficient data has been provided for COMPOSER to make a decision:

Theorem 2. *Let ST^* be the number of examples consumed at a step in COMPOSER's hill-climbing search under the default settings Eqs. (A.1) and (A.3)), where δ is the error bound and T is the set of transformations at that step. Then:*

(1) *For small $|T|/\delta$ the expected sample complexity is bounded by a polynomial in T and $1/\delta$:*

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

(2) *For large $|T|/\delta$ the expected sample complexity is governed by:*

$$E[ST^*] \approx c \cdot \ln\left(\frac{|T|}{\delta}\right)$$

where c is a constant whose value depends on the expected incremental utility and variance in incremental utility values for the transformations in T.

Proof. This follows directly from Theorem 1 and the default definition of $\alpha = \delta/|T|$. The constant c is the expected value of σ_i^2/μ_i^2 where i is the last transformation that is adopted/rejected at a step.

Thus, the expected number of examples required at a step grows at most quadratically in $|T|$ 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. Similarly, the sample complexity will increase polynomially as we require greater statistical confidence. □

5.3. Run time complexity

The expected run time of the algorithm depends on the number of examples used by the algorithm and the cost to process each example which 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, so the cost of processing is tied to the complexity of the problem solver.

Theorem 3. *Let R be an upper bound on the cost of solving a problem. Then:*

(1) *For small $|T|/\delta$ the expected run time complexity is:*

$$O\left(R\frac{|T|^2}{\delta^2}\right).$$

(2) *For large $|T|/\delta$ the expected run time complexity is:*

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

Proof. Where $|T|/\delta$ is small, 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 solution attempt can cost at most R leading to a maximum cost of $R|T|$ to process each example. The analogous argument holds for large $|T|/\delta$.

Therefore, expected cost of a hill-climbing step grows linearly in R, at most quadratically in the required confidence, $1/\delta$, and at most cubically in the number of transformations at that step, $|T|$. □

5.4. Discussion

Theorem 1 has some interesting consequence for the COMPOSER's performance. 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, COMPOSER does not necessarily perform steepest ascent hill climbing. This was observed in the comparison with PALO-RI.

A problem arises when all of the transformations in T have near-zero incremental utility. COMPOSER does not terminate until some transformation has been accepted

or all have been rejected. As the incremental utility of transformations tends to zero, however, the sample complexity increases dramatically. Although it is unlikely that every transformation has near-zero utility, if this occurs, the algorithm may not be able to make a decision. Rather, the algorithm will simply exhaust all of its training examples with out making any improvements in expected utility. 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 Section 6.

6. Limitations and future work

COMPOSER provides a probabilistic solution to the utility problem and has demonstrated its practicality in the problem solving applications described herein, and in two other implementations reported elsewhere [26,27]. While these successes are encouraging, it is important to realize that COMPOSER embodies many design commitments that restrict its generality. In this section we discuss these limitations and possible extensions to the approach. We first characterize some conditions that are necessary for successful results. We then consider limitations and extensions to three aspects of the utility problem: organizing the search through the space of modifications, estimating utility of transformations, and gathering statistics. COMPOSER embodies a particular set of commitments for each of these aspects and thus can be seen as one point in a large space of possible commitments (see also [31]).

6.1. Applicability conditions

We can summarize three basic conditions that COMPOSER requires for satisfactory results.

6.1.1. A 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 of search steps, with a relatively small branching factor.

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.

For our experiments, a problem solver is transformed by the addition of a single control rule to its existing set of control rules. One might imagine entertaining as a single transformation one which adopts two or three control rules at a time. This would ameliorate some of the problems of hill climbing. However, COMPOSER would probably not work effectively with transformations that allowed as many as ten or twenty control rules to act as a unit. The cost of actually performing the transformation to generate the new problem solver should also be small. Transformations such as

changing a spatial reasoner from a surface representation to a volume representation would not be likely candidates for the COMPOSER approach.

6.1.2. *Availability 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 sufficiently large body of training problems.

A task distribution that cycles through a set of different sub-patterns, or one that shifts slowly over time presents a difficulty to the COMPOSER approach. Shifts in the distribution violate the fixed distribution assumption. However there are approaches for partially overcoming these difficulties. With quickly shifting but cyclic patterns, it may suffice to average over the cycle. One might draw problems throughout the extent of the cycle and then randomize the problems to destroy the systematic changes. Training on this randomized distribution will result in a problem solver that does not take advantage of the shifts, but will nonetheless improve average performance. When there are slow steady shifts in the distribution one can take windows of training problems, or use other methods to periodically re-train the problem solver as the distribution shifts [49]. Tracking and taking advantage of predicted shifts is an important area of future research.

6.1.3. *Low problem solving cost*

Extracting incremental utility statistics by solving training examples under various transformations is only feasible if problems can be solved with a sufficiently low cost. This is perhaps the strongest limitation of the technique. It may not be feasible to use COMPOSER to improve, for example, an average case exponential time problem solver. One real-world domain to which COMPOSER has been applied is the problem of scheduling communications between earth orbiting satellites and ground based antennas [26]. This application demonstrated the necessity of reducing learning cost. COMPOSER's modeling assumptions helped dramatically in managing this complexity but we were forced to make some additional innovations to maintain reasonable efficiency. Potential ways of relaxing this reliance on tractable initial problem solving are discussed below.

6.2. *Organizing search*

A key challenge is successfully navigating through the vast space of possible composite modifications. COMPOSER's hill-climbing restriction attempts to ensure efficient search. In many ways, this restriction is too strong. When there are strong interactions between transformations, COMPOSER may find poor local maxima, or no solution 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 vital to focus the search on the most promising alternatives first. COMPOSER follows a generate and test paradigm. Performance can be improved by making generation as intelligent as possible. PRODIGY's learning component achieves some measure of intelligent generation through its EBL component. This carefully analyzes each problem solving trace for search inefficiencies and only propose sound transformations that

address observed deficiencies. Where applicable, heuristics like Etzioni's non-recursive hypothesis may also help filter out unpromising transformations.

Finally, we are investigating how to apply notions from work on bounded rationality [13,41] and *bandit problems* [20,21,42] to help control the cost of identifying good adaptations. These methods balance the improvement due to reasoning with the cost of achieving those improvements. Currently, COMPOSER identifies transformations with high incremental utility and the efficiency of search is ensured by imposing a bias on the exploration—the actual search behavior of the algorithm arises from the interaction between 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 an improvement. For some results in this area see [25,28,32].

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 precise shape of the distribution and 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 grows only as the 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, this includes 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 weak method 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 an independent entity, even though 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, although the sample complexity only grows by the log of the number of transformations. 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 [45,69]. While not always possible, it is an important area of future research.

Several statistical methods can be applied to further improve the efficiency of the estimation process. For example, currently a transformation is eliminated only if the resulting strategy is significantly worse than the default control strategy. However, given that other transformations may be better than the default, transformations could be more quickly eliminated if they are compared with the most promising transformation rather than the default control strategy. We investigate this and other extensions to COMPOSER in [6]. Similar strategies for improving the statistical inference appear in [50,58].

Heuristics and prior information can replace or augment statistical estimates. Syntactic measures like the *operationality criteria* of Mitchell, Keller, and Kedar-Cabelli [54] can be seen as approximate binary measures of incremental utility. Unfortunately, syntactic measures have difficulty capturing the distributional nature of incremental utility. Instead, we are investigating 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 may be useful as a bias on later 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 with the original and intermediate problem solvers. While this dependence on problem solving is a serious limitation, in some applications it can be overcome or mitigated. For example, in the problem solving domains discussed in this article, we were able to take advantage of properties of the transformations to process examples more efficiently. In general, some transformation vocabularies may be easier to implement within the COMPOSER framework than others. Perhaps the issue can be resolved by identifying hybrid statistical/analytic means to estimate 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 [35] 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. [53,61]) or by observing a teacher (e.g. [12,55,70]).

7. Conclusion

This article has argued that 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 an application and satisfy the oft argued need for declarative and modular knowledge representation. On the other hand, special purpose approaches are best suited to the 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 step toward reconciling the conflicting needs of generality and efficiency.

In this article we have developed a formal characterization of the utility problem which 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 for an application. COMPOSER is still a heuristic approach, but by casting it within a statistically sound framework we are able to 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.

A larger theme is that any learning algorithm must strike a balance between maximizing performance yet doing so efficiently. COMPOSER embodies numerous commitments to achieve efficient learning performance. We have argued that effective learning is composed of essentially three basic and roughly independent problems. First, there is the problem of searching the space of possible composite modifications. Second, there is the issue of obtaining estimates of local properties of transformations across the pattern of task, in our case estimating incremental utility of atomic transformations. Finally, there is the issue of efficiently gathering the information to produce these estimates. 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 proceed simply by the development of large techniques like COMPOSER or PRODIGY, but by the development of smaller, well understood, methods that may be combined in a variety of ways to produce a complete algorithm.

Acknowledgements

Many thanks are due to Steve Minton, who's PRODIGY system inspired this work. Russ Greiner contributed greatly through many technical discussions. Adam Martinsek assisted us on numerous statistical issues. Thanks also to Marsha Brofka, Dan Oblinger, Oren Etzioni, and the anonymous reverers for critical comments on earlier versions of this work. This research was conducted primarily at the Beckman Institute at the University of Illiois with support by the National Science Foundation under grants NSF-IRI-87-19766 and NSF-IRI-92-09394.

Appendix A. Implementation tradeoffs

Our motivation in designing COMPOSER was not simply to provide a statistically sound learning technique, but to provide a practical tool. A chief drawback of COMPOSER's statistical approach is that it can be expensive. This section discusses pragmatic issues and techniques for improving COMPOSER's performance by tailoring it to the specific characteristics of an application.

The principal impediment to COMPOSER's approach to the utility problem is managing the computational expense of identifying good transformations. To maximize COMPOSER's performance we would like to efficiently process each example and to consume few examples to perform its statistical inferences. When applying COMPOSER to a particular application, we see three ways in which this expense can be mitigated:

(1) Tailoring of the *Bound* function.

(2) Tailoring of the discrepancy modeling function $Q(\alpha)$.

(3) Tailoring of methods for gathering incremental utility statistics.

This appendix describes the rationale behind the standard configuration of COMPOSER and describes alternative approaches. This tailoring allows the application designer to take advantage of any domain specific knowledge to improve learning efficiency.

A.1. *Tailoring Bound*$(\delta, |T|)$

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. Here we consider a more general definition of *Bound* that includes the current step in the hill-climbing search: *Bound*$(\delta, i, |T|)$. 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 τ of estimates, and the cumulative error over steps 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 acceptable error in step i.

A.1.1. *Error in a step*

On step i, we are investigating a set T of transformations. Given that the error of each of the estimates is α, the expected total error for the step, β_i, is bounded below by α and above by $|T|\alpha$. (The upper bound follows from Bonferroni's inequality which states that the probability of a joint event is less than or equal to the sum of probabilities of each event [37, p. 363].) That the step error may be greater than α is most clearly seen in the situation 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 incorrectly 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, meaning β_i is a function of the size of T. The true relationship depends on the covariance between the distribution of incremental utility data points associated with each transformation, though, this information is generally unavailable.

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

$$\text{Worst case:} \quad \alpha := \beta_i/|T| \text{ (default)}, \tag{A.1}$$

$$\text{best case:} \quad \alpha := \beta_i. \tag{A.2}$$

In the worst case, error grows linearly in the size of the number of transformations, for example when every transformation has negative incremental utility and is negatively correlated. This situation will probably never arise in practice, however, it does provide a strong guarantee that the observed statistical error will not be higher than expected. The best-case model, Eq. (A.2), assumes the error does not grow appreciably as the size of T grows. We have performed some empirical evaluations showing that this assumption can be reasonable if T is relatively small (e.g., 30). 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 generator to add new transformations into T as we are evaluating the existing members.

A.1.2. Error across steps

Let β_i denote the chance of adopting a transformation with negative incremental utility on the ith step. As the number of steps grows, so does the chance that at least one step will actually result in a decrease in expected utility. However, even if some steps reduce utility, 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 several steps forward. The worst-case approach is to limit the probability that COMPOSER will adopt *any* incorrect step to at most δ (i.e., $\sum_i \beta_i \leqslant \delta$). This guarantees that COMPOSER satisfies the error requirement, but may require many more examples than the former approach. There are different ways to implement the worst-case approach depending on whether the number of possible steps is known in advance. We have considered three methods for setting β_i in terms of the overall error parameter δ:

$$\text{Liberal bound:} \quad \beta_i := \delta \quad \text{(default)}, \tag{A.3}$$

$$\text{worst-case bound/limited steps:} \quad \beta_i := \frac{\delta}{k}, \tag{A.4}$$

$$\text{worst-case bound/unlimited steps:} \quad \beta_i := \frac{6\delta}{i^2\pi^2}. \tag{A.5}$$

The default policy, Eq. (A.3), 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. The later two equations are useful when efficiency must be sacrificed for rigor. When the number of steps can be limited in advance, one can simply divide the error evenly over each of the k steps (Eq. (A.4)). 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. Eq. (A.5) satisfies this requirement.[12]

Once the application implementor chooses a model for the error within a step and a model for the error across steps, these should be unified into an overall function

[12] This equation was suggested to us by Russell Greiner and is the basis for his PALO algorithm (see [35]).

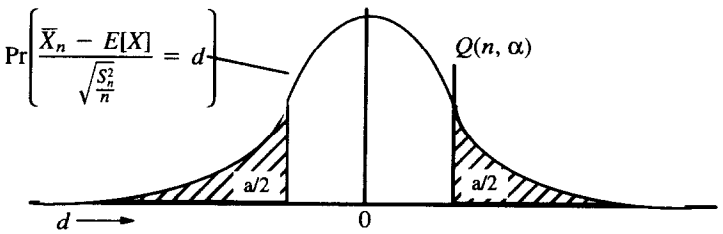


Fig. A.1. Probability distribution of the normalized difference between the sample mean and true mean of the original distribution. $Q(n, \alpha)$ is the value such that the probability of achieving a distance greater than this is less than or equal to $\alpha/2$.

Bound$(\delta, i, |T|)$ which combines the two choices into an error level for each estimate for the ith step in the hill-climbing search.

A.2. *Tailoring* $Q(\alpha)$ *and* n_0

The function $Q(\alpha)$ models the normalized expected discrepancy between the estimated incremental utility and the true incremental utility. Here we consider a more general definition, $Q(n, \alpha)$. The estimate is the average of a finite sample of n data incremental utility observations. This sample will be roughly representative of the actual distribution of data points as it is drawn randomly from the fixed distribution. However the sample mean will only approximate the true mean of the distribution. The Nádas stopping rule bounds 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 Fig. A.1. The bell shaped curve shows the likelihood of observing each normalized difference. 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 $\alpha/2$.

It is rarely possible to know the precise distribution of differences for a given learning situation. Fortunately, for *any* distribution, as n increases, the distribution of differences 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. This fact implies, under some weak conditions,[13] that function $Q(n, \alpha)$ can be approximated satisfactorily using the quantile of a standard normal distribution (see [39, Section 5.3]). 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 defining $Q(n, \alpha)$:

[13] The distribution must have positive variance and hence finite mean.

Standard normal: $Q_\Phi(n,\alpha) = x$

$$\text{such that } \int_x^\infty \left(\frac{1}{\sqrt{2\pi}}\right) e^{-0.5y^2} dy = \frac{\alpha}{2} \quad \text{(default)}, \tag{A.6}$$

$$T: \quad Q_t(n,\alpha) = x \quad \text{such that } \int_x^\infty \frac{\Gamma[(n+1)/2]}{\sqrt{\pi n}\Gamma(n/2)(1+y^2/r)^{(n+1)/2}} dy = \frac{\alpha}{2}. \tag{A.7}$$

The first is COMPOSER's default. It is based on the standard normal distribution model. The second is based on a model called the *student t* distribution. This second model is 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, higher variance in incremental utility values requires a greater n_0 to ensure the approximation model is close. Smaller values for δ require more precise modeling of the error and therefore a better approximation. Thus, the smaller the requested error level, the greater that 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 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.

A.3. Gathering statistics

The largest cost in using COMPOSER tends to be cost to obtain utility observations. Given a current problem solver *PS* and a set of transformations *T*, COMPOSER must obtain utility observations for each transformation over a large sample of problems. There are many techniques, however, that can significantly reduce this cost depending on the characteristic of the application. These techniques can reduce cost in two ways; first by reducing the number of utility values necessary to observe, and second, by reducing the cost of obtaining each utility value.

A sampling technique known as *blocking* can reduce the number of utility values necessary to make statistical decisions [4,58]. As was shown in the theoretical analysis, the number of examples needed to make statistical inferences grows with the variance in the incremental utility values. Blocking works by minimizing this variance. To understand blocking, consider the problem of finding the highest yielding variety of wheat. Wheat yield is effected by factors other than the variety of wheat, which is the factor of interest. We will call these other influences the *nuisance factors* (e.g., the weather conditions in the year the crop was grown). Often these nuisance factors have the greatest influence, washing out the contribution of the factor of interest, and thus increasing the variance in the data. A standard solution, called *randomized block design* is to combine all data with identical values on their nuisance factors into a single block, and only consider the differences in the observations within the block when computing utility values. For the wheat example, a block corresponds to a plot of land in some location. Block design

suggests having several plots of land in different locations, and planting every variety of wheat within each plot. One then only considers the difference in yield between varieties within a plot of land, and averages these differences across the different plots.

In COMPOSER, the nuisance factors are the specific characteristics of each problem drawn from the task distribution—some problems are easy, others hard, and these differences are likely to overwhelm the differences due to the choice of transformation. The solution we have adopted is to block transformations by problem. We take each problem (the block) and observe the behavior of each possible transformation on that problem. Incremental utility values are then derived by subtracting the utility of the default strategy from the utility of each transformation in turn, for that block. When the problem influences are dominant this procedure can lead to a significant reduction in the number of examples needed for statistical inference. These problem influences tend to dominate in many of the intended applications as transformations generally make relatively small incremental changes to the current problem solver, and therefore each transformed problem solver will perform similarly on similar problems. The alternative to blocking is to compute the incremental utility where each utility value is derived from a different problem. In some situations, it may be necessary to perform this strategy as it may not be possible to repeatedly solve the identical problem. Blocking is of limited benefit if problem differences are small relative to the effect of the transformations.

There are also techniques for reducing the cost of obtaining utility data. The simple strategy we recommend is to solve a given problem with each of the candidate problem solvers. The complexity of this brute-force processing is tied to the complexity of each of the $|T|$ problem solvers. In some learning situations, brute-force processing may prove too expensive. For example, one or more of the candidate problem solvers may be intractable. Furthermore, the brute-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 employ any information that could reduce this cost.

The cost of obtaining utility observations can be dramatically reduced if there is a detailed cost model that can efficiently derive the ramification of the proposed transformations without actually solving the problem (e.g. [36, 68]). With such a model we could simply draw a random training example and then use the model to determine the effect of different transformations. Such models are rarely available, but short of this, it may be possible to extract the necessary statistics to determine the effectiveness of different transformations by solely observing the normal operations of the current problem solver. Section 4 describes one such *unobtrusive* implementation that worked in PRODIGY. Sometimes it is only possible to extract partial information in this way. Greiner and Jurisica [35] propose one method for using such partial information that does not conflict with COMPOSER's assumptions and could be incorporated.

Appendix B. BIN-WORLD domain

The BIN-WORLD domain was introduced in [30] to highlight some deficiencies in both PRODIGY/EBL's utility analysis and Etzioni's non-recursive hypothesis. The

domain is a robot assembly task where the goal is to construct a composite part from a set of components (represented as achieving the state where part-assembled is true). All the components for a particular part are stored in a bin. If all of the components in the bin are free of defects, the part may be assembled. Otherwise another bin must be examined for acceptability. The INSPECT-BIN operator determines if a given bin is suitable for assembly. The ASSEMBLE-COMPONENTS operator constructs the part.

B.1. Domain theory

Assemble-components	
Preconds:	$\exists x$(parts-bin(x) $\wedge$ defect-free-components(x))
Effects:	(add part-assembled( ))

Inspec-bin	
Preconds:	$\forall y$(in-bin(y, x) $\wedge$ good(x, y))
Effects:	(add defect-free-components(x))

B.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 with the defect-free bin 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. The components are ordered with the defective component last. The bins are ordered with the defect-free bin 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. Estimating utility of a bi-modal distribution (or any distribution with high variance) from a single example results in an inaccurate representation of the true incremental utility of any learned control rule.

References

[1] R.E. Bechhofer, A single-sample multiple decision procedure for ranking means of normal populations with known variances, *Ann. Math. Stat.* **25** (1) (1954).

[2] J.O. Berger, *Statistical Decision Theory and Bayesian Analysis* (Springer, Berlin, 1980).

[3] A. Borgida and D.W. Etherington, Hierarchical knowledge bases and efficient disjunctive reasoning, in: *Proceedings First International Conference on Principles of Knowledge Representation and Reasoning*, Toronto, Ont. (1989) 33–43.

[4] H. Büringer, H. Martin and K.-H. Scriever, *Nonparametric Sequential Selection Procedures* (Birkhäuser, Boston, 1980).

[5] P. Cheeseman, B. Kanefsky and W.M. Taylor, Where the really hard problems are, in: *Proceedings IJCAI-89*, Sidney (1989) 163–169.

[6] S.A. Chien, J.M. Gratch and M.C. Burl, On the efficient allocation of resources for hypothesis evaluation in machine learning: a statistical approach, *IEEE Trans. Pattern Anal. Mach. Intell.* **17** (1995) 652–665.

[7] T. Dean and M. Boddy, An analysis of time-dependent planning, in: *Proceedings AAAI-88*, St. Paul, MN (1988) 49–54.

[8] T.L. Dean and M.P. Wellman, *Planning and Control* (Morgan Kaufmann, San Mateo, CA, 1991).

[9] R. Dechter, Constraint networks, in: S.C. Shapiro, ed., *Encyclopedia of Artificial Intelligence* (Wiley, New York, 1992).

[10] R. Dechter and J. Pearl, Network-based heuristics for constraint-satisfaction problems, *Artif. Intell.* **34** (1987) 1–38.

[11] M.H. DeGroot, *Optimal Statistical Decisions* (McGraw-Hill, New York, 1970).

[12] G.F. DeJong and R.J. Mooney, Explanation-based learning: an alternative view, *Mach. Learn.* **1** (1986) 145–176.

[13] J. Doyle, Rationality and its roles in reasoning (extended version), in: *Proceedings AAAI-90*, Boston, MA (1990) 1093–1100.

[14] O. Etzioni, A structural theory of search control, PhD thesis, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA (1990).

[15] O. Etzioni, Why Prodigy/EBL works, in: *Proceedings AAAI-90*, Boston, MA (1990) 916–922.

[16] O. Etzioni, STATIC a problem-space compiler for PRODIGY, in: *Proceedings AAAI-91*, Anaheim, CA (1991) 533–540.

[17] O. Etzioni and S. Minton, Why EBL produces overly-specific knowledge: a critique of the PRODIGY approaches, in: *Proceedings Ninth International Conference on Machine Learning*, Aberdeen (1992) 137–143.

[18] R.E. Fikes and N.J. Nilsson, STRIPS: a new approach to the application of theorem proving to problem solving, *Artif. Intell.* **2** (1971) 189–208.

[19] M. Fisher, The Lagrangian relaxation method for solving integer programming problems, *Manage. Sci.* **27** (1981) 1–18.

[20] P.W.L. Fong, A quantitative study of hypothesis selection, in: *Proceedings Twelfth International Conference on Machine Learning*, Tahoe City, CA (1995) 226–234.

[21] J.C. Gittins, *Multi-Armed Bandit Allocation Indices* (Wiley, New York, 1989).

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

[23] Z. Govindarajulu, *The Sequential Statistical Analysis* (American Sciences Press, Columbus, OH, 1981).

[24] J.M. Gratch, COMPOSER: a decision-theoretic approach to adaptive problem solving, Tech. Rept. UIUCDCS-R-93-1806, Department of Computer Science, University of Illinois, Urbana, IL (1993).

[25] J. Gratch, On efficient approaches to the utility problem in adaptive problem solving, PhD thesis, Department of Computer Science, University of Illinois at Urbana-Campaign, Urbana, IL (1995).

[26] J. Gratch and S. Chien, Learning search control knowledge for the deep space network scheduling problem, in: *Proceedings Tenth International Conference on Machine Learning*, Amherst, MA (1993).

[27] J. Gratch, S. Chien and G. DeJong, Learning search control knowledge to improve schedule quality, in: *Proceedings IJCAI93 Scheduling Workshop* (1993).

[28] J. Gratch, S. Chien and G. DeJong, Improving learning performance through rational resource allocation, in: *Proceedings AAAI-94*, Seattle, WA (1994).

[29] J. Gratch and G. DeJong, A hybrid approach to guaranteed effective control strategies, in: *Proceedings Eighth International Workshop on Machine Learning*, Evanston, IL (1991).

[30] J. Gratch and G. DeJong, COMPOSER: a probabilistic solution to the utility problem in speed-up learning, in: *Proceedings AAAI-92*, San Jose, CA (1992) 235–240.

[31] J. Gratch and G. DeJong, A framework of simplifications in learning to plan, in: *Proceedings First International Conference on Artificial Intelligence Planning Systems*, College Park, MD (1992) 78–87.

[32] J. Gratch and G. DeJong, Rational learning: a principled approach to balancing learning and action, Tech. Rept. UIUCDCS-R-93-1801, Department of Computer Science, University of Illinois, Urbana, IL (1993).

[33] R. Greiner and W.W. Cohen, Probabilistic hill-climbing, in: *Proceedings Computational Learning Theory and "Natural" Learning Systems* (1992).

[34] R. Greiner and C. Elkan, Measuring and improving the effectiveness of representations, in: *Proceedings AAAI-91*, Sidney (1991).
[35] R. Greiner and I. Jurisica, A statistical approach to solving the EBL utility problem, in: *Proceedings AAAI-92*, San Jose, CA (1992) 241–248.
[36] R. Greiner and J. Likuski, Incorporating redundant learned rules: a preliminary formal analysis of EBL, in: *Proceedings IJCAI-89*, Detroit, MI (1989) 744–749.
[37] Y. Hochberg and A.C. Tamhane, *Multiple Comparison Procedures* (Wiley, New York, 1987).
[38] R.V. Hogg and A.T. Craig, *Introduction to Mathematical Statistics* (Macmillan, New York, 1978).
[39] R.V. Hogg and E.A. Tanis, *Probability and Statistical Inference* (Macmillan, New York, 1983).
[40] L.B. Holder, Empirical analysis of the general utility problem in machine learning, in: *Proceedings AAAI-92*, San Jose, CA (1992) 249–254.
[41] E.J. Horvitz, G.F. Cooper and D.E. Heckerman, Reflection and action under scarce resources: theoretical principles and empirical study, in: *Proceedings IJCAI-89*, Detroit, MI (1989) 1121–1127.
[42] L.P. Kaelbling, *Learning in Embedded Systems* (MIT Press, Cambridge, MA, 1993).
[43] C. Knoblock, Learning hierarchies of abstraction spaces, in: *Proceedings Sixth International Conference on Machine Learning*, Ithaca, NY (1989) 241–245.
[44] R.E. Korf, Planning as search: a quantitative approach, *Artif. Intell.* **33** (1987) 65–88.
[45] P. Laird, Dynamic optimization, in: *Proceedings Ninth International Conference on Machine Learning*, Aberdeen (1992) 263–272.
[46] 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).
[47] S. Letovsky, Operationality criteria for recursive predicates, in: *Proceedings AAAI-90*, Boston, MA (1990) 936–941.
[48] N.J. Lewins, Practical solution-caching for PROLOG: an explanation-based learning approach, PhD thesis, Department of Computer Science, University of Western Australia (1993).
[49] N. Littlestone and M.K. Warmuth, The weighted majority algorithm, *Inform. Comput.* **108** (1994) 212–261.
[50] O. Maron and A.W. Moore, Hoeffding races: accelerating model selection search for classification and function approximation, in: *Advances in Neural Information Processing Systems* **6** (Morgan Kaufmann, Los Altos, CA, 1994).
[51] K. Melhorn, *Data Structures and Algorithms 1: Sorting and Searching* (Springer, Berlin, 1984).
[52] D.P. Miller, R.S. Desai, E. Gat, R. Ivlev and J. Loch, Reactive navigation through rough terrain: experimental results, in: *Proceedings AAAI-92*, San Jose, CA (1992) 823–828.
[53] S. Minton, in: *Learning Search Control Knowledge: An Explanation-Based Approach* (Kluwer Academic Publishers, Norwell, MA, 1988).
[54] T.M. Mitchell, R. Keller and S. Kedar-Cabelli, Explanation-based generalization: a unifying view, *Mach. Learn.* **1** (1986) 47–80.
[55] T.M. Mitchell, S. Mahadevan and L.I. Steinberg, LEAP: a learning apprentice for VLSI design, in: *Proceedings IJCAI-85*, Los Angeles, CA (1985) 573–580.
[56] D. Mitchell, B. Selman and H. Levesque, Hard and easy distributions of SAT problems, in: *Proceedings AAAI-92*, San Jose, CA (1992) 459–465.
[57] T.M. Mitchell, R.E. Utgoff and R. Banerji, Learning by experimentation: acquiring and refining problem-solving heuristics, in: R. Michalski, J. Carbonell and T. Mitchell, eds., *Machine Learning: An Artificial Intelligence Approach* (Morgan Kaufman, San Mateo, CA, 1983).
[58] A.W. Moore and M.S. Lee, Efficient algorithms for minimizing cross validation error, in: *Proceedings Eleventh International Conference on Machine Learning*, New Brunswick, NJ (1994).
[59] J. Mostow, Mechanical transformation of task heuristics into operational procedures, PhD thesis, Department of Computer Science, CMU, Pittsburgh, PA (1981).
[60] A. Nádas, An extension of a theorem of Chow and Robbins on sequential confidence intervals for the mean, *Ann. Math. Stat.* **40** (1969) 667–671.
[61] B.K. Natarajan, On learning from exercises, in: *Proceedings Second Annual Workshop on Computational Learning Theory*, Santa Cruz, CA (1989) 72–87.
[62] M.A. Perez and O. Etzioni, DYNAMIC: a new rule for training problems in EBL, in: *Proceedings Ninth International Conference on Machine Learning*, Aberdeen (1992) 367–372.

[63] B. Roy, Problems and methods with multiple objective functions, *Math. Programming* **1** (2) (1971).

[64] S. Russell and E. Wefald, Principles of metareasoning, in: *Proceedings First International Conference on Principles of Knowledge Representation and Reasoning*, Toronto, Ont. (1989) 400–411.

[65] A.L. Samuel, Some studies in machine learning using the game of checkers, *IBM J.* **3** (3) (1959).

[66] M. Schoppers, Building plans to monitor and exploit open-loop and closed-loop dynamics, in: *Proceedings First International Conference on Artificial Intelligence Planning Systems*, College Park, MD (1992) 204–213.

[67] U.M. Schwuttke and L. Gasser, Real-time metareasoning with dynamic trade-off evaluation, in: *Proceedings AAAI-92*, San Jose, CA (1992) 500–506.

[68] D. Subramanian and R. Feldman, The utility of EBL in recursive domain theories, in: *Proceedings AAAI-90*, Boston, MA (1990) 942–949.

[69] D. Subramanian and S. Hunter, Measuring utility and the design of provably good EBL algorithms, in: *Proceedings Ninth International Conference on Machine Learning*, Aberdeen (1992) 426–435.

[70] P. Tadepalli, Learning with inscrutable theories, in: *Proceedings Eighth International Workshop on Machine Learning*, Evanston, IL (1991) 544–548.

[71] M.P. Wellman and J. Doyle, Modular utility representation for decision-theoretic planning, in: *Proceedings First International Conference on Artificial Intelligence Planning Systems*, College Park, MD (1992) 236–242.