Statistical Selection
Among Problem-Solving Methods

Eugene Fink

January 1997

CMU-CS-97-101

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

The choice of an appropriate problem-solving method, from available methods, is a crucial skill for human experts in many areas. We describe a technique for automatic selection among methods, based on a statistical analysis of their past performances.

We formalize the statistical problem involved in selecting an efficient problem-solving method, derive a solution to this problem, and describe a selection algorithm. The algorithm not only chooses among available methods, but also decides when to abandon the chosen method, if it proves to take too much time. We extend our basic statistical technique to account for problem sizes and for similarity between problems.

We give empirical results of the use of this technique to select among search engines in the PRODIGY system. We also test the selection technique on artificially generated performance data, using several different probability distributions.

This work was sponsored by the Wright Laboratory, Aeronautical Systems Center, Air Force Materiel Command, USAF, and Defense Advanced Research Project Agency (DARPA) under grant number F33615-93-1-1330.

Keywords: Learning, problem solving, statistical analysis.

1 Introduction

The choice of an appropriate problem-solving method is one of the main themes of Polya's famous book *How to Solve It* [Polya, 1957]. Polya showed that the selection of an effective approach to a problem is a crucial skill for a student of mathematics. Psychologists have accumulated much evidence that confirms Polya's pioneering insight: the performance of human experts in many areas depends on their proficiency in choosing a method that fits a problem [Newell and Simon, 1972; Gentner and Stevens, 1983].

The purpose of our research is to automate the selection of a problem-solving method. This research is motivated by our work on the PRODIGY problem-solving system, which includes several search engines [Veloso and Stone, 1995] and a number of learning modules [Veloso *et al.*, 1995]. We need to provide a mechanism for deciding which learning modules and which search engine are appropriate for a given problem. Moreover, since programs in the real world cannot run forever, we need some means to decide when to interrupt an unsuccessful search.

We describe a learning algorithm that gathers data on the performance of available methods and uses these data to select a method that fits a given problem. The algorithm also selects a time bound for the chosen method; we interrupt the method if it hits the bound without solving the problem. Our technique is aimed at selecting a method and time bound *before* solving a given problem. We do not provide a mechanism for switching a method or revising the selected bound *during* the search for a solution. Developing such a mechanism is an important open problem.

The selection technique is very general and independent of particular problem-solving engines and problem domains; it does *not* use any specific properties of PRODIGY. We can use our learning algorithm in any AI system that offers the choice of multiple problem-solving engines or allows the selection of appropriate values of "knob" parameters. The technique is equally effective for small and large-scale problem domains.

Even though AI problem solving provided the motivation for our work, the resulting technique is applicable to situations outside of AI. We may use it to select between several alternative approaches to a task, or to decide on the amount of effort that we should invest in achieving a goal before giving up. For example, we can apply this technique to find out which of several encyclopedias is most effective for finding certain types of data. As another example, we may use it to decide how long one should wait on the phone, before hanging up, if her or his party does not answer.

The selection takes very little computation and its running time is usually negligible compared to the problem-solving time. The time of selecting a search engine in the PRODIGY system is three orders of magnitude smaller than the time of the subsequent search.

We begin by formalizing the statistical problem of estimating the expected performance of a method (Section 2). We derive a solution to this problem (Section 3), show how to use it in selecting a method and time bound (Section 4), and give results of selecting among PRODIGY search engines (Section 5). We then apply our technique to determine how long one should wait before hanging up, when the other party does not answer the phone.

We describe the use of an approximate measure of problem complexity (Section 6) and similarity between problems (Section 7) to improve the accuracy of performance estimates.

Note that we do not need a perfect estimate; *we only need accuracy sufficient for selecting the right method and a near-optimal time bound.* Finally, we test the selection technique on artificially generated performance data, for several different probability distributions (Section 8).

2 Motivating example

We give an example of a method-selection task in the PRODIGY system, use it to formalize the statistical problem of choosing from available methods, and discuss the simplifying assumptions underlying our formal model.

Suppose that we use PRODIGY to construct plans for transporting packages by vans between different locations in a city [Veloso, 1994]. We consider the use of three different search methods. The first of them is based on the control rules designed by Veloso [1994] and Pérez [1995], which guide PRODIGY's search in the transportation domain. This method applies the selected actions to the current state of the simulated transportation world as early as possible; we call this method APPLY.

The second method uses the same control rules and a special rule that delays the application of the selected actions and forces more emphasis on the backward search [Veloso and Stone, 1995]; we call it DELAY. This method is a combination of the SABA search algorithm, implemented by Veloso and Stone, with the domain-specific control rules.

The third method, ALPINE [Knoblock, 1994], is a combination of APPLY with an abstraction generator, which determines relative importance of the elements of a problem domain. ALPINE first ignores the less important elements and generates an outline of a solution; it then refines the solution to take care of the initially ignored details.

Experiments have demonstrated that delaying the application improves the efficiency of problem solving in some domains, but slows PRODIGY down in others [Stone *et al.*, 1994]; abstraction sometimes gives drastic time savings and sometimes worsens the performance [Knoblock, 1991; Bacchus and Yang, 1992]. The most reliable way to select an efficient method for a given problem domain is by empirical comparison.

The application of a method to a problem gives one of three outcomes: it may solve the problem; it may terminate with failure, after exhausting the available search space without finding a solution; or we may interrupt it, if it reaches some pre-set time bound without termination. (PRODIGY domains are completely deterministic and failures happen because of imperfect search heuristics rather than unexpected events during the execution.)

In Table 1, we give the results of solving thirty transportation problems, by each of the three methods. We denote successes by s, failures by f, and hitting the time bound by b. Note that our data are only for illustrating the selection problem, and *not* for the purpose of a general comparison of these three search techniques. Their relative performance may be very different in other domains. Also note that our selection technique does *not* rely on specific properties of PRODIGY search engines. It is equally applicable to the selection among multiple problem-solving methods in any AI system.

Even though each method outperforms the others on at least one transportation problem (see Table 1), a glance at the data reveals that APPLY's performance in this domain is probably the best among the three. We use statistical analysis to confirm this intuitive

#	time (sec) and outcome			# of	#	time (sec) and outcome			# of
	APPLY	DELAY	ALPINE	packs		APPLY	DELAY	ALPINE	packs
1	1.6 s	1.6 s	1.6 s	1	16	4.4 s	68.4 s	4.6 s	4
2	2.1 s	2.1 s	2.0 s	1	17	6.0 s	200.0 b	6.2 s	6
3	2.4 s	5.8 s	4.4 s	2	18	7.6 s	200.0 b	7.8 s	8
4	5.6 s	6.2 s	7.6 s	2	19	11.6 s	200.0 b	11.0 s	12
5	3.2 s	13.4 s	5.0 s	3	20	200.0 b	200.0 b	200.0 b	16
6	54.3 s	13.8 f	81.4 s	3	21	3.2 s	2.9 s	4.2 s	2
7	4.0 s	31.2 f	6.3 s	4	22	6.4 s	3.2 s	7.8 s	4
8	200.0 b	31.6 f	200.0 b	4	23	27.0 s	4.4 s	42.2 s	16
9	7.2 s	200.0 b	8.8 s	8	24	200.0 b	6.0 s	200.0 b	8
10	200.0 b	200.0 b	200.0 b	8	25	4.8 s	11.8 f	3.2 s	3
11	2.8 s	2.8 s	2.8 s	2	26	200.0 b	63.4 f	6.6 f	6
12	3.8 s	3.8 s	3.0 s	2	27	6.4 s	29.1 f	5.4 f	4
13	4.4 s	76.8 s	3.2 s	4	28	9.6 s	69.4 f	7.8 f	6
14	200.0 b	200.0 b	6.4 s	4	29	200.0 b	200.0 b	10.2 f	8
15	2.8 s	2.8 s	2.8 s	2	30	6.0 s	19.1 s	5.4 f	4

Table 1: Performance of APPLY, DELAY, and ALPINE on thirty transportation problems.

conclusion and to estimate its statistical significance. We also show how to select a time bound for the chosen problem-solving method.

If we identify several distinct problem types in a domain, we may discover that different types require different problem-solving methods and time bounds. In other words, the appropriate choice of a method may depend on the properties of a problem. We will analyze such situations in Section 7.

To compare different methods, we need to specify a utility function for evaluating their performances. We assume that we have to pay for running time and that we get a certain reward R for finding a solution. If the method solves the problem, the overall gain is $(R - time)$. If the method fails or hits a time bound, it is $(-time)$. We need to estimate the expected gain for all candidate methods and time bounds, and to select the method and bound that maximize the expectation, which gives us the following statistical problem.

Problem: *Suppose that a method solved n problems, failed on m problems, and was interrupted (upon hitting a time bound) on k problems. The success times were $s_1, s_2, ..., s_n$, the failure times were $f_1, f_2, ..., f_m$, and the interrupt times were $b_1, b_2, ..., b_k$. Given a reward R for solving a new problem and a time bound B, estimate the expected gain and determine the standard deviation of the estimate.*

We use the *stationarity assumption* [Valiant, 1984]; that is, we assume that the past problems and the new problem are drawn randomly from the same population, using the same probability distribution. We also assume that the method's performance does not improve over time (that is, no learning). Note that the model does *not* allow the dynamic adjustment of the bound B, based on the findings *during* the search for the new problem's solution.

We need a gain estimate that makes the best use of the available data, even if they are not sufficient for statistical significance. We cannot ask for more data, since experimentation is

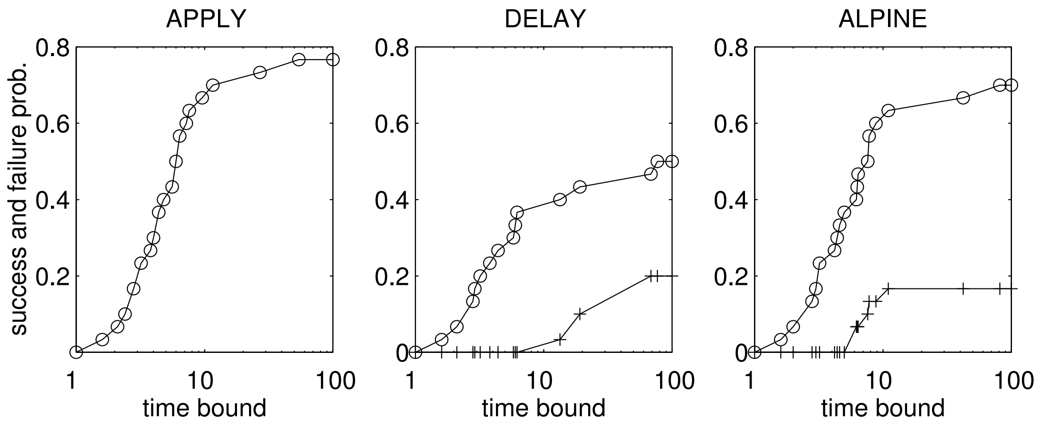


Figure 1: Dependency of the success (o) and failure (+) probabilities on the time bound.

usually much more expensive than solving the new problem. Another important requirement is the speed of statistical computations, especially since the model does not account for this addition to the overall problem-solving time.

3 Statistical foundations

We determine the probability of solving the problem within a given time bound, the probability of terminating with failure, the estimate of the expected gain, and the standard deviation of the estimate.

We assume, for convenience, that the success, failure, and interrupt times are sorted in the increasing order; that is, $s_1 \le ... \le s_n$, $f_1 \le ... \le f_m$, and $b_1 \le ... \le b_k$. We first consider the case when the time bound B is no larger than the lowest of the past time bounds, $B \le b_1$. We denote the number of success times that are no larger than B by c, and the number of failures within B by d; that is, $s_c \le B < s_{c+1}$ and $f_d \le B < f_{d+1}$.

We estimate the probability of success by the fraction of problems that were solved within time B, which is $\frac{c}{n+m+k}$; similarly, the probability of terminating with failure is $\frac{d}{n+m+k}$. For example, the probability that ALPINE with time bound 6.0 solves a transportation problem is $\frac{11}{30} = 0.37$, and the probability that it terminates with failure is $\frac{2}{30} = 0.07$.

In Figure 1, we show the dependency between the time bound (given in the horizontal axis, on a logarithmic scale) and the estimated success and failure probabilities for APPLY, DELAY, and ALPINE, in the transportation domain. We do not show the failure-probability estimate for APPLY, because the data for this method contain no failures and, thus, the estimate is zero. We computed the probabilities only for the points marked by circles and pluses, and connected them by straight segments.

We estimate the expected gain by averaging the gains that would be obtained in the past, if we used the reward R and time bound B. The method would solve c problems, earning the gains $R - s_1, R - s_2, ..., R - s_c$. It would terminate with failure d times, resulting in the negative gains $-f_1, -f_2, ..., -f_d$. In the remaining $n + m + k - c - d$ cases, it would hit the time bound, each time earning $-B$. The expected gain is equal to the mean of all these

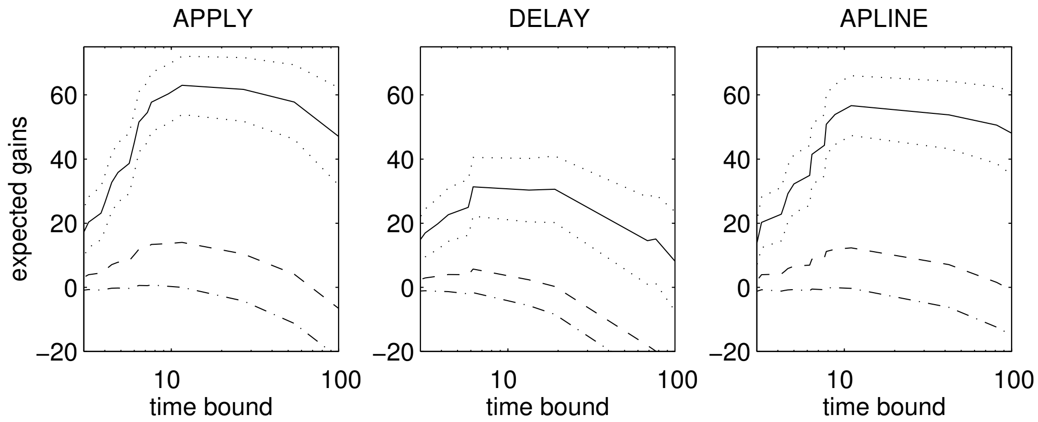


Figure 2: Dependency of the expected gain on the time bound, for the reward of 10.0 (dash-and-dot lines), 30.0 (dashed lines), and 100.0 (solid lines). The dotted lines show the standard deviation of the expected gain for the 100.0 reward.

$n + m + k$ gains:

$$\frac{\sum_{i=1}^{c}(R - s_i) + \sum_{j=1}^{d}(-f_j) + (n + m + k - c - d) \cdot (-B)}{n + m + k}.$$

For example, if we use ALPINE to solve transportation problems, with reward 30.0 and time bound 6.0, then the expected gain is 6.0.

Since we have computed the mean gain for a randomly selected sample of problems, it may be different from the mean of the overall population. We estimate the standard deviation of the expected gain using the formula for the deviation of a sample mean:

$$\sqrt{\frac{SqrSum - \frac{Sum^2}{n+m+k}}{(n + m + k) \cdot (n + m + k - 1)}},$$

where

$$Sum = \sum_{i=1}^{c}(R - s_i) + \sum_{j=1}^{d}(-f_j) + (n + m + k - c - d) \cdot (-B),$$
$$SqrSum = \sum_{i=1}^{c}(R - s_i)^2 + \sum_{j=1}^{d} f_j^2 + (n + m + k - c - d) \cdot B^2.$$

This formula is an approximation based on the *Central Limit Theorem*, which states that the distribution of sample means is always close to normal (see, for example, Mendenhall's textbook [1987]). The accuracy of the approximation improves with sample size; for thirty or more sample problems, it is near-perfect. For example, if we use ALPINE with reward 30.0 and time bound 6.0, the standard deviation of the expected gain is 2.9.

In Figure 2, we show the dependency of the expected gain on the time bound for our three methods. We give the dependency for three different values of the reward R, 10.0 (dash-and-dot lines), 30.0 (dashed lines), and 100.0 (solid lines). The dotted lines show the standard deviation of the gain estimate for the 100.0 reward: the lower line is "one deviation below" the estimate, and the upper line is "one deviation above."

We have so far assumed that $B \leq b_1$. We now consider the case when B is larger than e of the past interrupt times; that is, $b_e < B \leq b_{e+1}$. We cannot use $b_1, b_2, ..., b_e$ directly in the gain estimate, because these interrupt times are smaller than B. The use of the time bound B would cause the method to run beyond these old bounds. The collected data do not tell us whether the method with the time bound B would have succeeded on the corresponding past problems.

If we had not interrupted the method at b_1 in the past, it would have succeeded or failed at some larger time, or hit a larger time bound. We may estimate the expected gain using the data on the past problem-solving episodes in which the method ran beyond b_1. We get this estimate by averaging the gains for all the larger-time outcomes. We incorporate this averaging into the computation by removing b_1 from the sample and distributing its chance to occur among the larger-time outcomes.

To implement this re-distribution, we assign weights to the outcomes. Initially, the weight of every outcome is 1. After removing b_1, we distribute its weight among all the larger-than-b_1 outcomes. If the number of such outcomes is a_1, each of them gets the weight of $1 + \frac{1}{a_1}$. Note that $b_2, ..., b_e$ are all larger than b_1, and thus they all get the new weight.

We next remove b_2 from the sample and distribute its weight, which is $1 + \frac{1}{a_1}$, among the larger-than-b_2 outcomes. If the number of such outcomes is a_2, then we increase their weights by $(1 + \frac{1}{a_1}) \cdot \frac{1}{a_2}$; that is, their weights become $(1 + \frac{1}{a_1}) \cdot (1 + \frac{1}{a_2})$. We repeat the distribution process for the all interrupt times smaller than B; that is, for $b_3, ..., b_e$.

We illustrate the use of this re-distribution technique using the data on ALPINE's performance. Suppose that we interrupted ALPINE on problem 4 after 4.5 seconds of the execution and on problem 7 after 5.5 seconds, thus obtaining the data shown in Table 2(a), and that we need to estimate the gain for $B = 6.0$. This bound B is larger than two interrupt times.

We first have to distribute the weight of b_1. In this example, b_1 is 4.5 and there are 21 problems with larger times. We remove 4.5 from the sample data and increase the weights of the larger-time outcomes from 1 to $1 + \frac{1}{21} = 1.048$ (see Table 2b). We next perform the distribution for b_2, which is 5.5. The table contains 15 problems with larger-than-b_2 times. We distribute b_2's weight, 1.048, among these 15 problems, thus increasing their weight to $1.048 + \frac{1.048}{15} = 1.118$ (Table 2c).

We have thus distributed the interrupt times $b_1, b_2, ..., b_e$ and assigned weights to the successes times, failure times, and remaining interrupt times. We denote the resulting weights of the success times $s_1, ..., s_c$ by $u_1, ..., u_c$, and the weights of the failure times $f_1, ..., f_d$ by $v_1, ..., v_d$ (recall that these success and failure times are smaller than B). All success, failure, and interrupt times larger than B have the same weight, which we denote by w. Note that the sum of the weights is equal to the number of problems in the original sample; that is, $\sum_{i=1}^{c} v_i + \sum_{j=1}^{d} v_j + (n + m + k - c - d - e) \cdot w = n + m + k$.

We have obtained $(n+m+k-e)$ weighted times. We first use them to estimate the success and failure probabilities. The probability of solving a problem, within the time bound B, is $\frac{u_1+u_2+...+u_c}{n+m+k}$; similarly, the probability of terminating with failure is $\frac{v_1+v_2+...+v_d}{n+m+k}$. If we use the data in Table 2(c) to determine these probabilities for ALPINE with time bound 6.0, we get the success probability of $\frac{11.048}{30} = 0.37$ and the failure probability of $\frac{2.096}{30} = 0.07$.

#	ALPINE's time
1	1.6 s
2	2.0 s
3	4.4 s
4	4.5 b
5	5.0 s
6	81.4 s
7	5.5 b
8	200.0 b
9	8.8 s
	...
29	10.2 f
30	5.4 f

(a)

$\longrightarrow$

weight	time
1.000	1.6 s
1.000	2.0 s
1.000	4.4 s
–	–
1.048	5.0 s
1.048	81.4 s
1.048	5.5 b
1.048	200.0 b
1.048	8.8 s
...	
1.048	10.2 f
1.048	5.4 f

(b)

$\longrightarrow$

weight	time
1.000	1.6 s
1.000	2.0 s
1.000	4.4 s
–	–
1.048	5.0 s
1.118	81.4 s
–	–
1.118	200.0 b
1.118	8.8 s
...	
1.118	10.2 f
1.048	5.4 f

(c)

Table 2: Distributing the weights of interrupt times among the larger-time outcomes.

We next use the $(n+m+k-e)$ weighted times to compute the expected gain:

$$\frac{\sum_{i=1}^{c} u_i \cdot (R - s_i) + \sum_{j=1}^{d} v_j \cdot (-f_j) + (n+m+k-c-d-e) \cdot w \cdot (-B)}{n+m+k}.$$

Similarly, we use the weights in estimating the standard deviation of the expected gain:

$$\sqrt{\frac{SqrSum - \frac{Sum^2}{n+m+k}}{(n+m+k) \cdot (n+m+k-e-1)}},$$

where

$$Sum = \sum_{i=1}^{c} u_i \cdot (R - s_i) + \sum_{j=1}^{d} v_j \cdot (-f_j) + (n+m+k-c-d) \cdot w \cdot (-B),$$
$$SqrSum = \sum_{i=1}^{c} u_i \cdot (R - s_i)^2 + \sum_{j=1}^{d} v_j \cdot f_j^2 + (n+m+k-c-d) \cdot w \cdot B^2.$$

The application of these formulas to the data in Table 2(c), for ALPINE with reward 30.0 and time bound 6.0, gives the expected gain 6.1 and the standard deviation 3.0.

If B is larger than the largest of the past bounds (that is, $B > b_k$) and the largest time bound is larger than all past success and failure times (that is, $b_k > s_n$ and $b_k > f_m$), then the re-distribution procedure does not work. We need to distribute the weight of b_k among the larger-time problems, but the sample has no such problems. Thus, the data are not sufficient for the statistical analysis because we do not have any past experience with large enough time bounds.

We have assumed in the derivation that the execution cost is proportional to the running time; however, we may readily extend the results to any other monotone dependency between time and cost, by replacing the terms $(R - s_i)$, $(-f_i)$, and $(-B)$ with more complex functions.

Note that we do *not* use past rewards in the statistical estimate. The reward R may be different from the rewards earned on the sample problems. We may extend our results to

c	number of the already processed success times (the next success time to process will be s_{c+1})
d	number of the already processed failure times
e	number of the already processed interrupt times
h	number of the already processed time bounds
S_Num	sum of the weights of the processed successes, $\sum_{i=1}^{c} u_i$
F_Num	sum of the weights of the processed failures, $\sum_{j=1}^{d} v_j$
S_Sum	weighted sum of the gains for the processed successes, $\sum_{i=1}^{c} u_i \cdot (R - s_i)$
F_Sum	weighted sum of the gains for the processed failures, $\sum_{j=1}^{d} v_j \cdot (-f_j)$
Sum	weighted sum of the gains for all sample problems, for the current time bound B_{h+1}
S_SqrSum	weighted sum of the squared gains for the processed successes, $\sum_{i=1}^{c} u_i \cdot (R - s_i)^2$
F_SqrSum	weighted sum of the squared gains for the processed failures, $\sum_{j=1}^{d} v_j \cdot f_j^2$
SqrSum	weighted sum of the squared gains for all sample problems, for the time bound B_{h+1}

Figure 3: Variables used in the gain-estimate algorithm in Figure 4.

situations when the reward is a function of the solution quality, rather than a constant, but it works only if the reward function does *not* change from problem to problem. We replace each term $(R - s_i)$ by $(R_i - s_i)$, where R_i is the reward earned for the corresponding problem. The resulting expression combines the estimate of the expected reward and expected running time, which gives us the expected gain:

$$\frac{\sum_{i=1}^{c} u_i \cdot (R_i - s_i) + \sum_{j=1}^{d} v_j \cdot (-f_j) + (n + m + k - c - d - e) \cdot w \cdot (-B)}{n + m + k}.$$

If we use this approach, we also have to replace $(R - s_i)$ by $(R_i - s_i)$ in estimating the standard deviation of the expected gain.

To summarize, we can estimate the expected gain if either the reward does not depend on the solution quality or the dependency is the same for all problems. Relaxing this condition is an important open problem. We will discuss other limitations of the analysis and ways to overcome them in Section 9.

We now present an algorithm for computing the success and failure probabilities, gain estimates, and estimate deviations, for multiple values of the time bound B. We describe the variables used in the computation in Figure 3 and give the pseudocode in Figure 4.

The algorithm determines weights and computes gain estimates in one pass through the sorted list of success, failure, and interrupt times, and time-bound values. When processing a success or failure time, the algorithm increments the corresponding sums of the weighted gains and weighted squares of the gains. When processing an interrupt time, the algorithm modifies the weight value. When processing a time bound, the algorithm uses the accumulated sums of gains and squared gains to compute the gain estimate and deviation for this bound.

The time of this pass through the sorted list of bounds and running times is linear. That is, for l time bounds and a sample of n successes, m failures, and k interrupts, the algorithm's complexity is $O(l + n + m + k)$. The complexity of pre-sorting the lists is $O((l + n + m + k) \cdot \log(l + n + m + k))$, but in practice it takes much less time than the

The input of the algorithm includes: the reward R; the sorted list of success times, $s_1, ..., s_n$; the sorted list of failure times, $f_1, ..., f_m$; the sorted list of interrupt times, $b_1, ..., b_k$; and a sorted list of candidate time bounds, $B_1, ..., B_l$. The variables used in the computation are described in Figure 3.

Set the initial values:

$c := 0$; $d := 0$; $e := 0$; $h := 0$

$S\_Num := 0$; $F\_Num := 0$

$S\_Sum := 0$; $F\_Sum := 0$

$S\_SqrSum := 0$; $F\_SqrSum := 0$

Repeat the computations until finding the gains for all time bounds; that is, until $h = l$:

- Select the smallest among the following four times: s_{c+1}, f_{d+1}, b_{e+1}, and B_{h+1}.
- If the success time s_{c+1} is selected, increment the related sums:

  $S\_Num := S\_Num + w$

  $S\_Sum := S\_Sum + w \cdot (R - s_{c+1})$

  $S\_SqrSum := S\_SqrSum + w \cdot (R - s_{c+1})^2$

  $c := c + 1$
- If the failure time f_{d+1} is selected, increment the related sums:

  $F\_Num := F\_Num + w$

  $F\_Sum := F\_Sum + w \cdot (-f_{d+1})$

  $F\_SqrSum := F\_SqrSum \cdot w \cdot f_{d+1}^2$

  $d := d + 1$
- If the interrupt time b_{e+1} is selected:

  If no success or failure times are left (that is, $c = n$ and $d = m$),

  then terminate (the data are not sufficient to estimate the gains for the remaining bounds).

  Else, distribute the interrupt's weight among the remaining times, by incrementing w and e:

  $w := w + w \cdot \frac{n+m+k-c-d-e}{n+m+k-c-d-e-1}$

  $e := e + 1$
- If the time bound B_{h+1} is selected:

  First, compute the sum of the sample-problem gains and the sum of their squares:

  $Sum := S\_Sum + F\_Sum + (n + m + k - c - d - e) \cdot w \cdot (-B_{h+1})$

  $SqrSum := S\_SqrSum + F\_SqrSum + (n + m + k - c - d - e) \cdot w \cdot B_{h+1}^2$

  Now, compute the success and failure probability, gain estimate, and deviation, for B_{h+1}:

  Success probability: $\frac{S\_Num}{n+m+k}$. Gain estimate: $\frac{Sum}{n+m+k}$.

  Failure probability: $\frac{F\_Num}{n+m+k}$. Estimate deviation: $\sqrt{\frac{SqrSum - Sum^2/(n+m+k)}{(n+m+k)\cdot(n+m+k-e-1)}}$.

  Finally, increment the number of processed bounds:

  $h := h + 1$

Figure 4: Computing the success and failure probabilities, gain estimates, and estimate deviations.

rest of the computation. We implemented the algorithm in Common Lisp and tested it on Sun 5, the same machine as we used for solving the transportation problems. The running time is about $(l + n + m + k) \cdot 3 \cdot 10^{-4}$ seconds.

4 Selection of a method and time bound

We describe the use of the statistical estimate to choose among problem-solving methods and to determine appropriate time bounds. We provide heuristics for combining the exploitation of past experience with exploration of new alternatives and for making a choice in the absence of past data.

The basic technique is to estimate the gain for a number of time bounds, for each available method, and select the method and time bound with the maximal gain. For example, if the reward in the transportation domain is 30.0, than the best choice is APPLY with the time bound 11.6, which gives the expected gain of 14.0. This choice corresponds to the maximum of the dashed lines in Figure 2. If the expected gain for all time bounds is negative, than we are better off not solving the problem at all. For example, if the only available method is DELAY and the reward is 10.0 (see the dash-and-dot line in Figure 2), we should skip the problem.

For each method, we use its past success times as candidate time bounds. We compute the expected gain only for these candidate bounds. If we computed the gain for some other time bound B, we would get a smaller gain than for the closest lower success time s_i (where $s_i < B < s_{i+1}$), because extending the time bound from s_i to B would not increase the number of successes on the past problems.

In practice, we multiply the success times by 1.001 to obtain candidate bounds, in order to avoid the chance of interrupting a method too early because of rounding errors. We used such candidate bounds to construct the graphs in Figures 1 and 2. If several candidate bounds are "too close" to each other, we drop some of them, to reduce the amount of computation. In our implementation, we consider two bounds too close if they are within the factor of 1.05 from each other.

We now describe a technique for incremental learning of the performance of available methods. We assume that we begin with no past experience and accumulate performance data as we solve more problems. For each new problem, we use our statistical technique to select a method and time bound. After applying the selected method, we add the result to the performance data.

The use of the previous section's results incrementally causes a deviation from rigorous statistics: the resulting success, failure, and interrupt times are not independent, because the time bound used for each problem depends on the times of solving the previous problems. In spite of this violation of rigor, the technique gives good results in practice.

Note that we need to choose a method and time bound even when we have no past experience. Also, we sometimes need to deviate from the maximal-expectation choice in order to explore new opportunities. If we always used the selection that maximizes the expected gain, we would be stuck with the problem-solving method that yielded the first success, and we would never choose a time bound higher than the first success time.

We have not constructed a statistical model for combining exploration and exploitation. Instead, we provide a heuristical solution, which has proved to work well for selecting problem solvers in PRODIGY. We first consider the task of selecting a time bound for a fixed method, and then show how to select a method.

Selecting a time bound

If we have no previous data on a method's performance, we choose the time bound equal to the reward. This heuristic is based on the observation that, for PRODIGY search engines, the probability of solving a problem, say, within the next second, usually declines with the passage of search time. If a method has not solved a problem within half a minute, chances are it will not find a solution in the next half minute either. Thus, if the reward is 30.0 and the method has already run for 30.0 seconds, it is time to interrupt the search.

Now suppose that we have accumulated some data on the method's performance, which enable us to determine the bound with the maximal expected gain. To encourage exploration, we select the largest bound whose expected gain is "not much different" from the maximum. Let us denote the maximal expected gain by $g_{\max}$ and its standard deviation by $\sigma_{\max}$. Suppose that the expected gain for some bound is g and its deviation is σ. Then, the expected difference between the gain g and the maximal gain is $g_{\max} - g$. If we assume that our estimates are normally distributed, then the standard deviation of the expected difference is $\sqrt{\sigma_{\max}^2 + \sigma^2}$. Note that this estimate of the deviation is an approximation, because the distribution for small samples may be Student's rather than normal, and because $g_{\max}$ and g are not independent variables, as they are computed from the same data.

We say that g is "not much different" from the maximal gain if the ratio of the expected difference to its deviation is bounded by some constant. In our experiments, we set this constant to 0.1, which tends to give good experimental results:

$$\frac{g_{\max} - g}{\sqrt{\sigma_{\max}^2 + \sigma^2}} < 0.1.$$

We thus select the largest time bound whose gain estimate g satisfies this condition.

We present the results of this selection strategy in Figure 5. We ran each of the three methods on the thirty transportation problems, in order. The horizontal axes show the problem's number (from 1 to 30) and the vertical axes are the running time. The dotted lines show the selected time bounds and the dashed lines mark the time bounds that give the maximal gain estimates. The solid lines show the running times; they touch the dotted lines where the methods hit the time bound. The successfully solved problems are marked by circles and the failures are shown by pluses.

APPLY's total gain is 360.3, which makes an average of 12.0 per problem. If we used the maximal-gain time bound, 11.6, for all problems, the average gain would be 14.0 per problem. Thus, the use of incremental learning yielded a near-optimal gain, in spite of the initial ignorance. The time bounds used with this method (dotted line) converge to the estimated maximal-gain time bounds (dashed line), since the deviations of the gain estimates decrease as we solve more problems. APPLY's estimate of the maximal-gain bound, after solving all problems, is 9.6. This estimate differs from the 11.6 bound, found from Table 1, because the use of bounds that ensure a near-maximal gain prevented sufficient exploration.

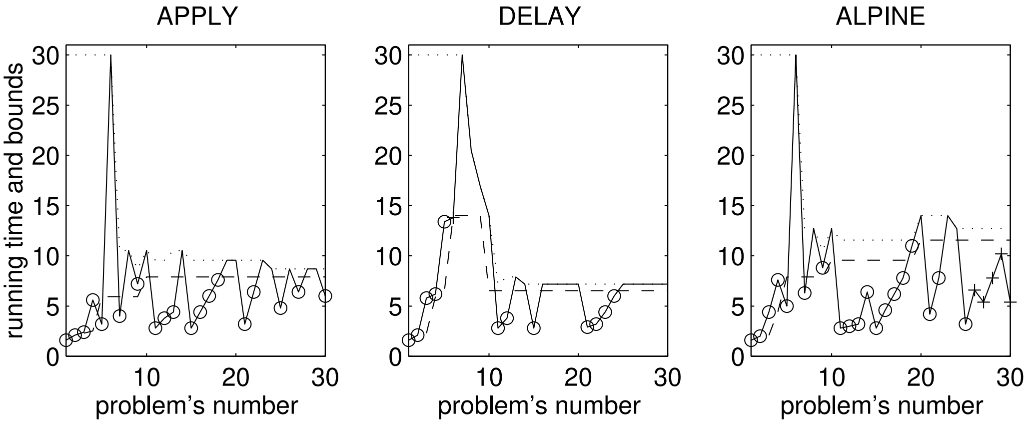


Figure 5: Results of the incremental learning of a time bound: running times (solid lines), time bounds (dotted lines), and maximal-gain time bounds (dashed lines). The successes are marked by circles and the failures by pluses.

DELAY's total gain is 115.7, or 3.9 per problem. If we used the data in Table 1 to find the optimal bound, which is 6.2, and solved all problems with this bound, we would earn 5.7 per problem. Thus, the incremental-learning gain is about two-thirds of the gain that could be obtained based on the advance knowledge. Finally, ALPINE's total gain is 339.7, or 11.3 per problem. The estimate based on Table 1 gives the bound 11.0, which would result in earning 12.3 per problem. Unlike APPLY, both DELAY and ALPINE eventually found the optimal bound.

Note that the main "losses" in the incremental learning occur on the first ten problems, when the past data are not sufficient for selecting an appropriate time bound. After this initial period, the choice of a time bound becomes close to the optimal.

The total time of the statistical computations while solving the thirty problems is 0.26 seconds, which makes less than 0.01 per problem. This time is negligible in comparison with the problem-solving time, which averages at 6.5 per problem for APPLY, 7.7 per problem for DELAY, and 7.1 per problem for ALPINE.

Selecting a method

We next describe the use of incremental learning to select a problem-solving method. If we have no data on the performance of some method, we always select this unknown method. If we have no data on several methods, we select among them at random. The optimistic use of the unknown encourages exploration during early stages of learning.

If we have past performance data for all methods, we first select a time bound for each method and determine the corresponding gain estimate and its standard deviation. We then make a weighted random selection among the methods; the chance to choose a method is equal to the probability that it is the best among the methods. This probabilistic selection results in the frequent application of methods that perform well, but also encourages some exploratory use of poor performers.

We now describe a technique for estimating the probability that a method is the best among the available methods. We use the statistical z-test to determine the probability that some method is better than another one. Suppose that the expected gain for some method is g_1 and its standard deviation is σ_1; similarly, the expected gain for another method is g_2 with deviation σ_2. The expected difference between these two gains is $g_1 - g_2$. If we assume that estimates are normally distributed, then the standard deviation of the expected difference is $\sqrt{\sigma_1^2 + \sigma_2^2}$. The z *value* is the ratio of the expected difference to its standard deviation; that is, $z = \frac{g_1 - g_2}{\sqrt{\sigma_1^2 + \sigma_2^2}}$. The z-test converts this value into the probability that the average gain of the first method is larger than that of the second method.

The use of the z-test for small samples of past performance data is an approximation; its accuracy improves with sample size. We can obtain a greater accuracy by using the t-test, which is more complex than the z-test. We did not use the t-test in the experiments, because we need the probability estimates only for our "occasional exploration" heuristic, which does not require high accuracy in determining the exploration frequency.

We find the probability that a method is the best by calculating the product of the probabilities that it outperforms individual methods. This computation is also an approximation, occasionally quite inaccurate, since the probability values that we multiply are not independent.

For example, suppose that we need to select among APPLY, DELAY, and ALPINE based on the data in Table 1. We select bound 13.1 for APPLY, which gives the gain estimate of 13.5 with deviation 3.3; bound 5.3 for DELAY, with gain estimate 5.3 and its deviation 3.0; and bound 13.2 for ALPINE, with expected gain 11.2 and its deviation 3.2. We now use the z-test to determine the probability that APPLY's gain is larger, on average, than that of DELAY. The z value is $\frac{13.5-5.3}{\sqrt{3.3^2+3.0^2}} = 1.84$; this value of z corresponds to the 0.97 probability that APPLY gives a larger average gain. Similarly, the probability of APPLY's average superiority over ALPINE is 0.69, and the probability of ALPINE's superiority over DELAY is 0.91. The probability that APPLY is the best among the three is estimated as $0.97 \cdot 0.69 = 0.67$. Similarly, the probability that ALPINE is the best is $(1 - 0.69) \cdot 0.91 = 0.28$, and DELAY's chance of being the best is $(1 - 0.97) \cdot (1 - 0.91) = 0.003$. The resulting probabilities do not add up to 1.0 because of the approximation used in estimating them. We now choose one of the methods randomly; the chance of choosing each method is proportional to its estimated probability of being the best.

We show the results of using this selection strategy in the transportation domain, for the reward of 30.0, in Figure 6. In this experiment, we first use the thirty problems from Table 1 and then sixty additional transportation problems. The horizontal axis shows the problem's number and the vertical axis is the running time. We mark successes by circles and failures by pluses. The rows of symbols below the curve show the method selection: a circle for APPLY, a cross for DELAY, and an asterisk for ALPINE.

The total gain is 998.3, which makes an average of 11.1 per problem. The overall time of the statistical computations is 0.78, or about 0.01 per problem. The selection converges to the use of APPLY with the time bound 12.7, which is optimal for this set of ninety problems. If we used this selection on all the problems, we would earn 13.3 per problem. Note that the convergence is slower than in the bound-selection experiments (see Figure 5), because we test each method only on about third of all problem.

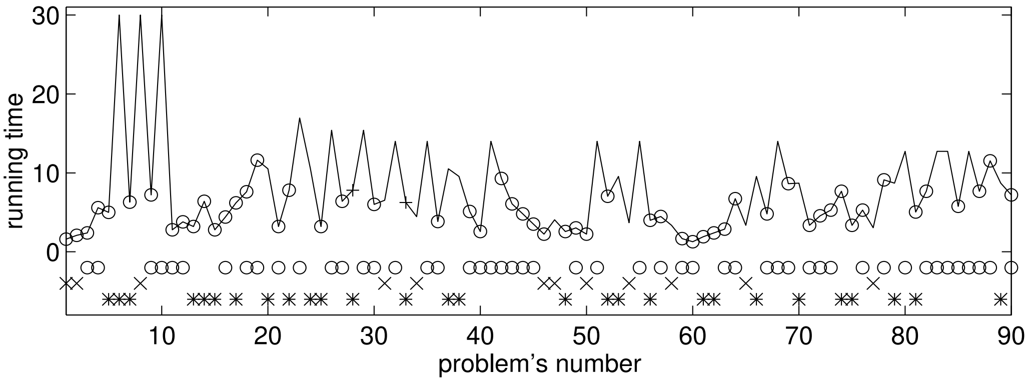


Figure 6: Results of the incremental selection of a method and time bound, on ninety transportation problems. The graph shows the running times (solid line), successes (o) and failures (+), and the selection made among APPLY (o), DELAY (x), and ALPINE ($*$).

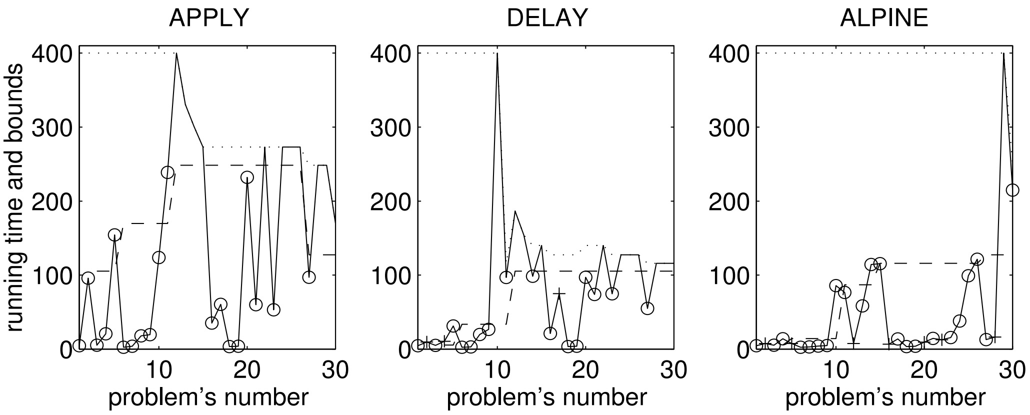


Figure 7: Incremental learning of time bounds in the extended transportation domain: running times (solid lines), time bounds (dotted lines), and maximal-gain time bounds (dashed lines). The successes are marked by circles and the failures by pluses.

5 Empirical examples

We have demonstrated the effectiveness of the statistical selection in a simple transportation domain. We now give results in two other domains.

We first consider an extended version of the transportation domain, in which we use airplanes to carry packages between cities and vans for the local delivery within cities [Veloso, 1994]. The problems in this domain are more complex, and the behavior of PRODIGY search methods differs from that in the simpler domain used in the previous sections. In Table 3, we give the performance of APPLY, DELAY, and ALPINE on thirty problems.

We present the results of the incremental learning of a time bound, for the reward of 400.0, in Figure 7. The APPLY learning gives the gain of 110.1 per problem and eventually selects the bound 127.5. The optimal bound for this set of problems is 97.0. If we used the optimal bound for all problems, we would earn 135.4 per problem.

DELAY gains 131.1 per problem and chooses the 105.3 bound at the end of the learning.

#	time (sec) and outcome			# of	#	time (sec) and outcome			# of
	APPLY	DELAY	ALPINE	packs		APPLY	DELAY	ALPINE	packs
1	4.7 s	4.7 s	4.7 s	1	16	35.1 s	21.1 s	6.6 f	2
2	96.0 s	9.6 f	7.6 f	2	17	60.5 s	75.0 f	13.7 s	2
3	5.2 s	5.1 s	5.2 s	1	18	3.5 s	3.4 s	3.5 s	1
4	20.8 s	10.6 f	14.1 s	2	19	4.0 s	3.8 s	4.0 s	1
5	154.3 s	31.4 s	7.5 f	2	20	232.1 s	97.0 s	9.5 f	2
6	2.5 s	2.5 s	2.5 s	1	21	60.1 s	73.9 s	14.6 s	2
7	4.0 s	2.9 s	3.0 s	1	22	500.0 b	500.0 b	12.7 f	2
8	18.0 s	19.8 s	4.2 s	2	23	53.1 s	74.8 s	15.6 s	2
9	19.5 s	26.8 s	4.8 s	2	24	500.0 b	500.0 b	38.0 s	4
10	123.8 s	500.0 b	85.9 s	3	25	500.0 b	213.5 s	99.2 s	4
11	238.9 s	96.8 s	76.6 s	3	26	327.6 s	179.0 s	121.4 s	6
12	500.0 b	500.0 b	7.6 f	4	27	97.0 s	54.9 s	12.8 s	6
13	345.9 s	500.0 b	58.4 s	4	28	500.0 b	500.0 b	16.4 f	8
14	458.9 s	98.4 s	114.4 s	8	29	500.0 b	500.0 b	430.8 s	16
15	500.0 b	500.0 b	115.6 s	8	30	500.0 b	398.7 s	214.8 s	8

Table 3: Performance in the extended transportation domain.

The actual optimal bound for DELAY is 98.4, the use of which on all problems would give the per-problem gain of 153.5. Finally, ALPINE gains 243.5 per problem and chooses the bound 127.6. The optimal bound for ALPINE is 430.8, the use of which would give the per-problem gain of 255.8. (ALPINE outperforms APPLY and DELAY because it uses abstraction, which separates the problem of between-city transportation by airplanes from the problem of within-city deliveries.)

Even though the bound learned for ALPINE is much smaller than optimal (127.6 versus 430.8), the resulting gain is close to optimal. The reason is that, in this experiment, ALPINE's dependency of the expected gain on the time bound has a long plateau, and the choice of a bound within the plateau does not make much difference.

Note that ALPINE's optimal bound is larger than the reward (430.8 versus 400.0). This observation shows the imperfection of the heuristic for selecting the initial time bound (see Section 4), which assumes that the optimal bound is no larger than the reward.

We show the results of the incremental selection of a method in Figure 8. In this experiment, we first use the thirty problems from Table 3 and then sixty additional problems from the extended transportation domain. The method converges to the choice of ALPINE with time bound 300.6 and gives the gain of 207.0 per problem. The best possible choice for this set of problems is the use of ALPINE with the time bound 517.1, which would give the per-problem gain of 255.8. We identified this optimal choice in a separate experiment, by running every method on all ninety problems.

We next apply our technique to a bound selection when calling to a friend on the phone. We determine how many seconds (or rings) you should wait for an answer before hanging up. The reward for reaching your party may be determined by the time that you are willing to wait in order to talk now, as opposed to hanging up and calling again later. In Table 4, we give

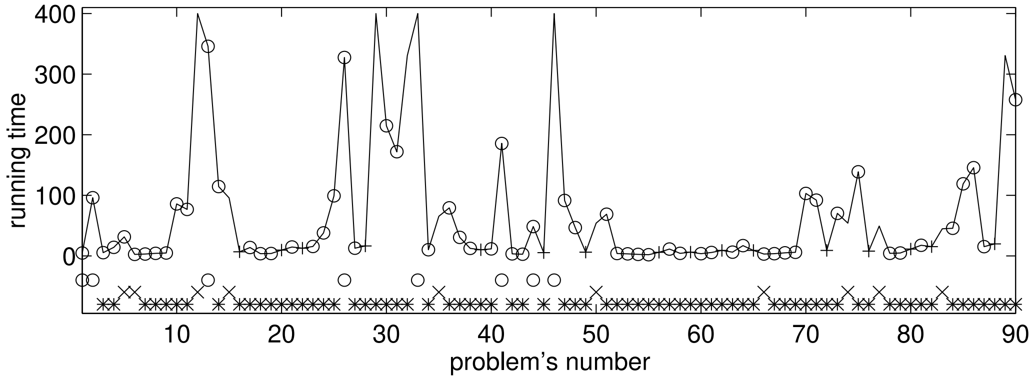


Figure 8: Selection of a method in the extended transportation domain: the running times (solid line), successes (o) and failures (+), and the selection made among APPLY (o), DELAY (x), and ALPINE (∗).

#	time	#	time	#	time	#	time	#	time
1	5.80 f	13	11.45 f	25	11.30 f	37	26.70 f	49	10.05 s
2	8.25 s	14	3.70 s	26	10.20 f	38	6.20 s	50	6.50 s
3	200.00 b	15	7.25 s	27	4.15 s	39	24.45 f	51	15.10 f
4	5.15 s	16	4.10 s	28	14.70 s	40	29.30 f	52	25.45 s
5	8.30 s	17	8.25 s	29	2.50 s	41	12.60 s	53	20.00 f
6	200.00 b	18	5.40 s	30	8.70 s	42	26.15 f	54	24.20 f
7	9.15 s	19	4.50 s	31	6.45 s	43	7.20 s	55	20.15 f
8	6.10 f	20	32.85 f	32	6.80 s	44	16.20 f	56	10.90 s
9	14.15 f	21	200.00 b	33	8.10 s	45	8.90 s	57	23.25 f
10	200.00 b	22	200.00 b	34	13.40 s	46	4.25 s	58	4.40 s
11	9.75 s	23	10.50 s	35	5.40 s	47	7.30 s	59	3.20 f
12	3.90 s	24	14.45 f	36	2.20 s	48	10.95 s	60	200.00 b

Table 4: Waiting times (seconds) in sixty phone-call experiments.

the time measurements on sixty phone calls, rounded to 0.05 seconds[1]. A success occurred when our party answered the phone. A reply by an answering machine was considered a failure.

The graph in Figure 9(a) shows the dependency of the expected gain on the time bound, for the rewards of 30.0 (dash-and-dot line), 90.0 (dashed line), and 300.0 (solid line). We assume here that the caller is not interested in leaving a message, which means that a reply by a machine gets the reward of zero. The optimal bound for the 30.0 and 90.0 rewards is 14.7 (three rings); the optimal bound for the 300.0 reward is 25.5 (five rings).

If the caller plans to leave a message, then the "failure" reward is not zero, though it may be smaller than the success reward. The graph in Figure 9(b) shows the expected gain for the success reward of 90.0 with three different failure rewards, 10.0 (dash-and-dot line), 30.0 (dashed line), and 90.0 (solid line). The optimal bound for the 10.0 failure reward is

[1]We made these calls to sixty different people at their home numbers. We measured the time from the beginning of the first ring, skipping the static silence of the connection delays.

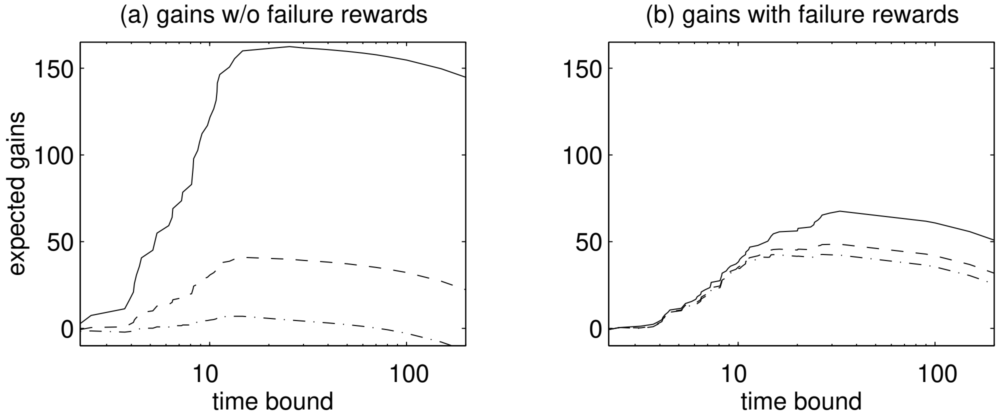


Figure 9: The dependency of the expected gain on the time bound in the phone-call domain: (a) for the rewards of 30.0 (dash-and-dot line), 90.0 (dahsed line), and 300.0 (solid line); (b) for the success reward of 90.0 and failure rewards of 10.0 (dash-and-dot line), 30.0 (dahsed line), and 90.0 (solid line).

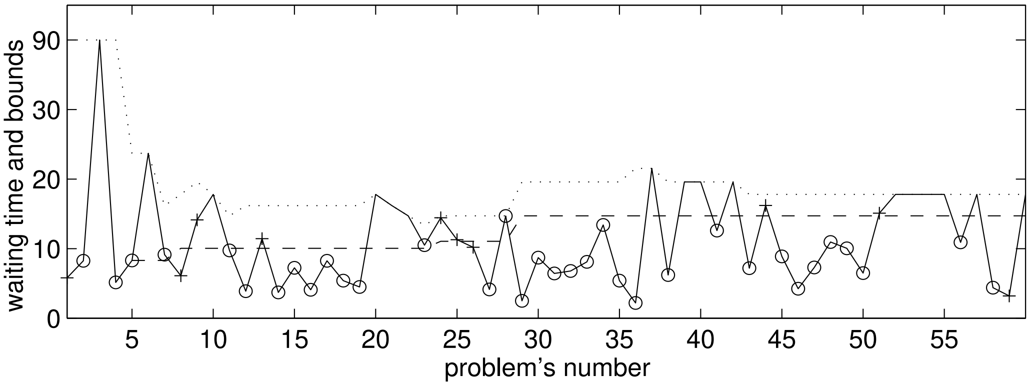


Figure 10: Incremental learning of a time bound in the phone-call domain.

26.7 (five rings); for the other two rewards, it is 32.9 (six rings).

The graph in Figure 10 shows the results of selecting a time bound incrementally, for the 90.0 success reward and zero failure reward. The learned time bound converges to the optimal bound, 14.7. The average gain obtained during the learning is 38.9 per call. If we used the optimal bound for all calls, we would earn 41.0 per call.

The experiments in the two PRODIGY domains and the phone-call domain demonstrated that the incremental-learning procedure usually finds a near-optimal time bound after solving twenty or thirty problems, and that the gain obtained during learning is close to optimal. In Section 8, we will present a series of experiments with artificially generated time values, using normal, log-normal, uniform, and log-uniform distributions. We will demonstrate that the learning technique gives good results for all four distributions.

6 Use of problem sizes

We have considered the task of finding a problem-solving method and time bound that will work well for most problems in a domain. If we can estimate the sizes of problems, we improve the performance by adjusting the time bound to a problem size.

We define a *problem size* as an easily computable positive value that correlates with the problem complexity: the larger the value, the longer it usually takes to solve the problem. Finding an accurate measure of complexity is often a difficult task; however, many domains have features that provide at least a rough complexity estimate. For example, in the transportation domain, we may estimate the problem complexity by the number of packages to be delivered. In the rightmost column of Tables 1 and 3, we show the number of packages in each of the sample problems.

Note that measures of a problem size are usually domain-specific. The choice of a good measure is the user's responsibility. We allow the user to specify different measures for different problem-solving methods.

We use regression to find the dependency between the sizes of the sample problems and the times to solve them. We use separate regressions for the times of successes and for the times of failures. In PRODIGY, successes usually occur after exploring a small part of the search space, whereas failures require the exploration of the entire space, and the dependency of the success time on the problem size is quite different from that of the failure time.

We assume that the dependency of time on size is either polynomial or exponential. If it is polynomial, than the logarithm of time depends linearly on the logarithm of size; for an exponential dependency, the time logarithm depends linearly on size. We thus use linear regression to find both polynomial and exponential dependencies.

We use the least-squares technique to perform the regression. In Figure 11(a) and 11(b), we give the regression formulas for a polynomial dependency between size and time; the regression for an exponential dependency is similar. We denote the number of sample problems by n, the problem sizes by $size_1, ..., size_n$, and the corresponding running times by $time_1, ..., time_n$.

We evaluate the regression results using the t-test. The t value in this test is the ratio of the estimated slope of the regression line to the standard deviation of the slope estimate. We give the formula for computing t in Figure 11(c). The *TimeDev* value in this formula is the standard deviation of time logarithms. It shows how much, on average, time logarithms deviate from the regression line.

The t-test converts the t value into the probability that the use of the regression gives *no better* prediction of running time than ignoring the sizes and simply taking the mean; in other words, it is the probability that the regression does not help. This probability is called the P *value*; it is a function of the t value and the number n of sample problems. When the regressed line gives a good fit to the sample data, t is large and the P value is small.

In Figure 12, we give the results of regressing the success times for the sample transportation problems from Table 1; we do *not* show failure regression. The top three graphs give the polynomial dependency of the success time on the problem size; the bottom graphs are for the exponential dependency. The horizontal axes show the problem sizes (that is, the number of packages), and the vertical axes are the times. The circles show the sizes and

(a) Approximate dependency of the running time on the problem size:

$\log time = \alpha + \beta \cdot \log size$;

that is, $time = e^{\alpha} \cdot size^{\beta}$.

(b) Regression coefficients:

$\beta = \frac{\sum_{i=1}^{n} \log size_i \cdot \log time_i - SizeSum \cdot TimeSum/n}{SizeSqrSum - SizeSum^2/n}$,

$\alpha = \frac{TimeSum}{n} - \beta \cdot \frac{SizeSum}{n}$,

where

$TimeSum = \sum_{i=1}^{n} \log time_i$,

$SizeSum = \sum_{i=1}^{n} \log size_i$,

$SizeSqrSum = \sum_{i=1}^{n} (\log size_i)^2$.

(c) The t value, for evaluating the regression accuracy:

$t = \frac{\beta}{TimeDev} \cdot \sqrt{SizeSqrSum - \frac{SizeSum^2}{n}}$,

where

$TimeDev = \sqrt{\frac{1}{n-2} \cdot \left(\sum_{i=1}^{n} (\log time_i)^2 - \frac{TimeSum^2}{n} - \beta \cdot (\sum_{i=1}^{n} \log size_i \cdot \log time_i - \frac{SizeSum \cdot TimeSum}{n})\right)}$.

Figure 11: Rergression coefficients and the t value for the polynomial dependency of time on size.

times of the problem instances; the solid lines are the regression results. For each regression, we give the t values and the corresponding intervals of the P value under the graph.

We use the regression only if the probability P is smaller than a certain bound. In our experiments, we set this bound to 0.2; that is, we used problem sizes only for $P < 0.2$. This test ensures that we use sizes only if they provide a good correlation with problem complexity. If the size measure proves inaccurate, then the gain-estimate algorithm ignores sizes. We use the 0.2 bound rather than more "customary" 0.05 or 0.02 because an early detection of a dependency between sizes and times is more important for the overall efficiency than establishing a high certainty of the dependency.

For example, all three polynomial regressions in the top row of Figure 12 pass the $P < 0.2$ test. The exponential regressions for APPLY and ALPINE also satisfy this condition. On the other hand, the exponential regression for DELAY fails the test (see the middle bottom graph in Figure 12).

The choice between the polynomial and exponential regression is based on the value of t: we prefer the regression with the larger t. In the example of Figure 12, the polynomial regression wins for all three methods.

The user has an option to select between the two regressions herself. For example, she may insist on the use of the exponential regression. We also allow the user to set a regression slope. This option is useful when the human operator has a good notion of the slope value and the past data are not sufficient for an accurate estimate. If the user specifies a slope, the algorithm uses her value in the regression; however, it compares the user's value with the regression estimate of Table 11, determines the statistical significance of the difference, and gives a warning if the user's estimate is off with high probability.

Note that the least-square regression and the related t-test make quite strong assumptions

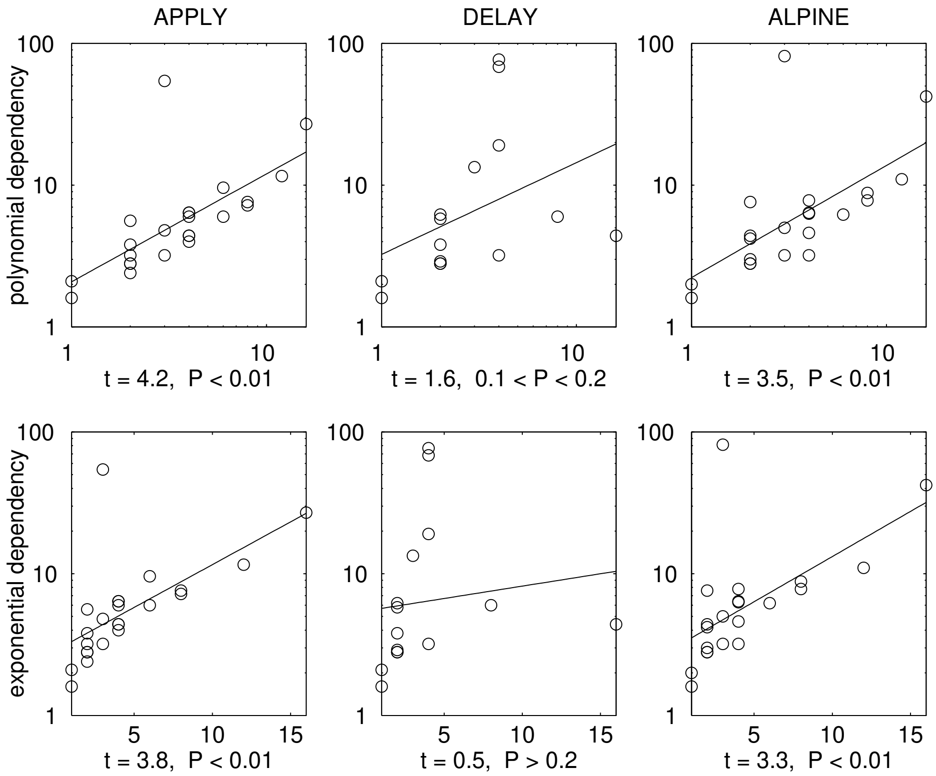


Figure 12: The dependency of the success time on the problem size. The top graphs show the regression for a polynomial dependency, and the bottom graphs are for an exponential dependency.

about the nature of the distribution. First, for problems of fixed size, the distribution of the time logarithms must be normal; that is, time must be distributed log-normally. Second, for all problem sizes, the standard deviation of the distribution must be the same. The regression, however, usually provides a good approximation of the dependency between size and time even when these assumptions are not satisfied.

The use of the problem size in estimating the gain is based on "scaling" the times of sample problems to a given size. We illustrate it in Figure 13, where we scale DELAY's times of a 1-package success, an 8-package success, and an 8-package failure for estimating the gain on a 3-package problem (the 3-package size is marked by the vertical dotted line). To scale a problem's time to a given size, we draw the line with the regression slope through the point representing the problem (see the solid lines in Figure 13), to the intersection with the vertical line through the given size (the dotted line). The ordinate of the intersection is the scaled time.

If the size of the problem is $size_{\rm old}$, the running time is $time_{\rm old}$, and we need to scale it to a size $size_{\rm new}$, using a regression slope β, then we compute the scaled time $time_{\rm new}$ as follows:

Polynomial regression:

$$\log time_{\rm new} = \log time_{\rm old} + \beta \cdot (\log size_{\rm new} - \log size_{\rm old});$$

that is, $time_{\rm new} = time_{\rm old} \cdot \left(\frac{size_{\rm new}}{size_{\rm old}}\right)^{\beta}$.

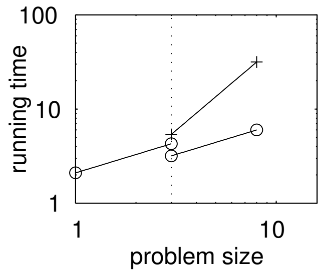


Figure 13: Scaling two success times (o) and a failure time (+) of DELAY to a 3-package problem.

Exponential regression:
$\log time_{\text{new}} = \log time_{\text{old}} + \beta \cdot (size_{\text{new}} - size_{\text{old}})$;
that is, $time_{\text{new}} = time_{\text{old}} \cdot \exp(\beta \cdot (size_{\text{new}} - size_{\text{old}}))$.

We use the slope of the success regression in scaling success times (see the lines through circles in Figure 13), and the slope of the failure regression in scaling failures (the line trough pluses). The slope for scaling an interrupt time should depend on whether the method would succeed or fail if we did not interrupt it; however, we do not know which of these two outcomes would occur. We use the simple heuristic of choosing between the success and failure slope based on which of them has the smaller P value. We also experimented with "distributing" each interrupt point between success and failure slopes, similar to the distribution of small interrupt times described in Section 3; however, it did not provide higher accuracy than the simple heuristic.

For a sample of n successes, m failures, and k interrupts, the overall time of computing the polynomial and exponential regression slopes, performing the t-test to select between the two regressions, and scaling the sample times to a given size is about $(n + m + k) \cdot 9 \cdot 10^{-4}$ seconds. For the incremental learning of a time bound, we implemented a procedure that incrementally updates the slope and t value after adding a new problem to the sample. The amortized running time of this procedure is approximately $((n+m+k) \cdot 2 + 7) \cdot 10^{-4}$ seconds per problem.

After scaling the times of the sample problems to a given size, we use the technique of Section 3 to compute the gain estimate and its standard deviation. The only difference is that we reduce the second term in the denominator for the standard deviation by 2, because the success and failure regressions reduce the number of degrees of freedom of the sample data. Thus, we compute the deviation as follows:

$$\sqrt{\frac{SqrSum - \frac{Sum^2}{n+m+k}}{(n+m+k) \cdot (n+m+k-e-3)}}.$$

In Figure 14, we show the dependency of the expected gain on the time bound when using APPLY on 1-package, 3-package, and 10-package problems in the simple transportation domain, described in Section 2.

If we estimate the problem sizes in the transportation domain by the number of packages to be delivered, and use these sizes in the incremental-selection experiments of Sections 4

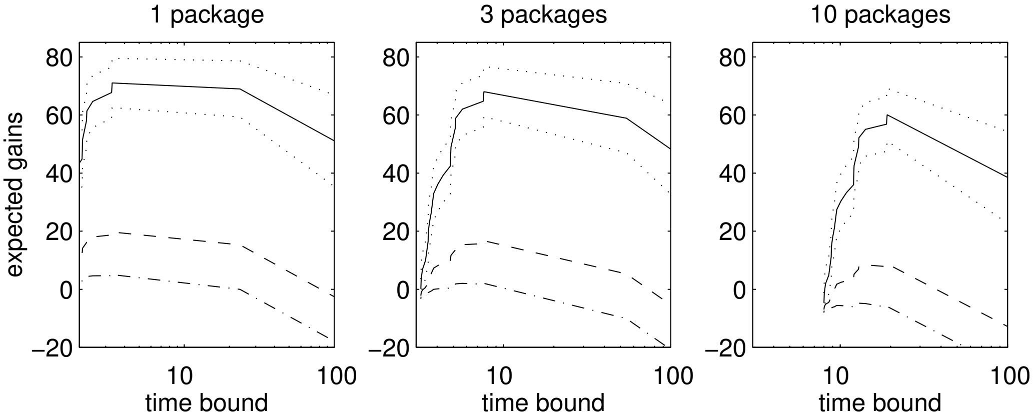


Figure 14: Dependency of APPLY's expected gain on the time bound in the simple transportation domain, for the rewards of 10.0 (dash-and-dot lines), 30.0 (dashed lines), and 100.0 (solid lines). The dotted lines show the standard deviation for the 100.0 reward.

	w/o sizes	with sizes
transportation by vans (Section 4)		
APPLY's bound selection	12.0	12.2
DELAY's bound selection	3.9	4.7
ALPINE's bound selection	11.3	11.9
method selection	11.1	11.8
transportation by vans and airplanes (Section 5)		
APPLY's bound selection	110.1	121.6
DELAY's bound selection	131.1	137.4
ALPINE's bound selection	243.5	248.3
method selection	207.0	215.6

Table 5: Per-problem gains in the learning experiments, without and with the use of sizes.

and 5, we get larger gains in all eight experiments. In Table 5, we give the per-problem gains in these experiments, without and with the use of problem sizes.

In Figure 15, we give a more detailed comparison of gains without and with the regression, for the bound-selection experiments of Section 4. The horizontal axes show the problem's number, from 7 to 30. We skip the first six problems, because the algorithm does not use sizes in selecting the time bounds for these problems: it has not yet accumulated enough data for regression with sufficiently small P value.

The vertical axes show the average per-problem gain up to the current problem. For example, the left end of the curve shows the average gain for the first seven problems and the right end gives the average for all thirty problems. The gain declines for problems 20 to 30 because these problems happen to be harder, on average, than the first twenty problems (see Table 1). The dotted lines give the average gains without the use of problem sizes, and the solid lines are for the gains obtained with the regression.

The graphs show that the use of problem sizes usually, though not always, provides a small improvement of the performance. The apparent advantage of the regression in DELAY's

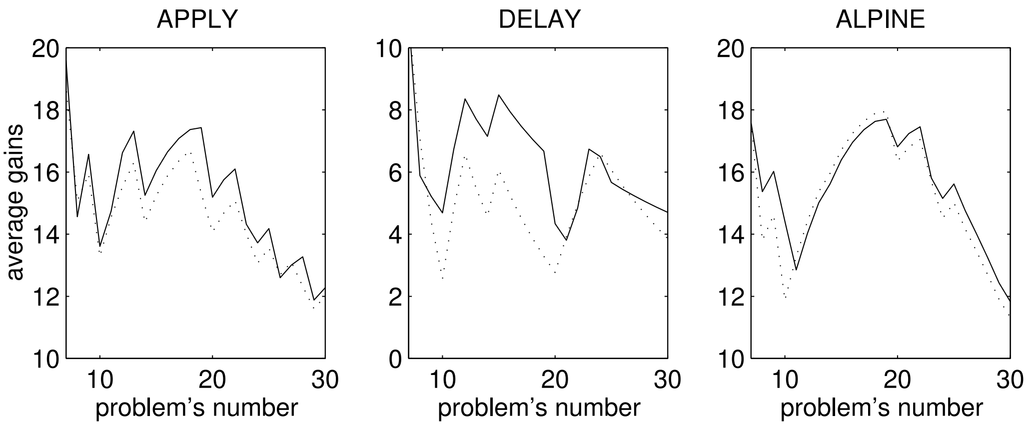


Figure 15: Average per-problem gains without the regression (dotted lines) and with the regression (solid lines), during the incremental learning of a time bound.

learning is mostly due to the choice of low time bounds for problems 9 and 10, which cannot be solved in feasible time. This luck in setting low bounds for two hard problems is not statistically significant. If the algorithm does not use problem sizes, it hits the time bounds of 16.9 and 14.0 on these problems (see Figure 5) and falls behind in its per-problem gain.

7 Similarity hierarchy

We have estimated the expected gain by averaging the gains for *all* sample problems. If we know which of them are similar to a new problem, we may improve the estimate accuracy by averaging only the gains for these similar problems.

We describe similarity among problems by a tree-structured *similarity hierarchy*. The leaf nodes of the hierarchy are groups of similar problems. The other nodes represent weaker similarity among groups. We assume that each problem belongs to exactly one group and that determining a problem's group takes little computational time.

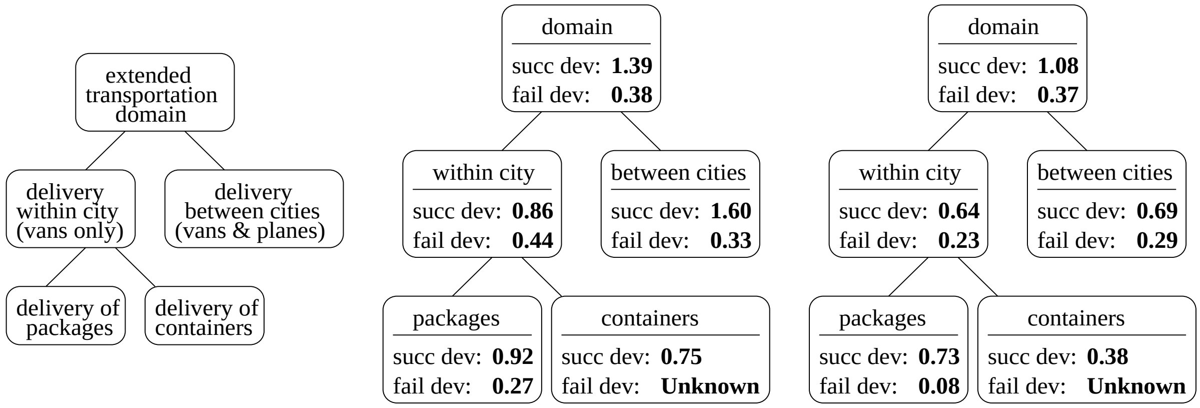
For example, we may divide the transportation problems into within-city and between-city deliveries. We extend this example by a new type of problems, which involves the transportation of containers within a city. A van can carry only one container at a time, which sometimes makes container delivery harder than package delivery. In Table 6, we give the performance of APPLY, DELAY, and ALPINE on ten container-transportation problems. We now subdivide within-city problems into package deliveries and container deliveries. We show the resulting similarity hierarchy in Figure 16(a).

The construction of a hierarchy is presently the user's responsibility. We plan to address the problem of learning a hierarchy automatically in the future work. We allow the user to construct a separate hierarchy for each problem-solving method or a common hierarchy for all methods. We also allow the use of different problem-size measures for different groups of problems.

We may estimate the similarity of problems in a group by the standard deviation of the

#	time (sec) and outcome			# of
	APPLY	DELAY	ALPINE	conts
1	2.3 s	2.3 s	2.1 s	1
2	3.1 s	5.1 s	4.1 s	2
3	5.0 s	20.2 s	4.8 s	3
4	3.3 s	8.9 s	3.2 s	2
5	6.7 s	36.8 s	6.4 s	4
6	200.0 b	200.0 b	10.1 f	8
7	3.2 s	3.2 s	3.2 s	2
8	24.0 s	200.0 b	26.3 s	8
9	4.8 s	86.2 s	3.4 s	4
10	8.0 s	200.0 b	9.4 s	6

Table 6: Performance on ten container-transportation problems.



(a) Similarity hierarchy (b) ALPINE's deviations w/o regression (c) ALPINE's deviations with regression

Figure 16: Similarity hierarchy and the deviations of ALPINE's success and failure logarithms.

logarithms of running times, computed for the sample problems that belong to the group:

$$TimeDev = \sqrt{\frac{1}{n-1} \cdot \left(\sum_{i=1}^{n} (\log time_i)^2 - \frac{(\sum_{i=1}^{n} \log time_i)^2}{n} \right)}.$$

We compute the deviations separately for successes and failures, and use these values as a heuristical measure of the hierarchy's quality. The smaller the deviations for the leaf groups, the better the user's hierarchy. If some deviation value is larger than a pre-set threshold, the system gives a warning. In the implementation, we set this threshold to 2.0.

If we use the regression, we apply it separately to each group of the similarity hierarchy. If the regression confirms the dependency between problem sizes and times, we compute the deviation of time logarithms by a different formula, given in the last line of Figure 11.

For example, the deviation values for ALPINE in the transportation domain are as shown in Figure 16. We give the deviations computed without the regression in Figure 16(b), and the deviations for gain estimates with the regression in Figure 16(c). The values show that within-city problems are more similar to each other than between-city problems.

Note that the deviations of the logarithms do not change if we multiply all times by the same factor, which means that they do not depend on the speed of a computer that runs problem-solving methods. Also, the deviation values do not change, on average, with adding more problems to the sample.

We may estimate the expected gain for a new problem by averaging the gains of the sample problems that belong to the same leaf group. Alternatively, we may use a larger sample from one of its ancestors. The leaf group has less data than its ancestors, but the deviation of these data is smaller. We need to analize this trade-off when selecting between the leaf group and its ancestors. Intuitively, we should use ancestral groups during early stages of the incremental learning and move to leaf groups after collecting more data.

We present a heuristical (rather than statistical) technique for selecting between a group and its parent, based on two tests. The first test is aimed at identifying the difference between the distribution of the group's problems and the distribution of the other problems in the parent's sample. If the two distributions prove different, we use the group rather than its parent for estimating the problem-solving gain. If not, we perform the second test, to determine whether the group's sample provides a more accurate performance estimate than the parent's sample. We now describe the two tests in detail.

If we do not use the regression, then the first test is the statistical t-test that determines whether the mean of the group's time logarithms differs from the mean of the other time logarithms in the parent's sample. We perform the test separately for successes and failures. In our experiments, we considered the means different when we could reject the null-hypothesis that they are equal with the 0.75 confidence. If we use the regression and it confirms the dependency between sizes and times, then we use a different t-test. Instead of comparing the means of time logarithms, we determine whether the regression lines are different with confidence 0.75.

A statistically significant difference for either successes or failures is a signal that the distribution of the group's running times differs from the distribution for the other problems in the group's parent. Therefore, if we need to estimate the gain for a new problem that belongs to the group, the use of the parent's sample may bias the prediction. We thus should use the group rather than its parent.

For example, suppose that we use the data in Tables 1, 3, and 6 with the hierarchy in Figure 16(a), and we need to estimate ALPINE's gain on a new problem that involves the delivery of packages within a city. We consider the choice between the corresponding leaf group and its parent. In this example, we do *not* use the regression.

The estimated mean of the success-time logarithms for the package-delivery problems is 4.07, and the standard deviation of this estimate is 0.20. The estimated mean for the other problems in the parent group, which are the container-delivery problems, is 4.03, and its deviation is 0.16. The difference between the two means is *not* statistically significant. Since the container-transportation sample has only one failure, we cannot estimate the deviation of its failure logarithms; therefore, the difference between the failure-logarithm means is also considered insignificant.

If we apply the regression to this example and use the t-test to compare the regression slopes, it also shows that package-transportation and container-transportation times are not significantly different.

The second test is the comparison of the standard deviations of the mean estimates for the group and its parent. The deviation of the mean estimate is equal to the deviation of the time logarithms divided by the square root of the sample size, $\frac{TimeDev}{\sqrt{n}}$. We compute it separately for success times and failure times. We use this value as an indicator of the sample's accuracy in estimating the problem-solving gain: the smaller the value, the greater the accuracy. This indicator accounts for the trade-off between the deviation of the running-time distribution and the sample size. It increases with an increase in the deviation and decreases with an increase in the sample size.

If the group's deviation of the mean estimate is smaller than that of the group's parent, for either successes or failures, then the group's sample is likely to provide a more accurate gain estimate; thus, we prefer the group to its parent. On the other hand, if the parent's mean-estimate deviation is smaller for both successes and failures, and the comparison of the group's mean with that of the other problems of the parent sample has not revealed a significant difference, then we use the parent to estimate the gain for a new problem.

Suppose that we apply the second test to the group selection for estimating ALPINE's gain on within-city package delivery. The standard deviation of the mean estimate of the success-time logarithms, for the corresponding leaf group, is 0.20; the deviation for its parent is 0.16. The deviation of the mean estimate of the failure-time logarithms is also smaller for the parent. Since the first test has not revealed a significant difference between the group's times and the other times in the parent's sample, we prefer the use of the parent.

After selecting between the leaf group and its parent, we use the same two tests to choose between the resulting "winner" and the group's grandparent. We then compare the new winner with the great-grandparent, and so on. In our example, we need to compare the selected parent group with the top-level node (see Figure 16a). After applying the first test, we find out that the mean of the group's success logarithms is 4.03 and the corresponding mean for the other problems in the top node's sample is 5.39. The difference between these means is statistically significant. We thus prefer the group of within-city problems to the top-level group.

The time taken by the statistical computations is proportional to the depth of a hierarchy. If we use a hierarchy in the incremental learning, and we have accumulated data on n successes, m failures, and k interrupts, then the amortized time of performing the necessary regressions, selecting a group, and scaling the times of this group to the size of the new problem is about $((n + m + k) \cdot 4 + 20) \cdot depth \cdot 10^{-4}$ seconds. This time is still very small compared to PRODIGY's problem-solving time.

We have considered several alterations of the described group-selection heuristic in our experiments. In particular, we tried replacing the deviation of time logarithms with the deviation of times divided over their mean. In most cases, the use of this measure led to the same selection. We also tried to use either success or failure times rather than both successes and failures. This alternative proved to be a less effective strategy. When successes are much more numerous than failures, which happens in most PRODIGY domains, the results of using successes and ignoring failures are near-identical to the results of using both success and failures; however, when the number of successes and failures is approximately equal, the use of both successes and failures gives better performance.

In Table 7, we present the results of using the similarity hierarchy of Figure 16 in the incremental learning, and compare them with the results obtained without a hierarchy. We ran the bound-selection experiments on a sequence of seventy transportation problems, which was constructed by interleaving the problem sets of Tables 1, 3, and 6. We used a three-times longer sequence of transportation problems for the method-selection experiments.

	using leaf groups	using the top group	heuristical group selection
	without the use of problem sizes		
APPLY's bound selection	11.8	10.5	12.1
DELAY's bound selection	7.0	4.7	7.5
ALPINE's bound selection	19.5	18.1	19.5
method selection	13.1	11.1	13.4
	with the use of problem sizes		
APPLY's bound selection	16.3	11.1	16.8
DELAY's bound selection	12.1	5.2	12.0
ALPINE's bound selection	22.6	18.4	22.6
method selection	19.4	13.7	21.0

Table 7: Per-problem gains in learning experiments, for different group-selection techniques.

In the first column, we give the results of using only leaf groups in estimating the gains. In the second column, we show the results of using the top-level group for all estimates, which means that we do not distinguish among the three problem types. The third column contains the results of using the similarity hierarchy, with our heuristic for the group selection. We first ran the experiments using both success and failure times in the group selection, and then re-ran them using only success times. In all eight cases, he results of using both successes and failures were identical to the results of using successes.

The experiments demonstrate that the use of the complete hierarchy gives larger gains than either the leaf groups or the top-level group; however, the improvement is not large.

We next use a similarity hierarchy in selecting a time bound for phone calls. We consider the outcomes of sixty-three calls to six different people. We called two of them, say A and B, at their office phones; we called the other four, C, D, E, and F, at their homes. We show our similarity hierarchy and the call outcomes in Figure 17.

For each group in the hierarchy, we give the estimated mean of success and failure time logarithms ("mean"), the deviation of the time logarithms ("deviation"), and the deviation of the mean estimate ("mean's dev"). The mean of success-time logarithms for calls to offices is significantly different from that for calls to homes, which implies that the distribution of office-call times differs from the distribution of home-call times.

The mean success logarithms for persons A and B are *not* significantly different from each other. Similarly, the success means of C, D, and E do *not* differ significantly from the mean of the home-call group. On the other hand, the success mean of F *is* significantly different from the mean for the other people in the home-call group, implying that the time distribution for F differs from the rest of its parent group. Finally, the failure-logarithm means of D, E, and F are all significantly different from each other.

We ran incremental-learning experiments on these data with the reward of 90.0. An experiment with the use of the leaf groups for all gain estimates yielded the gain of 57.8 per call. We then ran an experiment using the home-call and office-call groups for all estimates, without distinguishing among different people within these groups, and obtained the average gain of 56.3. We next used the top-level group for all estimates, which yielded 55.9 per call. Finally, we experimented with the use of our heuristic for choosing between the leaf groups

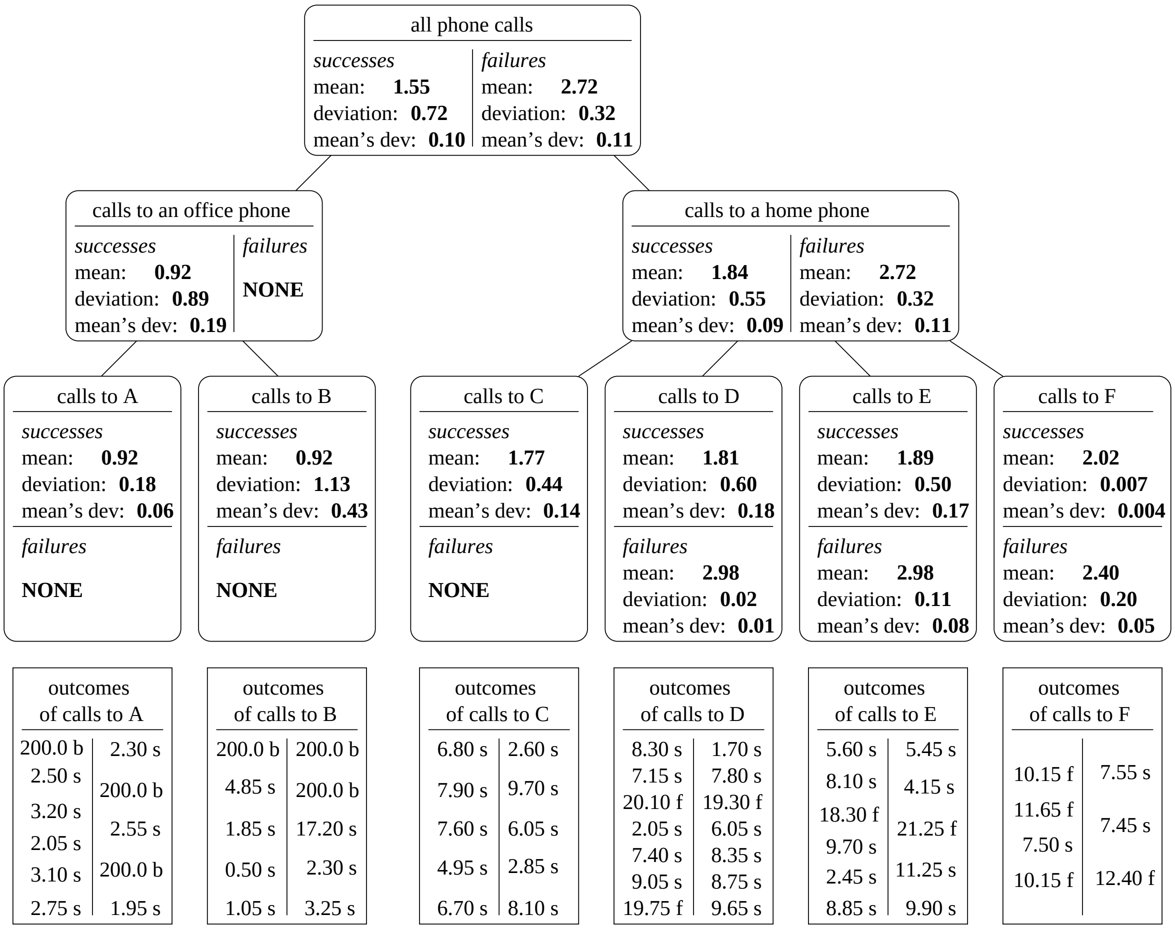


Figure 17: Similarity hierarchy and call outcomes in the phone-call domain.

and their ancestors based on the means and deviations of time logarithms; the gain in this experiment was 59.8 per call. If we knew the time distributions in advance, determined the optimal time bound for each leaf group, and used these optimal bounds for all call, then the average gain would be 61.9.

The phone-call experiments have confirmed that the use of a similarity hierarchy improves the performance, though not by much. Note, however, that the gain obtained with the use of the hierarchy is much closer to the optimal than the gain from the use of leaf groups or top-level group.

8 Artificial tests

We give the results of testing the selection mechanism on artificially generated values of success and failure times. The "running times" in these tests are the values produced by a random-number generator. The artificial data enable us to perform controlled experiments with known distributions.

The learning mechanism has proved effective for all tested distributions. The experiments have demonstrated that the gain obtained in the incremental learning is usually close to the optimal. They have also shown that the use of the regression improves the performance when there is a correlation between size and time, and does not worsen the results when there is no correlation. We have not found a significant difference in performance for different distributions.

We consider the following four distribution types:

Normal: The normal distribution of success and failure times corresponds to the situation when the running time for most problems is close to some "typical" value, and problems with much smaller or much larger times are rare.

Log-Normal: The distribution of times is called log-normal if time logarithms are distributed normally. Intuitively, this distribution occurs when the "complexity" of most problems is close to some typical complexity and the problem-solving time grows exponentially with complexity.

Uniform: The times are distributed uniformly if they belong to some fixed interval and all values in this interval are equally likely; thus, there is no "typical" running-time value.

Log-Uniform: The logarithms of running times are distributed uniformly. Intuitively, the complexity of problems is within some fixed interval, and running time is exponential in complexity.

For each of the four distribution types, we ran multiple tests, varying the values of the following parameters:

Success and failure probabilities: We varied the probabilities of success, failure, and infinite looping.

Mean and deviation: We experimented with different values of the mean and standard deviation of success-time and failure-time distributions.

Reward: We set the reward to 100.0 in all the experiments.

Length of the problem sequence: We tested the incremental-learning mechanism on sequences of 50, 150, and 500 problems.

Correlation between sizes and times: We ran tests both without and with the use of problem sizes. We experimented with three different correlations between size logarithms and time logarithms: 0.0, 0.6, and 0.9.

We ran fifty independent experiments for each setting of the parameters and averaged their results. Thus, every graph in this section shows the average of fifty experiments.

Since the learning technique has proved effective in all these tests, we conjecture that it also works well for most other distributions. We plan to experiment with a wider variety of distributions and identify situations in which the technique does not give good results.

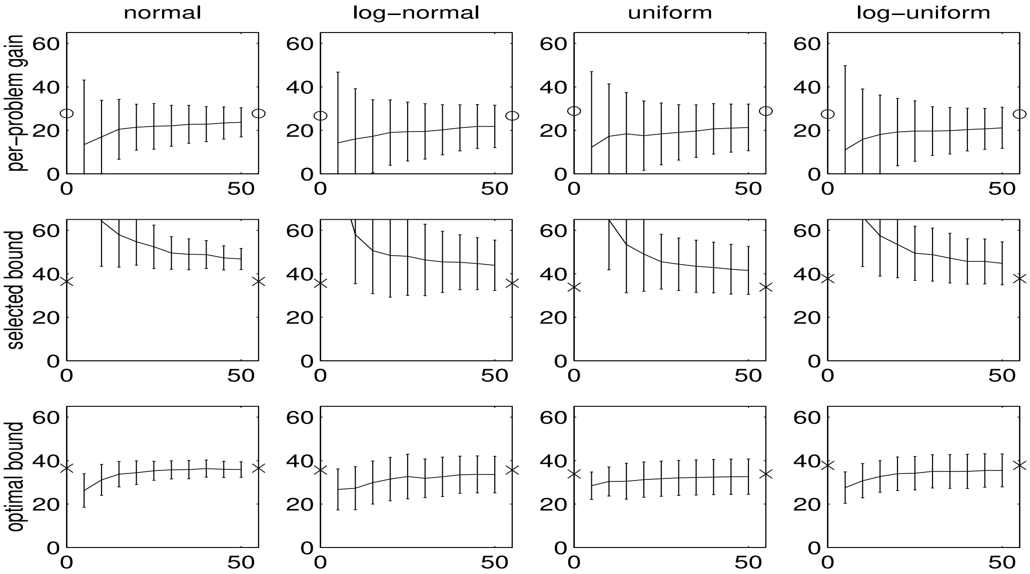


Figure 18: Per-problem gains (top row), time bounds (middle row), and estimates of the optimal time bounds (bottom row) in the incremental learning on 50-problem sequences. The crosses mark the optimal time bounds and the circles show the expected gains for the optimal bounds.

Experiments with short and long problem sequences

We first present the results of learning a time bound on sequences of 50 and 500 problems, without the use of problem sizes. The success probability in these experiments is 1/2, the failure probability is 1/4, and the probability of infinite looping is also 1/4. The mean of success times is 20.0 and their standard deviation is 8.0; the failure-time mean is 10.0 and standard deviation is 4.0. We experimented with all four distribution types. For each distribution, we ran fifty experiments and averaged their results.

In Figure 18, we summarize the results for 50-problem sequences. The horizontal axes in all graphs show the problem's number in a sequence. The top row of graphs gives the average per-problem gain obtained up to the current problem. The circles mark the gain that the system would obtain if it knew the distribution in advance and used the optimal time bound for all problems. The vertical bars show the width of the distribution of gain values obtained in different experiments. Each bar covers two standard deviations up and down, which means that 95% of the experiments fall within it.

The middle row of graphs shows the selected time bounds. The bottom row of graphs gives the system's estimates of the optimal time bound (recall that the selected bounds are larger than optimal, to encourage exploration). The crosses mark the values of the optimal time bounds. Note that the system's estimates of the optimal bounds converge to their real values.

In Figure 19, we give similar results for 500-problem sequences. In these experiments, per-problem gains come closer to the optimal values, but still do not reach them. The difference between the obtained and optimal gains comes from losses during early stages of learning and from the use of larger-than-optimal bounds.

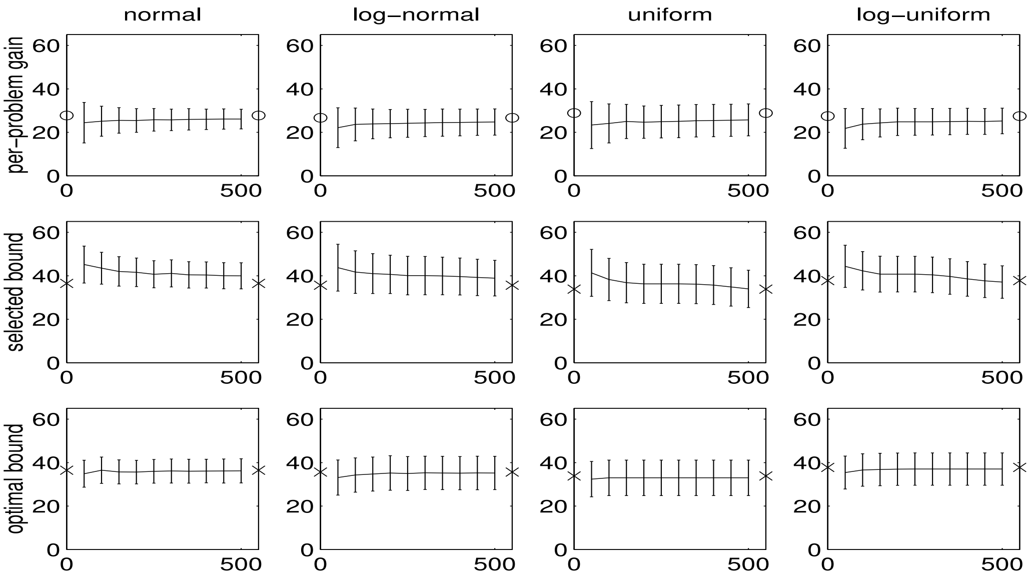


Figure 19: Per-problem gains (top row), time bounds (middle row), and estimates of the optimal time bounds (bottom row) in the incremental learning on 500-problem sequences.

Varying success and failure probabilities

We give the results of learning a time bound for different probabilities of successes and failures. The means and standard deviations of the success and failure times are the same as in the previous experiments.

We summarize the results in Figure 20. The top row of graphs is for a problem-solving method that succeeds, fails, and goes into an infinite loop equally often; that is, the probability of each outcome is 1/3. The middle row of graphs gives the results for a method that succeeds half of the time and fails half of the time, and never goes into an infinite loop. Finally, the bottom row is for a method that succeeds half of the time and loops forever otherwise.

The solid lines show the average per-problem gain up to the current problem; the dotted lines are the selected time bounds; and the dashed lines are the estimates of the optimal bound. The crosses mark the optimal time bounds, and the circles are the expected gains for the optimal bounds.

Note that, when the probability of infinite looping is zero (the middle row), any large time bound gives near-optimal results, because we never need to interrupt a method. Thus, the system never changes the initial time bound and gets near-optimal gains from the very beginning.

Varying the means of time distributions

We now vary the mean value of failure times. We keep the mean success time equal to 20.0 (with standard deviation 8.0). We experiment with failure means of 10.0 (with deviation 4.0), 20.0 (with deviation 8.0), and 40.0 (with deviation 16.0). We give the results in Figure 21.

The gains for normal and log-normal distributions come closer to the optimal values than the gains for uniform and log-uniform distributions. This observation suggests that

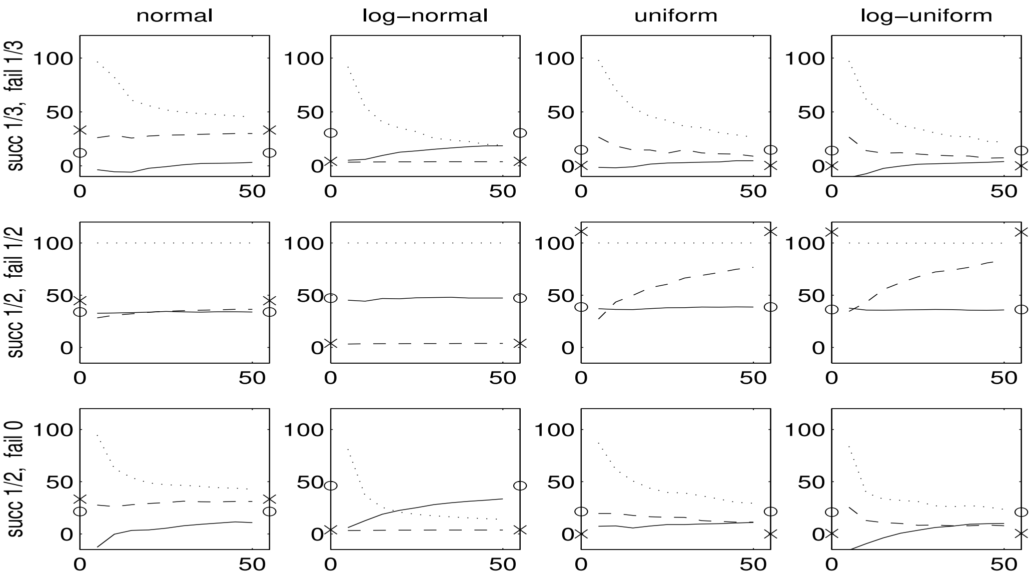


Figure 20: Per-problem gains (solid lines), time bounds (dotted lines), and estimates of the optimal time bounds (dashed lines) for different success and failure probabilities. The crosses mark the optimal time bounds and the circles show the expected gains for the optimal bounds. We give the values of success probability ("succ") and failure probability ("fail") to the left of each row.

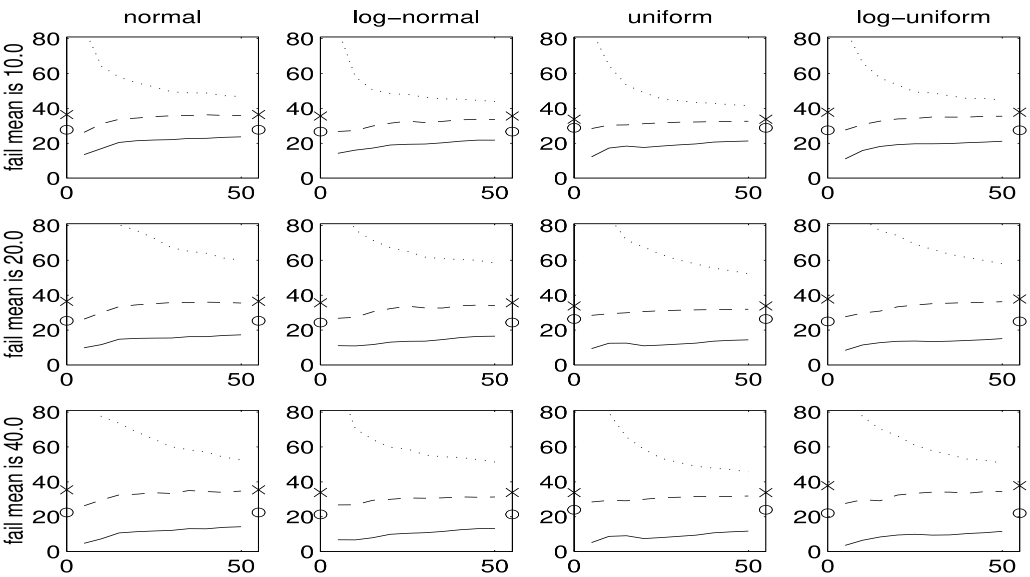


Figure 21: Per-problem gains (solid lines), time bounds (dotted lines), and estimates of the optimal time bounds (dashed lines) for different mean values of failure times. The mean of success times is 20.0 in all experiments.

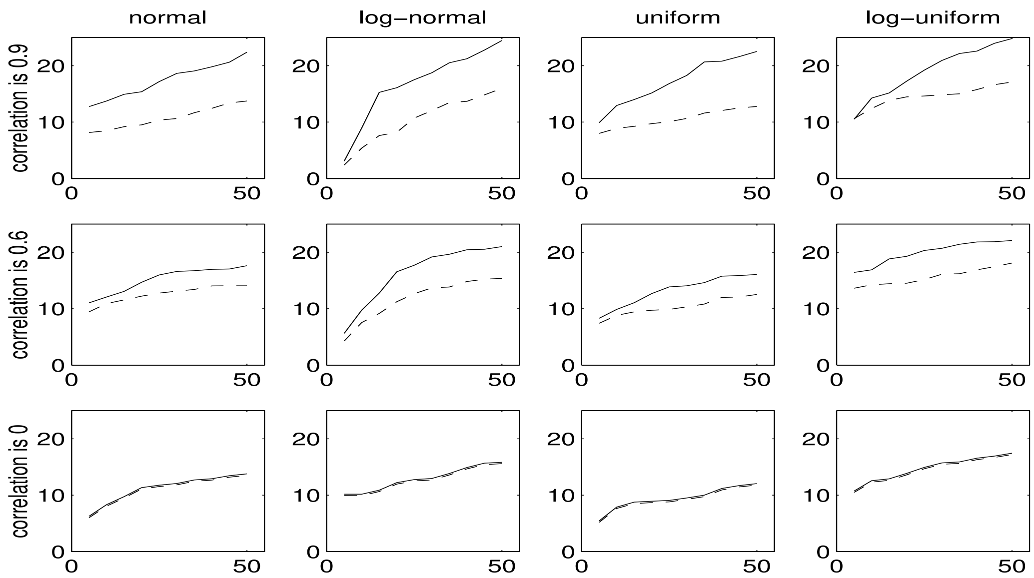


Figure 22: Per-problem gains without the use of sizes (dashed lines) and with sizes (solid lines), for different correlations between size logarithms and time logarithms.

our technique works better for the first two distributions. The difference, however, is not statistically significant.

Use of problem sizes

We compare the gains obtained without and with the use of the regression. Problem sizes in this experiment are natural numbers between 1 and 10, selected randomly. The logarithms of mean success and failure times are proportional to the problem-size logarithms. We adjusted the deviation values to obtain desired correlations between time logarithms and size logarithms. We used the correlation of 0.9 in the first series of experiments and 0.6 in the second series. Finally, we ran experiments with zero correlation; the mean times in this series were the same for all problem sizes.

We give the results in Figure 22, where dashed lines show the average per-problem gains without the regression, and the solid lines give the gains obtained with the regression. The use of the regression improves the performance and the improvement is greater for a larger correlation. If there is no correlation, the system disregards the results of the regression and performs identically without and with sizes.

Method selection

Finally, we show the results of the incremental selection among three problem-solving methods, on 150-problem sequences. In the first series of experiments, we adjusted mean success and failure times in such a way that the optimal per-problem gain for the first method was 10% larger than that for the second method and 20% larger than that for the third method.

We give the results in Figure 23. The top row of graphs shows the average per-problem gain without the use of the regression (dashed lines) and with the regression (solid lines). The circles mark the expected gains for the optimal time bounds, without the regression.

The other two rows of graphs give the probability of choosing each method, for the

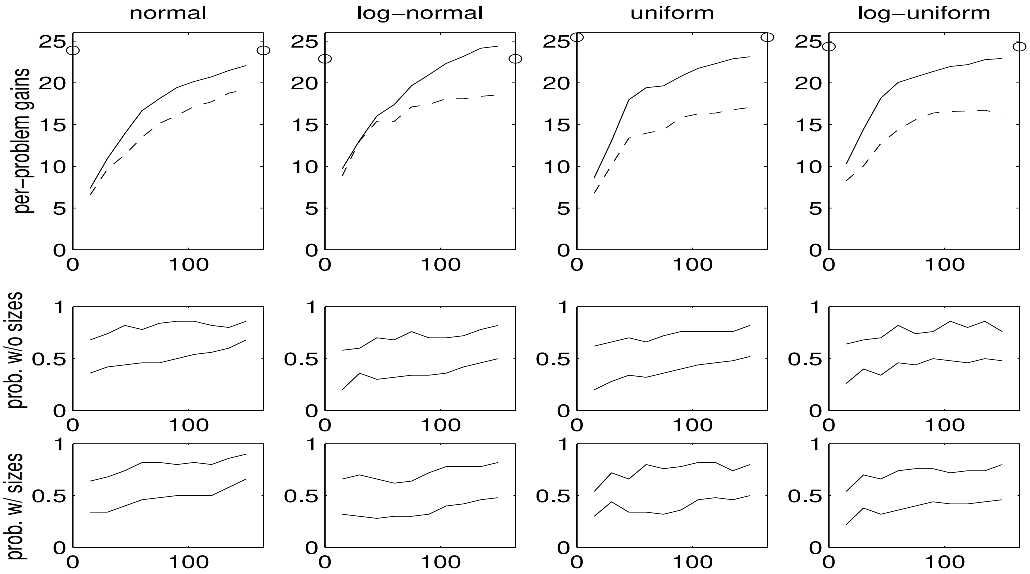


Figure 23: Incremental selection among three problem-solving methods, where the average gain for the first method is 10% larger than that for the second method and 20% larger than that for the third method. We show the average per-problem gains in the experiments without and with the use of the regression (the top row of graphs), and the probability of selecting each method (the other two rows).

experiments without and with the use of problem sizes. The distance from the bottom of the graph to the lower curve is the probability of selecting the first method, the distance between the two curves is the chance of selecting the second method, and the distance from the upper curve to the top is the third method's chance. The graphs show that the probability of selecting the first method (which gives the highest gain) increases in the process of learning. The probability of selecting the third (worst-performing) method decreases faster than that of the second method.

In the second series of experiments, the optimal gain of the first method was 30% larger than that of the second method and 60% larger than that of the third method. We give the results in Figure 24. Note that the probability of selecting the first method grows much faster, due to the larger difference in the expected gains.

9 Conclusions and extensions

We have stated the task of selecting among available problem-solving methods as a statistical problem, derived an approximate solution, and used it to build a system for the automatic selection of the most effective method. The system collects data on the results of using the available methods and estimates their average performance. It uses an approximate measure of problem sizes and information about similarity between problems to improve the accuracy of the estimates. The choice of the method is based on the estimated performance. The selection heuristics combine the exploitation of the past experience with the exploration of new alternatives.

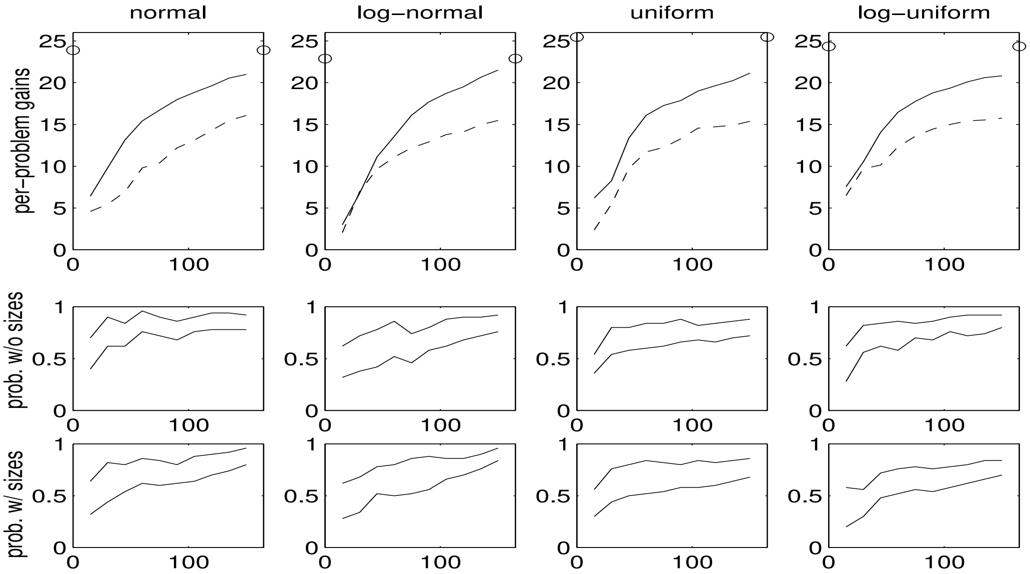


Figure 24: Incremental selection among three problem-solving methods, where the average gain for the first method is 30% larger than that for the second method and 60% larger than that for the third method.

We have demonstrated empirically the system's effectiveness in choosing the right method and a near-optimal time bound. The selection technique has proved effective for a variety of running-time distributions. The technique gives good results even when the distributions do not satisfy the assumptions of the statistical analysis. Its performance, however, depends on the user's proficiency in selecting an accurate measure of problem sizes and defining groups of similar problems.

The generality of the statistical model makes our technique applicable to selection among multiple search methods in any AI system. Besides, the model extends to a wide range of real-life situations outside of AI. The main limitation of applicability stems from the restrictions on the reward function. Another major drawback of the model is its inability to account for specific properties of given problem-solving methods.

We have implemented heuristics that enhance the statistical technique, though we have *not* used them in the described experiments. In particular, we allow the user to provide a prediction of the gains for different methods; we then combine the user's prediction with the statistical estimate. If the selected method has failed to solve a problem, we can choose another method for a second attempt to find a solution. We have designed an algorithm for making this choice of a new method. The algorithm re-evaluates the gain estimates, to incorporate the knowledge that the first attempt has failed. Finally, we provide a mechanism for combining *if-then* preference rules for method selection with our numerical estimates. This combination enables us to merge the user-coded semantic knowledge with the incremental learning.

The statistical model for method selection rises many open problems, which include relaxing our simplifying assumptions, improving the rigor of the statistical derivation, extending the model to account for more features of real-world situations, and improving the heuristics used with statistical estimates.

To make the model more flexible, we need to provide a mechanism for updating the time bound while searching for a solution. We also plan to explore the use of competing problem-solving methods on parallel machines, which will require an extension to the selection technique. Another open problem is to consider possible dependencies of the reward on the solution quality and enhance the model to account for such dependencies. We also need to allow interleaving of several promising methods, which is often more effective than sticking to one method.

To enhance the use of similarity hierarchies, we should allow multiple inheritance among groups and make appropriate extensions to the group-selection heuristics. Finally, we need to provide a means for learning similarity groups automatically, to minimize the deviation of time logarithms (see Figure 11c) within groups.

Acknowledgements

I am grateful to Svetlana Vayner, who contributed many insights into my research. She helped to construct the statistical model for estimating the performance of problem-solving methods and provided a thorough feedback on all my ideas. I owe thanks to Herbert Simon, Jaime Carbonell, Manuela Veloso, Karen Haigh, and Henry Rowley for their valuable comments and suggestions.

References

[Bacchus and Yang, 1992] Fahiem Bacchus and Qiang Yang. The expected value of hierarchical problem-solving. In *Proceedings of the Tenth National Conference on Artificial Intelligence*, 1992.

[Blumer *et al.*, 1987] Anselm Blumer, Andrzej Ehrenfeucht, David Haussler, and Manfred K. Warmuth. Occam's razor. *Information Processing Letters*, 24:377–380, 1987.

[Cohen, 1992] William W. Cohen. Using distribution-free learning theory to analyze solution-path caching mechanisms. *Computational Intelligence*, 8(2):336–375, 1992.

[Cohen, 1995] Paul R. Cohen. *Empirical Methods for Artificial Intelligence*. MIT Press, Cambridge, MA, 1995.

[Gentner and Stevens, 1983] Dedre Gentner and Albert L. Stevens, editors. *Mental Models*, Hillside, NJ, 1983. Lawrence Erlbaum Associates.

[Knoblock, 1991] Craig A. Knoblock. *Automatically Generating Abstractions for Problem Solving*. PhD thesis, School of Computer Science, Carnegie Mellon University, 1991. Technical Report CMU-CS-91-120.

[Knoblock, 1994] Craig A. Knoblock. Automatically generating abstractions for planning. *Artificial Intelligence*, 68:243–302, 1994.

[Mendenhall, 1987] William Mendenhall. *Introduction to Probability and Statistics*. Duxbury Press, Boston, MA, seventh edition, 1987.

[Newell and Simon, 1972] Allen Newell and Herbert A. Simon. *Human Problem Solving*. Prentice Hall, Englewood Cliffs, NJ, 1972.

[Pérez, 1995] M. Alicia Pérez. *Learning Search Control Knowledge to Improve Plan Quality*. PhD thesis, School of Computer Science, Carnegie Mellon University, 1995. Technical Report CMU-CS-95-175.

[Polya, 1957] George Polya. *How to Solve It*. Doubleday, Garden City, NY, second edition, 1957.

[Stone *et al.*, 1994] Peter Stone, Manuela M. Veloso, and Jim Blythe. The need for different domain-independent heuristics. In *Proceedings of the Second International Conference on AI Planning Systems*, pages 164–169, 1994.

[Valiant, 1984] Leslie G. Valiant. A theory of the learnable. *Communications of the ACM*, 27:1134–1142, 1984.

[Veloso and Stone, 1995] Manuela M. Veloso and Peter Stone. FLECS: Planning with a flexible commitment strategy. *Journal of Artificial Intelligence Research*, 3:25–52, 1995.

[Veloso *et al.*, 1995] Manuela M. Veloso, Jaime G. Carbonell, M. Alicia Pérez, Daniel Borrajo, Eugene Fink, and Jim Blythe. Integrating planning and learning: The PRODIGY architecture. *Journal of Experimental and Theoretical Artificial Intelligence*, 7(1):81–120, 1995.

[Veloso, 1994] Manuela M. Veloso. *Planning and Learning by Analogical Reasoning*. Springer Verlag, 1994.