Genetics-Based Learning of New Heuristics: Rational Scheduling of Experiments and Generalization

Benjamin W. Wah, *Fellow, IEEE*, Arthur Ieumwananonthachai, Lon-Chan Chu, and Akiko N. Aizawa, *Member, IEEE*

***Abstract*—In this paper we present new methods for the automated learning of heuristics in knowledge-lean applications and for finding heuristics that can be generalized to unlearned domains. These applications lack domain knowledge for credit assignment; hence, operators for composing new heuristics are generally model-free, domain independent, and syntactic in nature. The operators we have used are genetics-based; examples of which include mutation and cross-over. Learning is based on a generate-and-test paradigm that maintains a pool of competing heuristics, tests them to a limited extent, creates new ones from those that perform well in the past, and prunes poor ones from the pool. We have studied three important issues in learning better heuristics: 1) anomalies in performance evaluation, 2) rational scheduling of limited computational resources in testing candidate heuristics in single-objective as well as multiobjective learning, and 3) finding heuristics that can be generalized to unlearned domains. We show experimental results in learning better heuristics for 1) process placement for distributed-memory multicomputers, 2) node decomposition in a branch-and-bound search, 3) generation of test patterns in VLSI circuit testing, and 4) VLSI cell placement and routing.**

***Index Terms*—Branch-and-bound search, generalization, genetics-based learning, heuristics, knowledge-lean applications, performance evaluation, process mapping, resource scheduling, VLSI circuit testing, VLSI placement and routing..**

I. Introduction

The design of problem solving algorithms for many applications generally relies on the expertise of designers and the amount of domain knowledge available. This design is difficult when there is little domain knowledge or when the environment under consideration is different from which the algorithm is applied. In this paper we study two important problems in designing efficient algorithms: 1) automated design of problem solving heuristics in knowledge-lean application environments, and 2) systematic search of heuristics that can be generalized to unlearned domains.

A *problem solver* (PS) can be optimal or heuristic. An optimal problem solver is a realization of an optimal algorithm that solves the problem optimally with respect to certain *objectives*. In contrast, a heuristic problem solver has components that were designed in an ad hoc fashion, leading to possibly suboptimal solutions when applied. When there is no optimal algorithm, the design of effective heuristics is crucial. Without ambiguity, we simply use "problem solvers" in this paper to refer to "heuristic problem solvers."

Heuristics, in general terms, are "rules of thumb" or "common-sense knowledge" used in attempting the solution of a problem [22]. Newell, Shaw, and Simon defined heuristics as "A process that may solve a given problem, but offers no guarantees of doing so" [20]. Pearl defined heuristics as "Strategies using readily accessible though loosely applicable information to control problem-solving processes in human being and machines" [22]. In this paper, we define a *heuristic method* (HM) to mean a problem solving procedure in a problem solver. Without loss of generality, an HM can be considered as a collection of interrelated *heuristic decision elements* (HDE) or *heuristics decision rules*. As illustrated in Fig. 1, a problem solver takes a problem instance (or test case) and generates a solution. Note that the solution is generally suboptimal since heuristics are used.

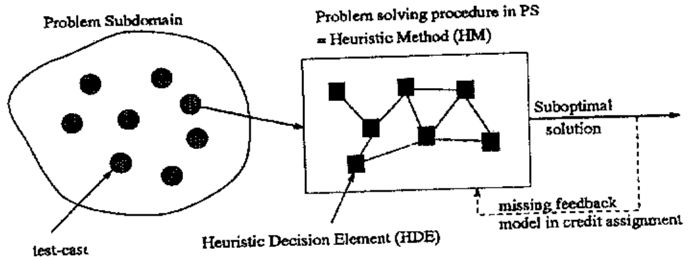

Problem Subdomain
Problem solving procedure in PS
= Heuristic Method (HM)
Suboptimal solution
missing feedback model in credit assignment
test-case
Heuristic Decision Element (HDE)

Fig. 1. A heuristic method applied to a problem instance in a knowledge-lean application domain.

Heuristics are usually designed by experts with strong expertise in the target application domain, or by automated learning systems using machine learning techniques. Both methods focus on explaining the relation between heuristics and their performance, and on generating "good" heuristics based on observed information or explained relations. There are three major issues in designing good heuristics.

1) *Generation of heuristics*. The way that heuristics are generated depends on domain knowledge available in the application environment. An application environment can be knowledge-rich or knowledge-lean with respect to the heuris-

Manuscript received Aug. 10, 1991; revised Dec. 12, 1994.
B.W. Wah and A. Ieumwananonthachai are with the Center for Reliable and High-Performance Computing, Coordinated Science Laboratory, University of Illinois at Urbana-Champaign, 1308 W. Main St., Urbana, IL 61808; e-mail: b-wah@uiuc.edu.
L.-C. Chu is with Microsoft, Inc., One Microsoft Way, Redmond, WA 98052.
A.N. Aizawa is with the National Center for Science Information System, 3-29-1 Otsuka, Bunkyo-ku, Tokyo 112, Japan.
IEEECS Log Number K96066.

tics to be designed. In knowledge-rich domains, a *world model* helps explain the relationship among decision, states, actions, and performance feedback generated by the learning system or measured in the environment. This model is important in identifying good heuristics that otherwise may be difficult to find. In contrast, such models do not exist in knowledge-lean domains. In this case, the heuristics generator cannot rely on performance feedback (or credit assignment as shown in Fig. 1) to decide how new heuristics should be generated or how existing heuristics should be modified. Note that we can use credit-assignment algorithms that do not rely on the world model; however, their effect on performance improvement is weak.

In this paper, we study the learning of heuristics in knowledge-lean application domains. Since credit assignment is difficult in these domains, operators for composing new HMs are usually model-free, domain-independent, and syntactic in nature. Heuristics are generally represented in a form that can be modified syntactically; examples include bit strings and collections of symbols and numbers. The operators we use are based on those in genetics-based learning (such as mutation, cross-over and random generation) that perturb existing parameters and rules in order to obtain new ones [6].

Genetics-based learning, [6] also called population-based learning [1], [31], of HMs is based on a generate-and-test paradigm that maintains a pool of competing HMs, tests them to a limited extent, creates new ones from those that perform well in the past, and prunes poor ones from the pool. It involves the application of genetic algorithms to machine learning problems. Examples of such learning include genetic programming [16] and classifier-system learning [6].

Genetics-based learning is suitable for learning performance-related HMs but not for learning correctness-related ones. An HM is said to be *performance related* if the constraints of the target problem are trivially satisfied, and the goal of learning is to improve the performance of the resulting solution, where performance is characterized by the *quality* of the resulting solution and the *cost* of getting it. In contrast, an HM is *correctness related* if the constraints of the problem are hard to satisfy, and the goal of learning is to find HMs that lead to efficient as well as feasible solutions. In knowledge-lean applications, HMs that are performance related are easier to learn than those that are correctness related. For this reason, we only study performance-related HMs in this paper.

2) *Testing of heuristics and evaluating their performance.* HMs generated are tested on a set of problem instances (or test cases) in a problem domain. In this paper, we are interested in two types of application domains: a) those with a large number of test cases and possibly an infinite number of deterministic HMs for solving them, and b) those with a small number of test cases but the HMs concerned have a nondeterministic component, such as a random initialization point, that allows different results to be generated for each test case. In both types, the performance of an HM is nondeterministic, requiring multiple evaluations of the HM on different test cases (type i) or multiple evaluations of the HM on the same test case (type ii). Consequently, we need to define valid statistical metrics for comparing two HMs without exhaustively testing all test cases using these HMs. This requires identifying subsets of test cases whose collective behavior on an HM can be evaluated statistically. We present in Section II issues on selecting appropriate aggregate metrics in performance evaluation of heuristics.

An important issue in implementing a learning system is the scheduling of finite computational resources for testing a possibly infinite set of test cases and infinitely many variations of HMs. This entails apportioning computational resources to tests so that the best HM is found when resources are expended. The problem is especially difficult when tests are expensive and noisy. (The latter means that multiple tests are necessary in order to determine the performance of an HM.) We study in Section IV the scheduling of computational time for learning.

3) *Generalization of heuristics learned to unlearned domains.* Since the problem space is very large and learning can only cover a small subset, it is necessary to generalize HMs learned to test cases not studied in learning. Generalization is difficult when HMs do not perform consistently or have different performance distributions across different test cases. Section V examines issues in generalization.

In short, we study in this paper the automated learning and generalization of performance-related heuristics by genetics-based learning methods for knowledge-lean applications. We assume that the performance of an HM is represented by one or more statistical metrics and is based on evaluating multiple test cases (noisy evaluations). The major issues that we study are methods to cope with inconsistencies in performance evaluation of heuristics (Section II), resource scheduling of tests of heuristics (Section IV), and generalization of learned heuristics to unlearned domains (Section V). Section III presents the overall learning framework, and experimental results are shown in Section VI.

II. Heuristics

In applying an HM, a problem solver applies a sequence of decisions defined in the HDEs of the HM, one after another, until an input test case is solved. These decisions, initiated by the problem solver at decision points, change the state of the application environment evaluated by a number of user-defined performance *measurables*. The problem solver then uses the performance measured to make further decisions.

A *solution* in this context is defined as a sequence of decisions made by the HM on an input test case in order to reach the final state.

The *performance* of an HM on a test case depends on the *quality* of the solution found by the HM for this test case as well as the *cost* (e.g., computation time) in finding the solution. Here, we define *quality* (resp., *cost*) of a solution with respect to an input test case to be one or more measures of how good the final state is (resp., how expensive it is to reach the final state) when the test case is solved, and be independent

TABLE I
EXAMPLES OF KNOWLEDGE-LEAN APPLICATIONS AND THEIR LEARNABLE DOMAIN-DEPENDENT HEURISTICS

Application	Objective(s)	Domain-Dependent Parametric Heuristics	Heuristic Elements	Example(s) of Element
Process mapping for placing a set of processes on a multi-computer [14]	Minimize overall completion time, and minimize time to find such mappings	If (processor utilization/average utilization of all processors) > (threshold), then evict one process	Numeric threshold value	1.10
Load balancing in distributed systems [19]	Minimize completion time of an incoming job	If (average WL(•) > (threshold)), then migrate this process	Workload function WL, numeric threshold value	WL(•), 2.0
Simulated annealing: TimberWolf [26]	Minimize area of layout with fixed maximum number of layers	If ((acceptance ratio) > (threshold)), then reduce temperature to next lower level	Numerical threshold value, cost function, temperature function	0.9, C(•), T(•)
Depth perception in stereo vision [25]	Minimize error in range estimation	Marr and Poggio's iterative algorithm	(low edge-detection threshold, channel width, high threshold)	(0.6, 2.0, 5.0)
Genetic search of the best VLSI test sequence [24]	Maximize fault coverage	Controls used in the genetic algorithm: iteration, rejection ratio, sequence depth, control factor, frequency of usage	Numeric values, fitness function	(2, 3, 42, 3.2, 100), H(•)
Branch-and-bound search for finding a minimum-cost tour in a graph	Minimize cost of tour while satisfying constraint on visiting each node exactly once	If a node has the smallest decomposition-function value among all active nodes, then expand this node	Symbolic formula	Lower bound + upper bound of node
Designing a blind equalizer	Minimize convergence time, accumulated errors and cost; maximize S/N ratio	Object (error) function for gradient descent	Symbolic fomula of the error function	E(•)

of the intermediate states reached. Note that cost and quality are in turn defined as functions of measurables in the application environment. We call quality and cost *performance measures* of an application.

In this section, we discuss issues related to the performance evaluation of HMs. We show that an HM can be found to be better or worse than another HM depending on the evaluation criterion. Such inconsistencies are called *anomalies* in this paper and are attributed to the different methods of evaluating performance and the different behavior of HMs under different conditions. We propose methods to cope with these anomalies. When such anomalies cannot be avoided, alternative HMs should be learned and generalized so that users can pick the best HM(s) to apply.

A. Example Applications

A problem solver in general consists of a domain-independent part and a domain-dependent part. The domain-independent part is a general solution method that is applicable across different applications. For example, a divide-and-conquer method is domain-independent because it can be applied to many different applications. In contrast, the domain-dependent part is specific for a particular application. For example, the mechanism of partitioning a problem in a divide-and-conquer method is domain-dependent.

The domain-independent and domain-dependent parts interact with each other to make decisions during the solution process. The domain-dependent part provides information on the current state to the domain-independent part, which returns a decision according to the information provided. The domain-dependent part then applies the decision to change the state of the application environment.

Heuristics can be used in the domain-dependent part to improve the solution cost or the solution quality (or both). In Table I, we present examples of practical applications and identify their domain-dependent heuristics.

B. Problem Subspace and Subdomain

In an application domain, different regions of its problem space may have different characteristics, each of which can best be solved by a unique HM [23]. Since learning is difficult when test cases are of different behavior and it is necessary to compare HMs quantitatively, we need to decompose the problem space into smaller partitions before learning begins. In the following, we define the concepts of problem subspace and problem subdomain.

A *problem subspace* is a user-defined partition of a problem space so that HMs for one subspace are learned independently of HMs in other subspaces. Such partitioning is generally guided by common-sense knowledge or by user experience in solving similar problems. To identify a problem subspace, we need to know one or more attributes to classify test cases and a set of decision rules to identify the subspace to which a test case belongs.

For instance, consider the development of HMs for CRIS [24], a genetic-algorithm package for generating test patterns in order to test sequential VLSI circuits. (More details are provided in Section VI.C.) CRIS aims to find a test sequence in order to discover as many faults as possible in a circuit. (Fault coverage measures the percentage of faults that can be detected by a test pattern generated by CRIS.) Previous experience shows that sequential circuits require tests that are different from those of combinatorial circuits. As a result, we can partition the problem space into two subspaces, one for combinatorial circuits and the other for sequential circuits.

As another example, consider solving a vertex-cover problem whose goal is to find a minimal set of vertices in a graph such that all edges have at least one of their vertices covered by this set. In designing a decomposition HM in a branch-and-bound (B&B) search for deciding which vertex to be included in the covered set, previous experience on other optimization problems indicates that HMs for densely connected graphs are generally different from HMs for sparsely connected ones. Consequently, the problem space of all graphs may be partitioned into two subspaces, one for tightly connected graphs and one for loosely connected ones.

Given a subspace of test cases, we next define a subdomain. A *problem subdomain* is a partitioning of a problem subspace into smaller partitions so that one or more HMs can be designed for each partition. The reason for this partitioning is to allow quantitative comparison of performance of HMs in a subdomain, which may be difficult across subdomains. In comparing the performance of HMs, it is necessary to aggregate their performance values into a small number of performance metrics (such as average or maximum). Computing these aggregate metrics is not meaningful when performance values are of different ranges and distributions. Hence, we define a subdomain as a maximal partitioning of test cases in a subspace so that different HMs in a subdomain can be compared quantitatively based on their aggregate metrics. It is important to point out that performance values may need to be normalized with respect to those of the baseline HM before aggregated.

In the same way that test cases are partitioned into subspaces, we need to know the attributes to classify test cases and a set of decision rules to identify the subdomain to which a test case belongs. For example, in solving a vertex-cover problem, we can treat graph connectivity as an attribute to classify graphs into subdomains.

In some applications, it may be difficult to determine the subdomain to which a test case belongs. This is true because the available attributes may not be well defined or may be too large to be useful. For instance, in test-pattern generation for sequential circuits, there are many attributes that can be used to characterize these circuits (such as length of the longest path and maximum number of fan-ins and fan-outs). However, none of these attributes is a clear winner. When we do not know the attributes to classify test cases into subdomains, we can treat each test case as a subdomain by itself. This works well when the HM to be learned has a random component: by using different random seeds in the HM, we can obtain statistically valid performance values of the HM on a test case. We have used this approach in the two circuit-related applications discussed in Section VI and have chosen each circuit as an independent subdomain for learning. Another possibility is to learn one HM for each subdomain, but apply multiple HMs when a new circuit is encountered.

After learning good HMs for each subdomain, we need to compare the performance of HMs across subdomains. This comparison may be difficult because test cases in different subdomains of a subspace may have different performance distributions, even though they can be evaluated by a common HM. As a result, it may be difficult to compare the performance of test cases statistically.

It should now be clear that there can be many subdomains in an application, and learning can only be performed on a small number of them. Consequently, it is important to generalize the HMs learned to unlearned subdomains. Informally, generalization entails finding a good HM from the set of learned HMs so that this HM has a high probability of performing better than other competing HMs for solving a randomly chosen test case in the subspace. In some situations, multiple HMs may have to be identified and applied together at a higher cost to find a solution of higher quality. In Section V, we propose a method for generalizing learned HMs.

To illustrate the importance and difficulty in generalization, consider an HM developed for the previously described CRIS [24]. An HM in this application is a vector of seven parameters and a random seed used in the genetic algorithm in CRIS. As mentioned earlier, we treat each circuit as a separate subdomain in learning because we do not know the attributes to group circuits in subdomains. Note that different fault coverages can be obtained for a circuit by varying the random seed used in an HM. Table II shows the maximum and average fault coverages (over ten random seeds) of an HM we have learned and generalized for CRIS across six circuits. It shows that a) the HM behaves differently across different circuits—not only is the range of fault coverages different, but it may perform better than CRIS [24] and HITEC (a program that uses a deterministic search to find good patterns) [21] for one circuit, but worse for another; b) multiple applications of the same HM using different random seeds can improve the coverage; and c) there are limitations in CRIS that may render it difficult to improve over HITEC.

TABLE II
FAULT COVERAGES OF A LEARNED AND GENERALIZED HM USED IN CRIS ON SIX CIRCUITS AS COMPARED TO FAULT COVERAGES OF THE ORIGINAL CRIS [24] AND HITEC [21]

Circuit ID	Total Faults	Fault Coverage		Generalized HM for CRIS	
		HITEC	CRIS	Avg.	Max.
s344	342	95.9	93.7	96.1	96.2
s382	399	90.9	68.6	72.4	87.0
s641	467	86.5	85.2	85.0	86.1
s832	870	93.9	42.5	44.1	45.6
s1238	1355	94.6	90.7	88.2	89.2
s5378	4603	70.3	65.8	65.3	69.9

As another example, we show in Table III the results of learning and generalization of decomposition HMs used in a B&B search for solving vertex-cover problems. Here, we treat all test cases to belong to one subspace, and graphs with the same average degree of connectivity are grouped into a subdomain. We applied genetics-based learning (to be discussed in Section III) to find two HMs, one for each of two subdomains with connectivities 0.1 and 0.5. We then applied our generalization procedure (to be discussed in Section V) to find one HM that can generalize across the two subdomains. Finally, we verified the speedups of the generalized HM on six subdomains. The results in Table III show that the generalized HM is not the top HM learned in each subdomain, indicating that the best HM learned in each subdomain may be too specialized to the subdomain. Further, we have found one HM that performs better than the baseline HM across the six subdomains.

TABLE III
AVERAGE SPEEDUPS OF LEARNED HMS AND GENERALIZED HMS USED IN A B&B SEARCH FOR SOLVING THE VERTEX-COVER PROBLEM

Subdomain	Speedups of HM Learned		Generalized
DC	DC = 0.1	DC = 0.5	HM
0.1	1.035	0.993	1.260
0.2	0.950	1.001	1.086
0.3	1.012	0.988	1.074
0.4	1.043	0.986	1.106
0.5	0.993	1.013	1.009
0.6	0.997	1.012	1.042

All speedups are normalized with respect to those of the baseline HM. Subdomains are classified by degree of connectivity—DC.

C. Anomalies in Performance Evaluation

To design a good and general HM for an application, we must compare HMs in terms of their performance. There are two steps in accomplishing this task. First, we must compare HMs in the same subdomain [31]. Second, we must compare the performance of HMs across multiple subdomains. Accomplishing the first step is necessary before we can deal with the second step. In this section, we present issues involved in these two steps.

C.1. Anomalies within a Subdomain

Recall that HMs studied in this research have nondeterministic performance, implying the need to evaluate each HM multiple times in a subdomain. Further, performance may be made up of multiple inter-related measures (for instance, higher quality may require higher cost).

To compare the performance of different HMs, it is necessary to aggregate performance values before comparing them. This is, however, difficult, as the objectives of an HM as well as their trade-offs may be unknown with respect to its performance measures. A possible solution is to derive a single objective function of the performance measures with tunable parameters, and to find a combination of values of these parameters that lead to the HM with the best trade-off. Using this approach, we have observed the following difficulties before [31].

- It is difficult to find a good combination of parameter values in the objective function so that HMs with the best quality-cost trade-offs can be found. We have seen similar difficulties in the goal attainment method [12].
- It is difficult to compare the performance of two HMs when they are evaluated on test cases of different sizes or behavior.
- Inconsistent conclusions (anomalies) about the performance of two HMs may be reached when they are compared using either different user-defined objective functions, or the same objective function with different parameters. In fact, it is possible to show that one HM is better than another by finding a new parametric objective function of the performance measures.

We have proposed before [31] three solutions to cope with these difficulties.

1) Identify a *reference* or *baseline* HM upon which all other HMs are compared. A good choice for an application is the best existing HM for this application.
2) Normalize each raw performance measure of a new HM with respect to the same measure of the reference HM (evaluated on either the same set of test cases or test cases with the same distribution of performance) so that it is meaningful to compare two HMs based on their normalized measures.
3) Compare two HMs based on individual normalized performance measures, not on a single parametric function of the measures. We have proposed before a multi-dimensional graphical representation of performance values, representing each performance measure in a separate axis [15] Two HMs are, therefore, compared based on their relative positions in this multi-dimensional plot. (This method is discussed later in Section VI.A.)

In this section, we extend the anomalies found earlier [31] and classify all the anomalies into three classes. Note that anomalies happen because there is more than one dimension of performance variations.

1) *Inconsistent performance across different test cases.* When an HM is evaluated on a set of test cases, we must determine a) the number of tests to be made and b) the evaluation method (or metric) for aggregating performance values (such as mean, maximum, average rank). Inconsistent conclusions may be reached when one HM is better than another on one set of test cases, but worse on a different set of test cases.

TABLE IV
INCONSISTENT PERFORMANCE OF HMS ACROSS TEST CASES

Quality	Test Case j			Average
Measure	1	2	3	
$t_{1,j}$	1474.89	1665.38	1381.34	$\bar{t}_1 = 1507.20$
$t_{2,j}$	1269.25	1513.14	1988.42	$\bar{t}_2 = 1590.27$

Let $t_{i,j}$ be the completion time of test case j using HM_i. HM_2 has better average rank and better average completion time than HM_1 after two tests, but worse average completion time and better average rank after three tests.

For example, assume that performance is evaluated by either the average metric or the average-rank metric. Each of these metrics may improve or degrade when more tests are done, changing the ordering of HMs. Table IV illustrates this phenomenon using HMs developed for Post-Game Analysis (PGA) (to be discussed in Section VI.A). These HMs were used to map a collection of communicating processes on a network of computers. This example shows that different conclusions can be drawn depending on the performance metrics used.

In general, we must first select a method for aggregating performance values of each measure. This can usually be determined from the application requirements. In this paper, we use the average metric as the primary method for comparing HMs, assuming that performance values of tests in a subdomain are i.i.d. In addition, we must examine the actual distribution obtained in the experiments, since the average metric alone is not enough to show the spread of performance values. HMs that have good average behavior but have large spread in performance are not desirable. On the other hand, when the metric is unknown, alternative HMs should be found for different metrics, and users can select the appropriate HM(s) to use.

2) *Multiple objectives with unknown trade-offs*. The performance of an HM may be evaluated by multiple objectives (such as quality and cost). Of course, we would like to find HMs with improved quality and reduced cost. However, this may not always be possible. The problem, generally known as the *multiobjective optimization problem* [12], involves tradeoffs among the objectives.

In evaluating HMs with multiple objectives, we must evaluate them based on individual performance measures and not combine multiple measures into a single parametric function [31]. During learning, the learning system should constrains *all but one* measures and optimize the single unconstrained measure. An HM is pruned from further testing when one of its performance constraints is violated. The goal is to find an HM that satisfies all the constraints and has the best performance in the unconstrained measure. If a good HM satisfying the constraints can be found, then the constraints are further refined, and learning is repeated (see Section V.B). A similar approach has been used in MOGA, a multiple objective genetic algorithm [11]. The difficulty with this approach is on setting constraints. We study in this paper the case when there are two performance measures. The general case in which there are more than two performance measures is still open at this time

3) *Inconsistencies in normalization*. Normalization involves choosing a baseline HM and computing relative performance values of a new HM on a set of test cases in a subdomain by the corresponding performance values of the baseline HM. This is necessary when performance is assessed by evaluating multiple test cases (type i as discussed in Section I), and is not needed when nondeterminism in performance is due to randomness in the problem solver (type ii as discussed in Section I). In the former case, performance from different tests may be of different ranges and distributions, and normalization establishes a reference point in performance comparison. In the latter, raw performance values within a subdomain are from one test case and presumably have the same distribution.

Normalization may lead to inconsistent conclusions about the performance of HMs when multiple normalization methods are combined. This anomaly is illustrated as follows.

EXAMPLE A. Referring to Table IV, if we use HM_1 as the baseline for normalization, we can compute the average normalized speedup of HM_2 by one of the following methods:

$$\overline{q}_2^n = \sum_{j=1}^{3} \frac{t_{1,j}}{t_{2,j}} = 0.986; \quad \overline{Q}_2^n = \sum_{j=1}^{3} \frac{t_{1,j} - 1200}{t_{2,j} - 1200} = 1.900. \quad (1)$$

Since the average normalized speedup of HM_1 is one, HM_2 is found to be worse using the first methods and better using the second. □

Inconsistencies may also occur when normalization overemphasizes or deemphasizes performance changes. For instance, the speedup measure is biased against slowdown (as slowdowns are in the range between 0 and 1, whereas speedups are in the range between 1 and infinity). Consider the following example.

EXAMPLE B. Suppose the speedups of an HM on two test cases are 10 and 0.1. Then the average speedup is 5.05, and the average slowdown is also 5.05, where the average slowdown is defined as the average of the reciprocals of speedup. Hence, the average speedup and average slowdown are both greater than one. □

In general, when normalizing performance values, it is important to note that the ordering of HMs may change when using a different normalization method, and that the spread of performance values may vary across subdomains in an application. Here, we propose three methods to cope with anomalies in normalization. First, we should use only one normalization method consistently throughout learning and evaluation, thereby preserving the ordering of HMs throughout the process. Second, we need to evaluate the spread of normalized performance values to detect bias. This can be done by detecting outliers and by examining higher-order moments of the performance values. Third, to avoid placing unequal emphasis on normalized values, we need a normalization method that gives equal emphasis to improvement as well as to degradation.

To simplify understanding, we describe this symmetric normalization method using the speedup measure. We define *symmetric speedup* as

$$Speedup_{symmetric} = \begin{cases} Speedup - 1 & \text{if } Speedup \geq 1 \\ 1 - \dfrac{1}{Speedup} & \text{if } 1 > Speedup \geq 0 \end{cases} \quad (2)$$

where *Speedup* is the ratio of the time of the original HM with respect to the time of the new HM. Note that slowdown is the reciprocal of speedup, and that symmetric speedup is computed for each pair of performance values. Equation (2) dictates that speedups and slowdowns carry the same weight: speedups are in the range from zero to infinity, and slowdowns are in the range from zero to negative infinity.

In a similar way, we can define *symmetric slowdown* as

$$Slowdown_{symmetric} = \begin{cases} Slowdown - 1 & \text{if } Slowdown \geq 1 \\ 1 - \dfrac{1}{Slowdown} & \text{if } 1 > Slowdown \geq 0 \end{cases} \tag{3}$$

It is easy to prove that $Speedup_{symmetric} = -Slowdown_{symmetric}$, thereby eliminating the anomalous condition in which average speedup and average slowdown are both greater than one or both less than one.

In Example A discussed earlier, the average symmetric speedup is –0.059 which shows that HM_2 is worse than HM_1. In Example B, both the average symmetric speedup and average symmetric slowdown are zero, hence avoiding the anomaly where the average speedup and average slowdown are both greater than one.

To further illustrate the difference between speedups and symmetric speedups, we show in Fig. 2 the distributions of speedups as well as symmetric speedups of an HM to solve the vertex-cover problem.

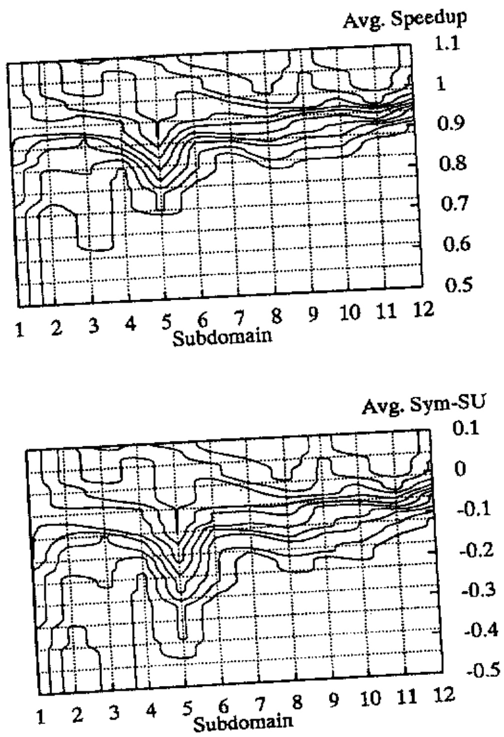


Fig. 2. Contour plots showing the distribution of performance values of one HM on 15 test cases for solving the vertex-cover problem.

C.2. Anomalies Across Subdomains

We now discuss the difficulty in comparing performance of HMs across multiple subdomains. This comparison is difficult when there is a wide discrepancy in performance across subdomains.

To illustrate this point, consider two HMs learned for CRIS (Table V). These HMs behave differently in different subdomains: not only can the range and distribution of performance values be different, but a good HM in one subdomain may not perform well in another. With respect to circuit *s*444, HM_{101} has worse fault coverages and a wider distribution of coverage values than HM_{535}, but performs better than HM_{535} for circuit *s*1196.

TABLE V
INCONSISTENT HM BEHAVIOR IN VARIOUS SUBDOMAINS

Circuit ID	HM	Fault Coverage (%)				
		Random Seeds used in HM			Max.	Avg.
		61801	98052	15213		
s444	101	60.3	13.9	11.2	60.3	28.5
	535	81.9	86.3	86.3	86.3	84.8
s1196	101	93.2	94.4	94.9	94.9	94.2
	535	93.2	92.5	93.6	93.6	93.1

The major difficulty in handling multiple subdomains is that performance values from different subdomains cannot be aggregated statistically. For instance, it is not meaningful to find the average fault coverage of HM_{101} in Table V. Scaling and normalization of performance values are possible ways to match the difference in distributions, but will lead to new inconsistencies for reasons discussed in 3) in the last subsection. Another way is to rank HMs by their performance values across different subdomains, and use the average ranks of HMs for comparing HMs. This does not work well because it does not account for actual differences in performance values, and two HMs with very close or very different performance may differ only by one in their ranks. Further, the maximum rank of HMs depends on the number of HMs evaluated, thereby biasing the average ranks of individual HMs.

To address this problem, we propose in Section V a new metric called probability of win. Informally, the probability of win is a range-independent metric that evaluates the probability that the *true mean performance* of an HM in one subdomain is better than the true mean performance of another randomly selected HM in the same subdomain. The advantage of using this measure is that it is between zero and one, independent of the number of HMs evaluated and the range and distribution of performance values.

III. TEACHER: A SYSTEM FOR LEARNING NEW HEURISTICS

In this section, we discuss TEACHER, an acronym for TEchniques for the Automated Creation of HEuRistics. TEACHER is a genetics-based learning system we have developed in the last six years [31]. Preliminary designs of TEACHER have been studied with respect to learning process-placement strategies for a network of workstations [19], learning process-placement strategies on distributed-memory multicomputers [15], tuning parameters in a stereo-vision algorithm [25], finding smaller feed-forward neural networks [29], and learning heuristics for B&B search [18], [33]. We have also studied resource scheduling strategies in genetics-based learning algorithms.

Our present learning system is aimed toward methods of coping with anomalies in performance evaluation, general resource scheduling strategies in multiobjective learning, and finding HMs that can be generalized. By combining the following three features, our system is unique as compared to other genetics-based learning studies.

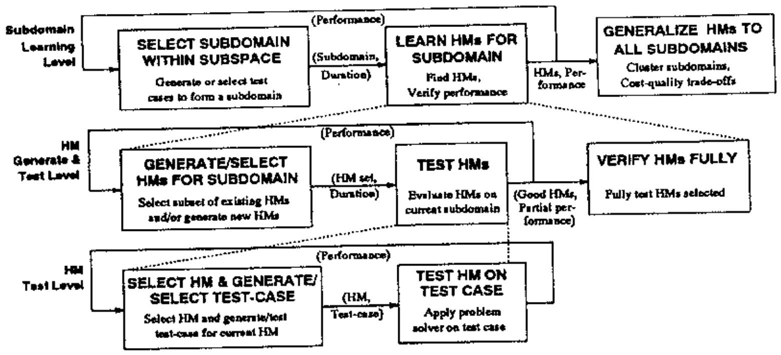

Subdomain Learning Level
SELECT SUBDOMAIN WITHIN SUBSPACE
Generate or select test cases to form a subdomain
(Performance)
(Subdomain, Duration)
LEARN HMs FOR SUBDOMAIN
Find HMs, Verify performance
HMs, Performance
GENERALIZE HMs TO ALL SUBDOMAINS
Cluster subdomains, Cost-quality trade-offs
HM Generate & Test Level
GENERATE/SELECT HMs FOR SUBDOMAIN
Select subset of existing HMs and/or generate new HMs
(Performance)
(HM set, Duration)
TEST HMs
Evaluate HMs on current subdomain
(Good HMs, Partial performance)
VERIFY HMs FULLY
Fully test HMs selected
HM Test Level
SELECT HM & GENERATE/ SELECT TEST-CASE
Select HM and generate/test test-case for current HM
(Performance)
(HM, Test-case)
TEST HM ON TEST CASE
Apply problem solver on test case

Fig. 3. The learning and generalization phases in TEACHER.

- Our environment is noisy so that the performance of an HM cannot be evaluated using a single test.
- We consider applications in which HMs behave differently under different situations (subdomains). Existing methods generally ignore this problem and focus on only one set of statistically related test cases.
- We assume that the cost of evaluating an HM on a test case is expensive. This forbids performing extensive tests on each HM. In the applications presented in Section VI, one to two thousand tests are what can be performed in a few days on a fast workstation. This is in contrast to many other studies that assume that tests are inexpensive and that many tests can be performed in the time allowed [16]. For simplicity, we consider logical time in this paper in which one unit of time is needed for performing one test of an HM.

The goal of resource scheduling is to learn, under limited computational resource, good HMs for solving application problems and to generalize the HMs learned to unlearned subdomains. We use the average metric for comparing HMs, and examine the spread of performance values when HMs have similar average performance. When there are multiple objectives in comparing HMs, we constrain all but one objectives during learning, and optimize the unconstrained objective. In this case, our learning system proposes more than one HMs, showing trade-offs among these objectives.

A. Three Phases of Learning in TEACHER

There are three phases of learning in TEACHER: classification, learning, and generalization. The first phase partitions the test cases in an application into distinct subsets. There are two steps in this phase.

1) *Subspace classification.* As is discussed in Section II.B, the problem space is first partitioned into a small number of distinct subspaces so that new HMs are learned/designed for each. Such partitioning is guided by common-sense knowledge expressed in the form of decision rules. By applying these rules, we can determine for a new test case the subspace it belongs to.
2) *Subdomain classification.* For a problem subspace, we need to partition it into subdomains so that the performance of HMs in each subdomain can be represented collectively by some meaningful statistical metrics. As we have seen in Sections II.B and II.C, the performance of HMs may not be comparable across subdomains in a learning experiment.

In the *learning phase*, the goal is to find effective HMs for each of a limited set of subdomains. The tasks in this phase and the generalization phase are shown in Fig. 3. To perform learning, the system first selects a subdomain, generates good HMs (or uses existing HMs) for this subdomain, and schedules tests of the HMs based on the available computational resources. When learning is completed, the resulting HMs need to be fully verified, as HMs obtained during learning may not be tested adequately. Note that learning is performed on one subdomain at a time. There are three key issues in this phase.

1) *Heuristics generation.* This entails the generation of good HMs given the performance of "empirically good" HMs. As is discussed in Section I, we use weak generation operators here [6], [16].
2) *Performance of HMs in a subdomain.* This problem is related to the performance evaluation of HMs during learning, given that there may be multiple performance measures, and that there is no defined relationship among them (Section II.C.1).
3) *Resource scheduling.* The issues here are on the selection of HMs for further testing, the termination of the current generation, and the initiation of the next generation, given performance information of HMs under consideration. These problems are important when limited computational resources are available and tests of HMs are expensive and noisy. We schedule computational resources rationally by choosing a) number of tests on each HM, b) number of competing HMs to be maintained at any time, and c) number of problem subdomains to be used for learning and for generalization. We study in Section IV two related problems in resource scheduling: *sample allocation* and *duration scheduling*.

The last phase is the *generalization phase* whose goal is to find an HM from the set of learned HMs and see if it has the same high level of performance improvement on unlearned subdomains. There are two key issues here.

1) *Performance of HMs across subdomains.* As is discussed in Section II.C, HMs may have different distributions of

performance values in different subdomains; hence, these values cannot be compared directly. We present in Section V a method to evaluate the performance of HMs for a group of subdomains.

2) *Cost-quality trade-offs*. This involves determining efficient HMs that perform well in the application. Should there be multiple HMs to be applied (at a higher total cost and better quality), or should there be one HM that is costly to run but generates high-quality results? These issues are studied in Section VI when we present experimental results on learning new HMs for four applications.

B. Architecture of Learning System for One Subdomain

Fig. 4 shows the architecture of our resource-constrained learning system for one subdomain [31]. There are five main components in the system:

1) *Resource Scheduler*, which decides the best way to use the available resources,
2) *Internal Critic*, which provides feedback based on the performance measured to indicate how well a particular HM has performed,
3) *Population-Based Learning Element*, which generates new HMs and maintains a pool of existing ones and their past performance,
4) *Test-Case Manager*, which generates and maintains a database of test cases used in HM evaluation, and
5) *Problem Solver*, which evaluates an HM using a test case.

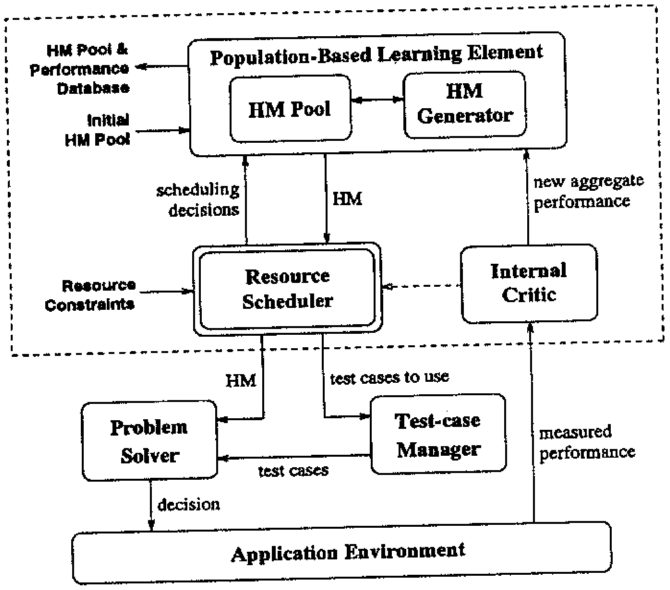


Fig. 4. Architecture of learning system for one subdomain.

In this paper, we assume that the application-specific problem solver and test-case manager are user-supplied. In our current implementation, the test-case manager selects from a user-supplied pool of test cases.

The internal critic normalizes the performance value of each test case tested by a candidate HM against the performance value of the same test case evaluated by the baseline HM. It then updates the performance metrics of the candidate HM. Note that this is similar to updating the fitness values of HMs in classifier-system learning.

In general, the internal critic performs credit assignment [28] that apportions credit/blame on HDEs using results obtained in testing (see Fig. 1). Credit assignments can be classified into temporal credit assignment (TCA) and structural credit assignment (SCA). TCA is the first stage in the assimilation of feedback and precedes SCA during learning. It divides up feedback between current and past decisions. Methods for TCA depends on whether the state space is Markovian: Non-Markovian representations often require more complex TCA procedures. On the other hand, SCA translates the (temporally local but structurally global) feedback associated with a decision point into modifications associated with various parameters of the decision process.

In knowledge-lean applications we consider in this paper, we are missing a world model that relates states, decisions, and feedback signals generated by the learning system or measured in the environment. As a result, credit assignment has much weaker influence on performance improvement. An example of such a TCA algorithm is the bucket-brigade algorithm in classifier-system learning [6]. Note that the lack of a world model for credit assignment is the main reason for maintaining competing HMs in our learning system.

The Resource Scheduler schedules tests of HMs based on the available resources. Note that scheduling is critical when tests are computationally expensive. Two related problems, sample allocation and duration scheduling, as well as the scheduling of tests under multiple performance objectives, are studied in the next section.

IV. Scheduling Tests in Genetics-Based Learning

Resource scheduling of tests in learning is crucial when tests are expensive. To illustrate the importance of scheduling, consider the testing of HMs in the vertex-cover problem discussed in Section II.B. Suppose we have identified two subdomains: D_A (with graph connectivity of 0.1) and D_B (with graph connectivity of 0.6). To illustrate the effect of scheduling, we generated randomly 100 decomposition HMs and evaluated each on D_A and D_B. Table VI shows the average symmetric speedups of HMs selected under three resource schedules with respect to those of the conventional HM. The results show that a) there are trade-offs between the number of HMs tested and the performance of the best HM found, and b) more detailed evaluation of several top HMs at the end of learning is beneficial.

In this section, we discuss our model and assumptions on the sample-allocation and duration-scheduling problems, issues on designing resource scheduling strategies, and our proposed scheduling algorithms.

TABLE VI
SYMMETRIC SPEEDUPS OF THE BEST HMS BASED ON THREE DIFFERENT SCHEDULES AND 150 TESTS

Schedule	Sub-domain	Sym-Speedup of the Best HM		
		Run 1	Run 2	Run 3
10 HMS of 15 tests each	D_A	–0.56	–2.08	0.01
	D_B	–0.22	–0.16	–0.01
75 HMs of 2 times each	D_A	0.01	0.01	0.01
	D_B	–0.03	–0.03	–0.03
20 HMs of 5 tests each; the 5 best HMs 10 times each	D_A	0	–0.56	0.01
	D_B	0	–0.16	–0.01

HMs in each run are selected randomly from a pool of 100 HMs. The average speedup is evaluated using 15 randomly generated test cases.

A. Model and Assumptions

We describe in this section a statistical model for scheduling tests in our learning system. A general comprehensive model is too complex to be analyzed since many parameters are unknown a priori. Here, we find good scheduling strategies based on a simplified model and apply these strategies as heuristics in practice.

We assume that the performance values of HM_i over a problem subdomain constitute a population with a distribution $f_i(x)$. Each evaluation of HM_i is equivalent to drawing a performance value from the distribution. We make the following assumptions in our study.

- In multiobjective applications, we assume that there are $k + 1$ performance metrics $\{C_1, ..., C_k, Q\}$, where $\{C_i, i = 1, .., k\}$ are *constrained metrics*, and Q is an *unconstrained metric* (see Section II.C). We define c_i, $\hat{c}_i$, and θ_i to be, respectively, the expected value, the approximated mean value, and the maximum acceptable expected value for constrained metric C_i. Here, $f(x)$ is the distribution of performance metric Q. We assume that Q for test cases in the subdomain is to be maximized, while the other performance metrics must satisfy the constraints.
- $f_i(x)$ is generally unknown and non-identical for different is, and tests drawn from $f_i(x)$ may be dependent. In our simplified analysis, we assume that samples drawn from $f_i(x)$ are i.i.d.
- The means of populations belong to a distribution that is hard to estimate. Further, cross-overs and mutations applied at the end of a generation may change this distribution in an unknown fashion. For simplicity, we ignore this effect in our simplified model.
- We assume that the time to evaluate one test case using one HM is one unit. That is, we consider only logical time in our scheduling study.

B. Previous Work

In this section, we discuss two problems in resource scheduling and their solutions in our population-based learning system: sample-allocation and duration-scheduling.

1) *Sample-allocation* entails the scheduling of tests of HMs in a generation, given fixed numbers of tests in the generation and HMs to be tested. This problem is known in statistics as the (sequential) *allocation problem* [4] [30] and the scheduler, the *local scheduler*. The original problem suggested by Bechhofer in 1954 [4] is to decide the optimal allocation of picks, given a fixed total number of picks, assuming that the population mean and variance are known. The objective of these strategies is to maximize $P(CS)$, the probability of correctly selecting the population with the highest population mean when time is expended. Optimal solutions to problems in this class are unknown, and many extensions have been proposed to accommodate various trade-offs and relaxed assumptions. Existing strategies can be classified into static and dynamic.

Static sample-allocation strategies have a selection sequence fixed ahead of time, independent of the values of the picks obtained during selection. They are easier to analyze due to their simplicity. The most commonly used static strategy is the *round-robin* strategy, which takes samples from each population in turn. It allocates T/n tests to each population, given T tests and n population, while maximizing the worst-case $P(CS)$ when all populations have the same variance. Its drawback is that it tests the worst population to the same extent as the best, an obviously inefficient way of using resources. This is also the most commonly used strategy in genetics-based learning systems [5], [6], [13].

Dynamic (or adaptive) sample-allocation strategies select the population for testing based on previous sample values and other run-time information. Although more flexible, they are more complicated. One such strategy was developed by Tong and Wetzell to optimize $P(CS)$ when the selection process ends. It focuses on populations with high sample means, but also tests others with smaller means if they were not tested enough [30].

Sample-allocation strategies developed in statistics are not directly applicable in our learning system because they were developed with different objectives. In statistical sample allocation, the objective is to maximize $P(CS)$, given a finite number of populations. In contrast, our objective is to maximize the expected population-mean value of the population selected, given infinitely many populations initially. Since the maximum number of tests in learning is limited, we are interested in how close the actual performance of the selected HM is to the maximum performance within a pool of HMs.

We have developed before a *minimum-risk* scheduling strategy [15] which is a dynamic sample-allocation strategy with the above objective in mind. The goal of the strategy is to minimize the risk of the best population:

$$\text{minimize } risk = \text{minimize } E\left[\left(\mu_{\max} - \hat{\mu}_{\max}\right)^2\right]. \quad (4)$$

In our derivation, we assume that the distribution of each population is normal with a common variance, an obviously restricted assumption for many applications.

2) *Duration scheduling* entails deciding when to terminate an existing generation and to start a new one. A common strategy is to allocate a fixed duration to each generation, although better decisions can be made if past information is used. Duration-scheduling strategies can be classified as *static* and *dynamic*.

A *static (or fixed) duration-scheduling strategy* simply sets the duration of each generation to a predetermined value. Previous work [10] has shown that the most appropriate duration is dependent on the total time allocated to learning and the target application. To find a proper duration size for a given time limit, experiments with different durations must be run. The overhead for this is deemed too high to be useful.

A *dynamic (or adaptive) strategy*, on the other hand, uses run-time information to determine when each generation should end. A new set of HMs should be generated when the expected improvement from the new HMs are larger than the expected improvement from further testing the current set of HMs. There is very little research on this problem in statistics. One strategy we have studied extends our minimum-risk sample-allocation strategy [15] by estimating the distribution of new populations to be generated in the next generation using statistics collected in previous generations [2], [3]. (In the first generation, samples have to be drawn to estimate the initial distribution.) This strategy is restricted because it assumes that all populations have normal distributions with the same variance.

Another dynamic strategy we have studied is based on Bayesian analysis [2], [3], which results in a strategy that increases the duration size as the variance ratio (ratio of sample variance to variance of the μ_is) decreases. When the variance of the μ_is is large, it is easy to identify good populations; hence, the duration should be small. In general, the variance ratio is large when learning begins, and decreases as learning proceeds. Consequently, the duration size should be small initially and increases gradually. The difficulty with this strategy is that it is hard to find the correct duration without making simplifying assumptions on the distributions.

Instead of varying the duration size, a dual strategy is to fix the duration of a generation but varies the number of populations in it [15]. This is less flexible because it is difficult to adjust the population size dynamically.

The main shortcoming of existing work is that it assumes that HMs generated always have acceptable performance, even though most HMs may be pruned after a few tests. This is especially true in multiobjective applications in which we set constraints on performance metrics, and there may not be any acceptable HMs at the end of a generation. We address this problem in Section IV.D.

C. Nonparametric Minimum-Risk Sample-Allocation Strategy

A general sample-allocation strategy should not require information on the distributions of performance measures, as they change dynamically and are difficult to estimate. In this section, we propose a nonparametric sample-allocation strategy for determining HMs to be evaluated based on run-time performance information of populations. Our *nonparametric minimum-risk* strategy is extended from the parametric minimum-risk method we have developed earlier [15].

The objective of resource scheduling is to find the best HMs when all resources are exhausted. In general, this objective cannot be achieved since we cannot model changes in distributions between generations. To cope with this problem, we restrict our objective on sample allocation within a generation to the following objective: "*minimize the risk that the populations selected for generating new HMs when the generation ends are wrong.*" Note that this objective is for scheduling *within* a generation, but *not across* generations.

Consider a generation of K populations. Population i is characterized by information such as n_i (number of tests performed), μ_i (unknown population mean), σ_i^2 (unknown population variance), $\hat{\mu}_i$ (sample mean), S_i^2 (sample variance), F_i (true fitness value), and f_i (sample fitness value), where $F_i \triangleq \mu_i - c$, $f_i \triangleq \hat{\mu}_i - c$, and c is a constant that is usually set to the minimum fitness value of all populations. Note that f_i is an unbiased estimator of F_i, since $\hat{\mu}_i$ and S_i^2 are unbiased estimators of μ_i and σ_i^2, respectively. In this paper, we use the average metric as the fitness function.

We define the loss due to believing f_i as $L_i \triangleq E[(f_i - F_i)^2]$. Given $\hat{\mu}_i$ and σ_i (or S_i), we can calculate the value of L_i, noting that $E[(f_i - F_i)^2] = Var[f_i - F_i]$

$$L_i = \frac{\sigma_i^2}{n_i}. \tag{5}$$

The probability that population i will be selected for generating new ones is defined as $P_i \triangleq f_i \Big/ \sum_{j=1}^{K} f_j$. The scheduling problem can be formulated (heuristically) as follows:

$$\begin{aligned} &\text{minimize} \quad \phi \triangleq \sum_{i=1}^{K} P_i L_i \\ &\text{subject to} \quad \sum_{i=1}^{K} n_i = N. \end{aligned} \tag{6}$$

where N is the number of tests performed in the current generation. By using a Lagrange multiplier, we have

$$\text{minimize} \quad \hat{\phi} \triangleq \phi + \lambda\left(N - \sum_{i=1}^{K} n_i\right). \tag{7}$$

By equating $\partial\hat{\phi}/\partial n_i$ to zero, we have the optimality criteria as follows

$$-\lambda = \frac{P_i\sigma_i^2}{n_i^2} = \frac{P_j\sigma_j^2}{n_j^2}, \quad \text{for } i \neq j. \tag{8}$$

At any time t, the strategy is to minimize (6) for time $(t + 1)$; i.e., only one of the n_is can be increased by 1.

ONE-STAGE POLICY:

$$n_j \leftarrow n_j + 1 \quad \text{where} \quad \max_i \frac{P_i\sigma_i^2}{n_i^2} = \frac{P_j\sigma_j^2}{n_j^2} \tag{9}$$

Equation (9) says that the population to be tested is one that has large fitness value (i.e., large probability of being chosen for reproduction in the next generation) and that has large variance (i.e., large uncertainty in its mean). Note that (9) only tries to find the next population to be tested. In this case, P_i generally changes slowly ($P_i' \approx P_i$), and P_i can be approximated using information in the current generation.

In our experiments, we use S_i as an approximation to σ_i. To estimate S_i, at least two tests must be performed in each population (preferably four tests or more). Note that the Lagrange-multiplier procedure is valid only when performance values are continuous.

As an example, consider population i with four samples: 0.971, 1.006, 0.988, and 1.055. In this case, $\hat{\mu}_i$, S_i, and n_i are 1.005, 0.036, and 4, respectively. Assuming that there are a total 30 populations and that the minimum $\hat{\mu}$ is 0.910, then the fitness value of population i is $\hat{\mu}_i - 0.910 = 0.095$. Further, assume the current total fitness of the remaining 29 populations to be 0.781. Hence, P_i is 0.095/(0.781 + 0.095) = 0.108, and the risk value of this population is $P_i S_i^2/n_i^2$, ($= 8.78 \times 10^{-6}$). Assuming that population i has the largest risk value, and a new sample with value 0.920 is drawn from it. With this new sample, $\hat{\mu}_i$, S_i, n_i, f_i, and P_i become 0.988, 0.049, 5, 0.078, and 0.091, respectively, and the new risk value is reduced to 7.59×10^{-6}. This example shows that populations with high mean (hence, high P_i) and high S_i^2/n_i are more likely to have high risk values and be tested. Generally, risk values are reduced as more samples are drawn..

D. Duration Scheduling for Multiobjective Applications

This subsection presents duration scheduling methods for multiobjective applications. As discussed in Section II.C, we must constrain all but one objectives and optimize the unconstrained objective. However, all HMs may be pruned during learning when constraints are too tight. Applying random generation at that point is not helpful because random generation is the weakest of all generation methods, and it is unlikely that newly generated HMs will satisfy the constraints. To avoid this undesirable scenario, we must relax our original goal and find HMs as close as possible to the desired level of constraints, given the available resources. To this end, we must first start with loose constraints and gradually tightens them as learning proceeds.

D.1. Constraint Handling

We outline in the following a method for determining the likelihood that an HM satisfies the given constraints using the notations defined in Section IV.A. It is not possible to prune every HM violating one or more constraints ($c_i > \theta_i$) on one or more test cases because a) c_i is unknown and the estimated $\hat{c}_i$ is used instead, b) there is uncertainty in determining $\hat{c}_i$, and c) it is not possible to set worst-case performance bounds of an HM on a test case because by the nature of heuristics, their worst-case behavior may not be bounded.

We want to penalize HMs based on P_{ok}, the probability of satisfying the given set of constraints. Since the problems we study have high evaluation cost, we need to prune HMs that are unlikely to satisfy the constraints ($P_{ok} << 0.5$). Further, we like to give higher chance to HMs with P_{ok} close to 1 for further reproduction and testing.

Given the performance values of an HM over n test cases with sample mean $\hat{c}_i$ and sample variance $S^2(c_i)$ for each constrained metric C_i, random variable $(\hat{c}_i - c_i)\sqrt{n}/S(c_i)$ has Student's t-distribution with $n - 1$ degrees of freedom [9]. Accordingly, we can compute the probability that this HM satisfies threshold value θ_i on C_i.

$$P[c_i \le \theta_i] = F_t\left(n-1, \frac{\theta_i - \hat{c}_i}{S(c_i)/\sqrt{n}}\right), \tag{10}$$

where $F_t(\nu, x)$ is the *cdf* of Student's t distribution with ν degrees of freedom. It is important to point out that (10) is only valid when c_i is the average metric.

When there are multiple constrained metrics, the probability that all constraints (θ_i for $i = 1, \ldots, k$) are satisfied is equal to $P[c_1 \le \theta_1 \cap \ldots \cap c_k \le \theta_k]$. Based on probability theory, we know that

$$\prod_i P[c_i \le \theta_i] \le P_{ok} = P[c_1 \le \theta_1 \cap \cdots \cap c_k \le \theta_k] \le \min_i P[c_i \le \theta_i]. \tag{11}$$

Hence, we use $\min_i(P[c_i \le \theta_i])$ as an approximation to P_{ok}.

D.2. Dynamic Multiobjective Duration-Scheduling Strategy (DMDS)

Using the relaxed goal, the learning system iteratively finds HMs using increasingly harder constraints, instead of trying to find HMs that satisfy the final target constraints immediately. The initial set of constraints are selected in such a way that almost all randomly generated HMs will be accepted. This will ensure that some HMs are available for generating new HMs in the next generation. To set constraints in successive iterations, we apply an iterative refinement method we have developed in a real-time search algorithm for solving time-constrained combinatorial optimization problems [7]. We set new thresholds so that the times used in learning with successive thresholds grow in a geometric fashion. In this way, a small portion of the total time is used in intermediate iterations, and most of the effort is spent in the last iteration.

Using this iterative method, we need to set intermediate thresholds on constrained variables $\{C_i\}$ that are easier to achieve than the final target thresholds. We define the jth set of intermediate thresholds on k performance metrics as $\{\hat{\theta}_{1,j}, \hat{\theta}_{2,j}, \ldots, \hat{\theta}_{k,j}\}$. Since we want increasingly tougher constraints, we have the following property.

$$\infty = \hat{\theta}_{i,0} > \hat{\theta}_{i,1} > \cdots > \hat{\theta}_{i,\infty} = \theta_i,\ i = 1, \ldots, k \tag{12}$$

where θ_i is the final target threshold for C_i, and $\hat{\theta}_{i,0}$ is the initial threshold at the start of learning.

To control both the duration of each generation and the intermediate thresholds, we have developed the DMDS duration scheduling strategy. This strategy has two stages.

1) Stage 1: $\exists C_i$ *with* $\hat{\theta}_{i,j} > \theta_i$ *(not all target thresholds are satisfied)*. In this stage, DMDS decides the time for updating constraints and the values of the thresholds. It starts a new generation and a new set of thresholds when HMs satisfying the current constraints have been found,

and most HMs violating the current constraints have been eliminated. DMDS determines new thresholds based on profile data collected on thresholds during learning and the amount of time spent in finding acceptable HMs using these thresholds. The thresholds are set so that the time spent in each iteration to find feasible HMs satisfying the constraints grows geometrically.

2) Stage 2: $\hat{\theta}_{i,j} = \theta_i$ *for all i (all target thresholds have been reached).* When all the performance constraints are satisfied in a generation, the learning system finds the best HM that satisfies all the constraints. To do so, good HMs found in this generation are tested more thoroughly to ascertain that they satisfy the constraints to within a high degree of certainty before the next generation begins.

We defer until Section VI to show the effects of the various scheduling algorithms discussed in this section.

V. Finding General Heuristic Methods for All Subdomains

One of the key reasons in learning is to find a good HM that can generalize to test cases in new problem subdomains. Generalization is important because we perform learning on a very small number of subdomains, and there may be infinitely many subdomains in an application. Further, it is desirable to have one or very few HMs to be used in an application rather than a new HM for each problem instance.

The goal of generalization is somewhat vague: we like to find one or more HMs that perform well most of the time across multiple subdomains as compared to a baseline HM (if one exists). There are four issues to achieve this goal.

- How to compare the performance of HMs within a subdomain in a range-independent and distribution-independent fashion?
- How to define the notion that one HM performs well across multiple subdomains?
- How to find the condition(s) under which a specific HM should be applied?
- What are the trade-offs between cost and quality in generalization?

A. Probability of Win within a Subdomain

There are many ways to address the first issue raised in this section, and the solutions to the remaining problems depend on this solution. As is discussed at the end of Section II, scaling, normalization, and ranking do not work well. In this section, we propose a metric called probability of win to select good HMs.

P_{win}, the *probability-of-win* of HM_i within a subdomain, is defined as the probability that the true mean of HM_i (with respect to one performance measure) is better than the true mean of HM_j randomly selected from the pool. When HM_i is applied to test cases in subdomain d_m, we have

$$P_{win}(HM_i, d_m) = \frac{1}{|s|-1} \sum_{j \neq i} P\left[\mu_i^m > \mu_j^m \,\middle|\, \hat{\mu}_i^m, \hat{\sigma}_i^m, n_i^m, \hat{\mu}_j^m, \hat{\sigma}_j^m, n_j^m\right], \quad (13)$$

where $|s|$ is the number of HMs under consideration, d_m is a subdomain, and n_i^m, $\hat{\sigma}_i^m$, $\hat{\mu}_i^m$, and μ_i^m are, respectively, the number of tests, sample standard deviation, sample mean, and true mean of HM_i in d_m.

Under the assumptions that 1) performance values of each HM are normally distributed, 2) true variance σ_i^2 of HM_i is known, and 3) HMs in a subdomain are independent, $\hat{\mu}_i$, the sample mean for HM_i, can be assumed to have $N(\mu_i, \sigma_i / \sqrt{n_i})$ distribution. Consequently, $\hat{\mu}_i - \hat{\mu}_j$ has

$$N\left(\mu_i - \mu_j, \sqrt{\sigma_i^2/n_i + \sigma_j^2/n_j}\right)$$

distribution, and

$$Z = \left(\left(\hat{\mu}_i - \hat{\mu}_j\right) - \left(\mu_i - \mu_j\right)\right) \Big/ \sqrt{\sigma_i^2/n_i + \sigma_j^2/n_j}$$

has $N(0, 1)$ distribution.

From the above assumptions, the probability that HM_i is better than HM_j in d_m can now be computed approximately as follows.

$$P\left(\mu_i^m > \mu_j^m \,\middle|\, \hat{\mu}_i^m, \sigma_i^m, n_i^m, \hat{\mu}_j^m, \sigma_j^m, n_j^m\right) = \Phi\left[\frac{\hat{\mu}_i^m - \hat{\mu}_j^m}{\sqrt{\sigma_i^{m2}/n_i^m + \sigma_j^{m2}/n_j^m}}\right] \quad (14)$$

where $\Phi(x)$ is the cumulative distribution function for the $N(0, 1)$ distribution.

When $n_i > 30$ and $n_j > 30$, assumptions 1) and 2) above can be relaxed. Using the Law of Large Numbers [9], the performance of each HM does not have to be normally distributed for the condition stated in (14) to hold. In addition, $\hat{\sigma}_i$, the sample standard deviation, can be used in place of σ_i, the actual standard deviation, without signifinant loss in accuracy. In this case, the following equation can be used instead:

$$P\left(\mu_i^m > \mu_j^m \,\middle|\, \hat{\mu}_i^m, \hat{\sigma}_i^m, n_i^m, \hat{\mu}_j^m, \hat{\sigma}_j^m, n_j^m\right) = \Phi\left[\frac{\hat{\mu}_i^m - \hat{\mu}_j^m}{\sqrt{\hat{\sigma}_i^{m2}/n_i^m + \hat{\sigma}_j^{m2}/n_j^m}}\right]. \quad (15)$$

Note that using (15) when n_i and n_j are less than or equal to 30 will result in less accurate prediction.

To illustrate the concept, we show in Table VII probabilities of win of four HMs tested to various degrees. Note that probability of win is not directly related to sample mean but rather depends on sample mean, sample variance and number of tests performed. Further, the probability that h_i is better than h_j and the probability that h_j is better than h_i are both counted in the evaluation. Hence, the sum of P_{win} over all HMs will be half of the number of HMs evaluated.

TABLE VII
Probabilities of Win of Four HMs

h_i	$\hat{\mu}_i$	$\hat{\sigma}_i$	n_i	$P_{win}(h_i)$
1	43.2	13.5	10	0.4787
2	46.2	6.4	12	0.7976
3	44.9	2.5	10	0.6006
4	33.6	25.9	8	0.1231

P_{win}, defined in (13), is range-independent and distribution-independent because all performance values are transformed into probabilities between 0 and 1. It assumes that all HMs are i.i.d. and takes into account uncertainty in their sample averages (by using the variance values); hence, it is better than simple scaling which only compresses all performance averages into a range between 0 and 1.

B. Probability of Win across Subdomains

The use of probability of win leads to two ways to solve the second issue posted earlier in this section, namely, how to define the notion that one HM performs better than another HM across multiple subdomains.

First, we assume that when HM h is applied over multiple subdomains in partition Π_p of subdomains, all subdomain are equally likely. Therefore, we compute the probability of win of h over subdomains in Π_p as the average probability of win of h over all subdomain in Π_p.

$$P_{win}\left(h, \Pi_p\right) = \frac{\sum_{d \in \Pi_p} P_{win}(h, d)}{\left|\Pi_p\right|}, \qquad (16)$$

where Π_p is the pth partition of subdomains in the subspace. The HM picked is the one that maximizes (16). HMs picked this way usually wins with a high probability across most of the subdomains in Π_p but occasionally may not perform well in a few subdomains.

Second, we consider the problem of finding a good HM across multiple subdomains in Π_p as a multiobjective optimization problem itself. As is indicated in Section II.C, evaluating HMs based on a combined objective function (such as the average probability of win in (16)) may lead to inconsistent conclusions. To alleviate such inconsistencies, we should treat each subdomain independently and find a common HM across all subdomains in Π_p satisfying some common constraints. For example, let δ be the allowable deviation of the probability of win of any chosen HM from q^m_{win}, the maximum P_{win} in subdomain m. Generalization, therefore, amounts to finding h that satisfies the following constraints for every subdomain $m \in \Pi_p$.

$$P_{win}(h, m) \geq \left(q^m_{win} - \delta\right) \text{ for every } m \in \Pi_p \qquad (17)$$

In this formulation, δ may need to refined if there are too many or too few HMs satisfying the constraints.

To illustrate the generalization procedure, consider the vertex-cover problem discussed in Section II.B. Assume that learning had been performed on six subdomains (with graph connectivities 0.1, 0.15, 0.3, 0.4, 0.5, and 0.6, respectively), and that the five best decomposition HMs were generated from each. After full evaluation of the 30 HMs across the six subdomains, we computed the probability of win for each HM in each subdomain. Fig. 5 shows the probabilities of win of several of these HMs. If we generalize HMs based on (16), then HM_{129} will be picked since it has the highest average P_{win}. In contrast, if we generalize using (17), then HM_{107} will be picked because it has the smallest deviation from the maximum P_{win} in each subdomain. Note that both HMs are reasonable choices as a generalized HM that can be applied across all subdomains. To narrow down to one single generalized HM, further evaluations on the spread of performance values would be necessary (see Section VI.B).

Using probabilities of win to assess an HM across subdomains, we can now address the last two issues raised earlier in this section, which deal with the selection of multiple HMs.

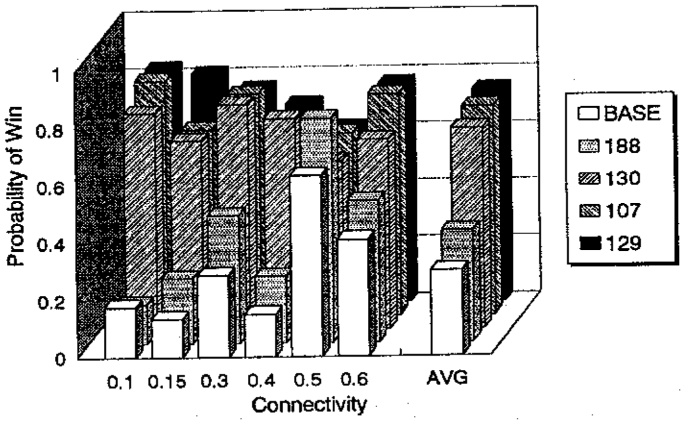


Fig. 5. Histogram showing probabilities of win of four HMs generalized across six subdomains and those of the baseline HM. (HM_{129} will be picked if (16) is used as the selection criterion; HM_{107} will be selected if (17) is used as the criterion.).

There are two ways that multiple HMs can be used:

1) each HM is used in a nonoverlapping subset of subdomains in the subspace (third issue), and
2) multiple HMs are always applied in solving a test case in the subspace (fourth issue).

The issue of finding condition(s) under which a specific HM should be applied is a difficult open problem. Solving this problem amounts to designing decision rules to partition the subspace of test cases into a finite number of partitions, each of which can be solved by one HM. This is possible in some applications that can be characterized by a small number of well-defined attributes. For instance, in the vertex-cover problem discussed in Sections II.B, graph connectivity is a unique attribute that allows us to decompose the space of all random graphs into partitions. For other applications, this may not be easy. For instance, in Post-Game Analysis discussed in Section II.C.1, there are a few attributes that can be used to decompose the subspace (e.g., number of processes in an application, number of processors in a multi-computer system). However, none of them is a good choice. In the CRIS test-pattern generation system [24] discussed in Section II.B, there are too many attributes that can be used to classify circuits (e.g., number of gates, length of the longest path). In this case, it is not clear which attributes should be used.

The last issue raised earlier in this section is on the tradeoffs between cost and quality in generalization. Since it may be difficult to identify a unique HM for each test case in the subspace, we can pick multiple HMs, each of which works well for some subdomains in the subspace, and apply all of them when a new test case is encountered. This is feasible only when the added cost of applying multiple HMs is compensated by the improved quality of the solutions. In this approach, the cost reflects the total computational cost of applying *all* the chosen HMs to solve a given test case.

The problem of selecting a set of HMs for a subspace amounts to picking *multiple* HMs and assigning each to a subdomain in the subspace. Assuming that $|H|$ such HMs are to be found, we need to decompose the subspace into $|H|$ partitions of subdomains, and assign one HM to all subdomains in each partition. The overall probability of win over the subspace is computed in a similar way as in (16). In mathematical form, let Ω be the set of all HMs tested in the subspace and Π be the set of all subdomains in this subspace, we are interested to find $H \subset \Omega$ such that $|H|$ is constant and the average P_{win} is maximized. That is,

$$P_{win}^{\max}(\Omega, \Pi) = \frac{\max\limits_{\substack{H \subset \Omega \\ |H| = \text{constant}}} \sum\limits_{d \in \Pi} \max\limits_{h \in H} P_{win}(h, d)}{|\Pi|}, \quad (18)$$

where $|\Pi|$ is the number of subdomains in subspace Π.

One way to find H in (18) is to enumerate over all possible ways of decomposing Π and assign the best HM to each partition. The problem is equivalent to the minimum-cover problem: given a set Π of subdomains and a set Ω of HMs (each of which covers one or more subdomains), find the minimum subset H of Ω so that each element of Π can be covered by one HM in H. The problem is NP hard and is solvable in polynomial time only when $|H|$ is two.

Fortunately, by applying (17), we can make the number of HMs arbitrarily small by choosing a proper δ. In this case, finding a fixed set of HMs that can best cover all subdomains in the subspace can be obtained by enumeration. Experimental results on such cost-quality trade-offs are presented in Section VI.C.

C. Generalization Procedure

The procedure to generalize HMs learned for subdomains in a problem subspace is summarized as follows:

1) Using the collective set of HMs obtained in the subdomains learned, find the probability of win (using (13)) of each HM in each subdomain learned or to be generalized.
2) Apply (18) to select the necessary number of HMs for evaluating test cases in the subspace. Equation (17) can be used to restrict the set of HMs considered in the selection process.

VI. Experimental Results

To illustrate the learning methods developed in this paper, we present in this section results on learning and generalization for the four applications discussed in Section II.

A. Process Mapping on Distributed-Memory Multicomputers

Process mapping involves placing a set of communicating processes on a multicomputer system so that the completion time of the processes is minimized. The problem is characterized by nondeterministic (data-dependent) execution times between interprocess communications and amount of data communicated between processes. It can be solved as a deterministic optimization problem using average execution times and data volumes; however, the solution is inaccurate when execution and communication activities change with time. Further, a deterministic model does not account for delays incurred due to blocked messages. Such unpredictable delays can only be found when the processes are actually executed or simulated.

Yan and Lundstrom proposed Post-Game Analysis (PGA), a simulation-based method for finding good mappings. Their system collects an execution trace consisting of actual execution times in between communications and amounts of data sent between processes, and uses them in a simulation system to find the actual completion time of a specific mapping. It then applies heuristics to propose a new mapping, evaluating the effectiveness of the new mapping through the simulation system. This iterative refinement is repeated until no further improvement is possible. PGA can be applied in practice by repeatedly collecting a trace for a short period of time, optimizing the mapping by PGA on a different computer while the original application program is run, and proposing a new mapping for the application program to use.

There are four components of the heuristics used in PGA:

1) *proposal-generation heuristics* for generating proposals to perturb a mapping based on independent optimization subgoals,
2) *priority-assessment heuristics* for prioritizing each site and process,
3) *transformation-generation heuristics* for generating possible transformations from the ordered list of sites and processes, and
4) *feasibility heuristics* for checking the feasibility of a move.

These heuristics are represented as expressions (or heuristic decision elements—HDEs) that combine values collected during program execution and are applied to make decisions.

These four heuristics components interact extensively in proposing alternative mappings. Consequently, we cannot isolate each set of heuristics and learned them independently. Instead, we consider the four components to make up a *PGA HM*, and learning aims to find the best collection of HDEs and the proper value for each threshold. PGA HMs used in learning are generated randomly, as well as by cross-overs and mutations.

PGA HMs are evaluated by two performance measures: quality of a mapping (process completion time of the mapping), and cost of finding the mapping (PGA execution time). Learning aims to find PGA HMs with the minimum completion time and cost within a user-specified limit. This is necessary since PGA has to be run concurrently with the application program, and a new mapping should be proposed within a time constraint.

In learning PGA HMs, we chose an application based on a divide-and-conquer algorithm: each process computes for a random amount of time, sends a message to each of its child processes to start their computation, and waits for the results from its descendents before reporting to its parent. We used

TABLE VIII
QUALITY-COST COMPARISON OF LEARNED AND GENERALIZED HMS USING A COST CONSTRAINT OF 30% OF THE ORIGINAL PGA COST

Subdomain			Generalized HM(s)		Learned HM(s)		Normalized Learning
ID	Architecture	Processes	Quality	Cost	Quality	Cost	Cost
1 *	3-by-3 mesh	100	0.934	0.251	0.934	0.251	583.3
2 +	4-by-4 mesh	100	0.933	0.231	0.948	0.277	349.9
3 +	5-by-5 mesh	100	0.954	0.235	0.951	0.230	428.8
4 *	3-by-3 mesh	200	0.993	0.283	0.993	0.283	505.8
5 +	4-by-4 mesh	200	0.986	0.244	0.993	0.274	348.0
6 +	5-by-5 mesh	200	0.964	0.274	0.961	0.269	416.9

Subdomains with "" are learned; subdomains with "+" are generalized from subdomains with the same number of processes. (Subdomains 1-3 belong to one partition, and subdomains 4-6 belong to another.) The cost of learning is the sum of normalized execution times with respect to the baseline HM.*

three machine architectures (3-by-3, 4-by-4, and 5-by-5 meshes) and two process sizes (100 and 200 processes), resulting in six subdomains. All subdomains were assumed to belong to one problem subspace. Each *PGA test case* specifies the number of processes in the divide-and-conquer tree, the execution time of each node of the tree, one of the machine architectures, and an initial mapping of the processes on the architecture. All experiments were performed on a Sun SparcStation 10/30. For the applications we have used, testing a PGA HM on one test case is time consuming: a small learning experiment with 6,400 evaluations (six subdomains, each with 800 tests for learning and 400 tests for final verification) took between seven to 11 days of CPU time.

We first evaluated three resource scheduling strategies: DMDS duration scheduling with minimum-risk sample allocation, fixed duration with minimum-risk sample allocation (MR), and fixed duration with round-robin sample allocation (RR). For each strategy, we performed five learning experiments of 800 tests over each of the six subdomains, using a cost constraint of 30% of the cost of the original PGA HM by Yan and Lundstrom [32]. We studied six cases of retaining 1, 3, 5, 10, 15, and 20 HMs at the end of learning for full verification, and compared the best and average qualities of the HMs achieved over five runs of each scheduling strategy.

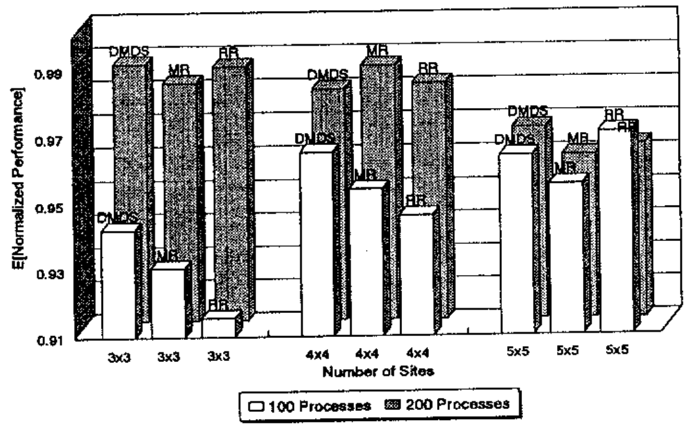


Fig. 6. Average quality of HMs selected under a cost constraint of 30% of the original PGA HM for three resource scheduling strategies: DMDS, MR and RR. (The number of HMs retained for full verification is five.)

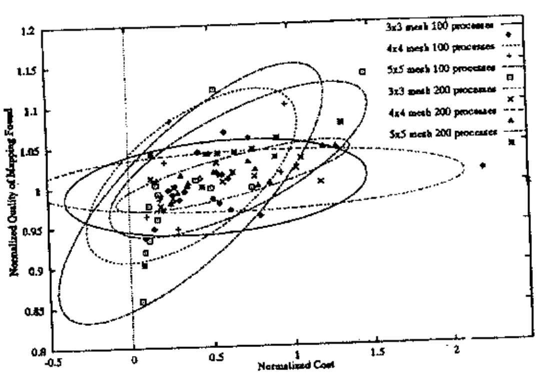
Fig. 6 shows the average speedups (quality) of HMs achieved by the three scheduling strategies over the six subdomains. Since the cost constraint is tight and all performance values represent slowdowns, we do not use symmetric speedups here. All these HMs have significantly lower costs and slightly worse qualities than those of the original PGA HM. Further, DMDS performs the best in four out of six subdomains and the second best in two other subdomains. Other results are similar, with smaller differences in quality as the number of HMs retained for verification increases.

DMDS also consistently finds better HMs more often than the two fixed-duration scheduling strategies when the cost constraint is tight. Of the 30 experiments performed under a 30% cost constraint, DMDS failed once when the five best HMs were retained for verification (RR failed five times and MR, four times). When the cost constraint is loose, DMDS does not perform better than the other scheduling strategies.

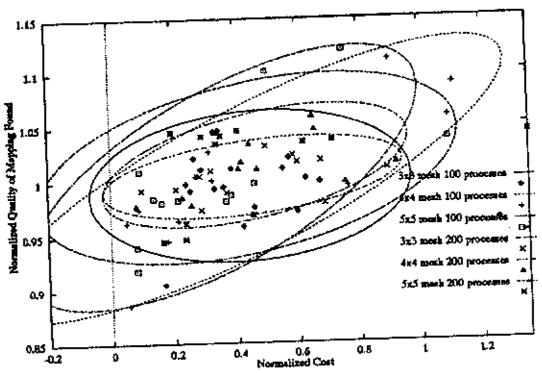
Our next experiments address the generalization of the HMs learned. We used three subdomains (3-by-3 mesh with 100 and 200 processes and 5-by-5 mesh with 200 processes) for learning, and generalized the HMs learned to the remaining three subdomains. In learning, 800 tests were performed for each subdomain, and the best five HMs that satisfied the cost constraint were selected for full verification. We considered two cost constraints: 30% and 100% of the cost of the original PGA HM by Yan and Lundstrom [32].

By applying (18), we have found that under the 1.0 cost constraint, all subdomains should belong to one partition and can be evaluated by one HM, and that under the 0.3 cost constraint, there are two partitions (partition 1 contains three subdomains with 100 processes, and partition 2, the remaining three subdomains with 200 processes). In this case, the PGA HMs learned do not generalize well and are biased towards the number of processes in the application program. Table VIII shows the costs and qualities of the generalized HMs as compared to those of the learned HMs. We see that both have similar costs and qualities. In contrast, the cost of learning is around 500 times higher than that of the generalized HMs.

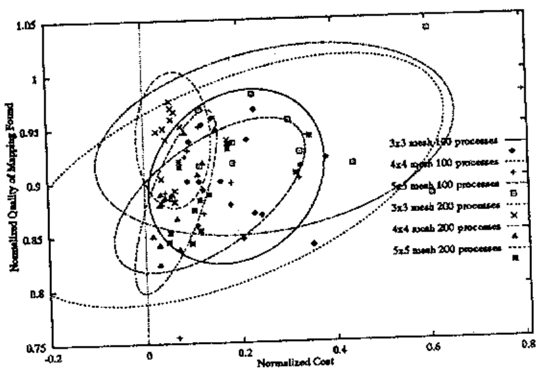
Next, we show the performance of the learned PGA HMs when generalized across the three multicomputer architectures under various combinations of cost constraint and number of processes in the application program. As discussed in Section II.C, performance values related to each objective in a



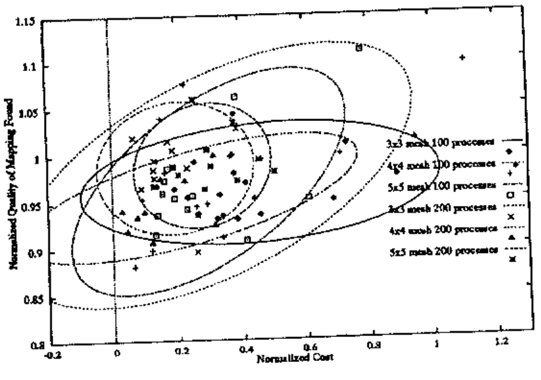
(a)



(b)



(c)



(d)

Fig. 7. Performance of PGA HMs learned for a 3-by-3 mesh architecture and (a) Top—100 process subdomain and 1.0 cost constraint, (b) Second—200 process subdomain and 1.0 cost constraint, (c) Third—100 process subdomain and 0.3 cost constraint, and (d) Bottom—200 process subdomain and 0.3 cost constraint.

multiobjective application need to be considered independently in order to avoid inconsistencies in evaluation.

To this end, we plot in a two-dimensional graph the distribution of the normalized quality of an HM on a test case and the corresponding normalized cost of the same HM and test case. Using a method we have developed earlier to show cost-quality trade-offs [15] we identified a 90% constant probability contour for each HM after removing outliers, checking for bivariate normality, and finding the 90% constant probability-density contour of the bivariate distribution.

Fig. 7 shows the cost-quality of generalized PGA HMs on various architectural configurations and numbers of nodes in the divide-and-conquer tree. Each of these four graphs represents the performance of one HM obtained by learning in one subdomain and generalizing to two other subdomains.

The HM used in Fig. 7a (resp., 7b) was obtained by generalizing HMs learned in a 100-process (resp., 200-process) subdomain under a 1.0 cost constraint and a 3-by-3 mesh architecture to the two remaining architectural configurations. Likewise, the HM used in Fig. 7c (resp., 7d) are HM_1 (resp., HM_4) in Table VIII. In learning the HMs, the fixed-duration minimum-risk strategy (resp., DMDS minimum-risk strategy) were used in Fig. 7a and 7b (resp., Fig. 7c and 7d). We see in Fig. 7a and 7b that all the contour plots are clustered together, implying that the PGA HMs selected under the 1.0 cost constraint generalize well to other subdomains. Further, these HMs have lower costs than the original HM (normalized to 1) while having qualities close to or better than the original HM. In Fig. 7c and 7d, we find both generalized HMs to have similar quality levels, but the HM generalized from the 100-process subdomains have higher costs than those from the 200-process subdomains. In this case, HMs that perform well for the 200-process subdomains would violate the cost constraint for the 100-process subdomains. This happens because the achievable cost for each subdomain in the 0.3 cost-constraint case is lower; for larger test cases, there is more room for improvement in terms of cost, and a lower relative cost can be achieved.

Similar conclusions can be drawn by computing P_{win}. For instance, the average P_{win} for a single HM generalized across all six subdomains is 0.78 for the 1.0 cost-constraint case, and 0.61 for the 0.3 cost-constraint case. In the latter case, the PGA HMs learned tend to specialize to the number of processes in the application program. For instance, the average P_{win} for two partitions in the problem subspace, one for the 100-process subdomains and the other for the 200-process subdomains, are 0.79 and 0.85, respectively. We have also observed similar results that generalization is easier for different divide-and-conquer trees or different applications problems when the cost constraint is loose [15] but more difficult when the cost constraint is tight.

B. Branch-and-Bound Search

A branch-and-bound (B&B) search algorithm is a systematic method for traversing a search tree or search graph in order to find a solution that optimizes a given objective while satisfying given constraints. It decomposes a problem into smaller subproblems and repeatedly decomposes them until a solution is found or infeasibility is proved. Each subproblem is represented by a node in the search tree/graph. The algorithm has four sets of HMs:

1) *selection HM* for selecting a search node for expansion based on a sequence of selection keys for ordering search nodes;
2) *decomposition HM* (or branching mechanism) for expanding a search node into descendents using operators to expand (or transform) a search node into child nodes;
3) *pruning HM* for pruning inferior nodes in order to trim potentially poor subtrees; and
4) *termination HM* for determining when to stop.

In this paper, we apply learning to find only new *decomposition HMs*; preliminary results on learning of selection and pruning HMs can be found in the reference [8]. We consider optimization search, which involves finding the optimal solution and proving its optimality.

We illustrate our method on three applications: traveling salesman problems on incompletely connected graphs mapped on a 2D plane (TSP), vertex-cover problems (VC), and knapsack problems (KS). Table IX shows the parameters used to generate a *test case* in each application. All subdomains are assumed to belong to one problem subspace.

TABLE IX
GENERATION OF 12 SUBDOMAINS OF TEST CASES FOR TESTING DECOMPOSITION HMS IN A B&B SEARCH

Appl.	Subdomain Attributes
VC	• Connectivity of vertices is (0.05, 0.1, 0.15, 0.2, 0.25, 0.3, 0.35, 0.4, 0.45, 0.5, 0.55, or 0.6) • Number of vertices is between 16 and 45
TSP	• Distributions of 8-18 cities (uniformly distributed between 0-100 on both the *X* and *Y* axes, uniformly distributed on one axis and normally distributed on another, or normally distributed on both axes). • Graph connectivity of cities is (0.1, 0.2, 0.3, or 1.0)
KS	• Range of both profits and weights is {(100-1000), (100-200), (100-105)} • Variance of profit/weight ratio is (1.05, 1.5, 10, 100) • 13-60 objects in the knapsack

The problem solver here is a B&B algorithm, and a test case is considered solved when its optimal solution is found. Note that the decomposition HM is a component of the B&B algorithm. We use well-known decomposition HMs developed for these applications as our baseline HMs (see Table XI). The normalized cost of a candidate decomposition HM is defined in terms of its *average symmetric speedup* (see (2)), which is related to the number of nodes expanded by a B&B search using the baseline HM and that using the new HM. Note that we do not need to measure quality as both the new and existing HMs when applied in a B&B search look for the optimal solution.

Our first experiments study the effects of fixed-duration strategies (RR and MR) on learning. DMDS was not used because there is only one objective measure. Fig. 8 shows the quality of HMs found as a function of learning time for the two scheduling strategies. Each point in the graph was obtained by averaging the symmetric speedups of HMs selected if learning had been stopped at that point.

Fig. 8 shows that MR outperforms RR, that MR is more likely to identify the top HM, and that MR requires less HMs to be retained for full verification at the end of learning. For this reason, we use the fixed-duration MR scheduling strategy in the remaining results in this subsection. Fig. 8 also shows that verifying more HMs at the end leads to better HMs (albeit higher verification cost).

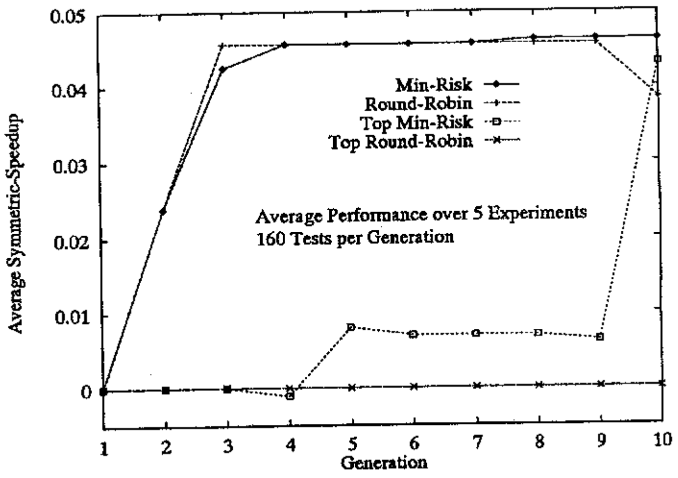


Fig. 8. Average performance over five runs of the HMs selected for the VC problem with edge connectivity of 0.15 using the MR and RR scheduling strategies. (Top-RR and Top-MR mean that only one HM was retained for verification; RR and MR mean that five HMs were retained.)

Next, we generalize the HMs learned to new subdomains. For each application, we selected six subdomains and performed learning on each using 1,600 tests. We then selected the top five HMs from each learned subdomain, fully verified them on all the learned subdomains, and selected one final HM to be used across all subdomains. Table X summarizes the learning, generalization, and validation results. For learning, we show the average symmetric speedup of the top HM learned in each subdomain and the normalized cost of learning, where the latter was computed as the ratio of the total CPU time for learning and the harmonic mean of the CPU times required by the baseline HM on test cases used in learning. The results show that a new HM learned for a subdomain has around 1-35% improvement in its average symmetric speedups and 3,000-16,000 times in learning costs.

Table X also shows the average symmetric speedups of the generalized HMs. We picked six subdomains randomly for learning. After learning and full verification of the top five HMs in each subdomain, we applied (18) to identify the generalized HM across all 12 subdomains. Our results show that the generalized HMs have between 0-8% improvement in average symmetric speedups. Note that these results are worse than those obtained by learning, and that the baseline HM is the best HM in solving the knapsack problem.

TABLE X
RESULTS OF LEARNING AND GENERALIZATION FOR VC, TSP, AND KS

Type	Application	Performance Measure	Subdomain												Average
			1	2	3	4	5	6	7	8	9	10	11	12	
Learning	VC	Sym-SU	0.000	0.011	0.041	0.000	0.044	0.022	0.008	0.013	0.000	0.000	0.000	0.000	0.012
		Cost	26343.5	23570.9	21934.1	12951.6	11034.3	12414.4	5871.0	8093.3	6878.0	5051.2	3826.2	3107.3	11756.3
	TSP	Sym-SU	0.194	0.073	0.288	0.378	0.106	0.068	0.267	0.382	0.048	0.165	0.208	0.083	0.188
		Cost	2846.6	1543.9	2077.7	2207.7	2314.9	1865.6	1889.9	1847.5	2509.7	1947.0	1445.4	1958.8	2037.9
	KS	Sym-SU	0.000	0.000	0.000	0.000	0.000	0.000	0.893	0.000	0.263	0.107	2.840	0.089	0.349
		Cost	25707.7	32587.9	9671.6	26408.1	24903.6	22309.1	3648.1	7943.1	8114.7	6476.2	772.9	10684.4	14935.6
Generalization	VC	Sym-SU	0.218	0.283*	0.031	0.068*	0.054	0.060*	0.017	0.049*	0.016	–0.000*	–0.011	0.028*	0.068
	TSP	Sym-SU	0.072*	0.004	0.082*	0.225	0.005*	0.061*	0.139	0.155	–0.010	0.054	0.090*	0.083*	0.080
	KS	Sym-SU	0.000*	0.000*	0.000	0.000	0.000	0.000*	0.000*	0.000	0.000*	0.000	0.000	0.000*	0.000
Validation	VC	Sym-SU	0.070	0.638	0.241	0.078	0.073	0.020	–0.013	–0.004	–0.018	–0.000	–0.019	–0.010	0.088
	TSP	Sym-SU	0.417	0.036	0.144	0.155	0.131	0.364	1.161	0.101	0.108	0.008	0.022	0.131	0.231
	KS	Sym-SU	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000

In the results on generalization, numbers with "" are the ones learned; only one common HM is generalized to all 12 subdomains.*

The bottom part of Table X shows the average symmetric speedups when we validate the generalized HMs on larger test cases. These test cases generally require 10-50 times more nodes expanded than those used earlier. Surprisingly, our results show better improvement (9-23%). Further, six of the twelve subdomains with high degree of connectivity in the vertex-cover problem have slowdowns. This is a clear indication that these subdomains should be grouped in a different subspace and learned separately.

TABLE XI
ORIGINAL AND GENERALIZED DECOMPOSITION HMs USED IN A B&B SEARCH

Application and Variables Used in Constructing HMs	Orig. HM	Gen. HM
Vertex-Cover Problem l = live degree of vertex (uncovered edges) d = dead degree of vertex (covered edges) n = average live degree of all neighbors Δl = difference between l from parent node to current node	l	$1000\,l$ $+n$ $-\Delta l$
Traveling-Salesman Problem c = length of current partial tour m = min length to complete current tour a = avg. length to complete current tour l = number of neighboring cities not yet visited d = number of neighbors already visited	c	$m*c$
Knapsack Problem p, w = profit/weight of object s = weight slack = weight limit – current weight $p_{\max}$, $p_{\min}$ = max./min. profit of unselected object $w_{\max}$, $w_{\min}$ = max./min. weight of unselected object	p/w	p/w

The new HM learned for VC can be interpreted as follows: l is the primary key, and $n - \Delta l$ is the secondary key.

Finally, Table XI shows the new decomposition HMs learned for the three applications. We list the variables that were supplied to the learning system. In addition to these variables, we have also included constants that can be used by the heuristics generator. An example of such a constant is shown in the HM learned for the vertex-cover problem. This formula can be interpreted as using l as the primary key for deciding which node to be included in the covered set. If the ls of two alternatives are different, then the remaining terms in the formula ($n - \Delta l$) are insignificant. In contrast, when the ls are the same, then we use $n - \Delta l$ as a tie breaker.

In short, our results show that reasonable improvements can be obtained by learning and by generalization. We anticipate further improvements by

1) learning and generalizing new pruning HMs in a depth-first search,
2) partitioning the problem space into a number of subspaces and learning a new HM for each, and
3) identifying attributes that help explain why one HM performs well in some subdomains.

C. Heuristics for Sequential Circuit Testing

CRIS is a genetic-search package developed by experts for generating test patterns to test VLSI circuits [24] It is based on continuous mutations of a given input test sequence and on analyzing the mutated vectors for selecting a test set. The package has been applied successfully to generate test patterns with high fault coverages for large combinatorial and sequential circuits.

In our application of TEACHER to improve CRIS, we treat CRIS as a problem solver in a black box. The inputs to CRIS are a set of eight parameters that we treat as our HM (see Table XII). We were also given a suite of 20 sequential benchmark circuits [17] Since these circuits are from different applications, we cannot characterize them by some common features. Consequently, we treat each circuit as an individual subdomain. We further assume that all the circuits are from one subspace, and wish to learn one common HM for all the circuits. Note that parameter P_8 is a random seed, implying that CRIS can be run multiple times using different random seeds in order to obtain better fault coverages. (In our experiments, we used a fixed sequence of ten random seeds.)

A major problem in using the original CRIS is that it is hard to find the proper values of the seven parameters (excluding the random seed) for a particular circuit. The designer of CRIS manually tuned these parameters for each circuit, resulting in HMs that are hard to generalize. This was done because the designer wanted to obtain the highest possible fault coverage for each circuit, and computation cost was only a secondary consideration. Note that the times for manually learning these HMs were exceedingly high and were not reported in the reference [24].

Our goal in learning and generalization is to develop one common HM that can be applied across all the benchmark circuits and that has similar or better fault coverages as compared to those of the original CRIS. The advantage of having one HM is that it can be applied to new circuits without further manual tuning.

In our learning experiments, we chose five circuits (subdomains) for learning. Each learning experiment had ten fixed-duration generations with a total of 1,000 applications of CRIS. The learning system started with a pool of 30 active HMs in each generation and retained the top ten HMs at the end of the generation. The minimum-risk sample-allocation strategy was used in learning. The 20 best HMs at the end of learning were selected for final verification.

Using the five best HMs from each of the five learning experiments, we evaluated the 25 HMs fully using ten random seeds on the five subdomains used in learning and five new subdomains. We then selected one generalized HM to be used across the ten circuits. The generalized HM we have found is shown in Table XII.

TABLE XII
HM PARAMETERS IN CRIS

Param.	Type	Range	Step	Definition	Learned Value
P_1	int.	1-10	1	related to the number of stages in a flip-flop	1
P_2	int.	1-40	1	sensitivity of changes of state of a flip-flop	12
P_3	int.	1-40	1	survival rate of a test sequence in the next generation	38
P_4	float	0.1-10.0	0.1	no. of test vectors concatenated to form a new vector	7.06
P_5	int.	50-800	10	number of useless trials before quitting	623
P_6	int.	1-20	1	number of generations	1
P_7	float	0.1-1.0	0.1	how genes are spliced in the genetic algorithm	0.1
P_8	int.	any	1	seed for the random number generator	–

The type, range, and step size were recommended to us by the designer of CRIS.

Table XIII shows the results after learning and generalization. For each circuit, we present the average and maximum (over 10 random seeds) fault coverages and the costs of the learned and generalized HMs. These fault coverages are compared against the published fault coverages of CRIS [24] as well as those of the deterministic HITEC system [21]. Note that the cost of generating the maximum fault coverage is 10 times the average cost.

TABLE XIII
PERFORMANCE OF HMS FOUND FOR CRIS

Ckt. ID	Fault Coverage		HITEC Cost	Learned HMs			Generalized HM		
	CRIS	HI-TEC		Avg. FC	Max. FC	Cost	Avg. FC	Max. FC	Avg. Cost
*s298	82.1	86.0	15984.0	84.9	86.4	15255.1	84.7	86.4	10.9
s344	93.7	95.9	4788.0	96.1	96.2	22305.3	72.4	87.0	7.2
+s382	68.6	90.9	43200.0	86.9	88.0	12612.3	72.4	87.0	7.2
s386	76.0	81.7	61.8	78.9	80.5	6750.6	77.5	78.9	3.5
*s400	84.7	89.9	46560.0	85.5	88.8	13402.8	71.2	85.7	8.4
s444	83.7	87.3	57960.0	85.8	87.1	13863.6	79.8	85.4	9.3
*S526	77.1	65.7	168480.	76.8	77.3	17278.2	70.0	77.1	10.0
s641	85.2	86.5	1080.0	85.7	86.5	18914.3	85.0	86.1	19.5
+s713	81.7	81.9	91.2	81.6	81.9	24714.5	81.3	81.9	23.0
s820	53.1	95.6	5796.0	47.8	55.2	55948.0	44.7	46.7	51.3
*s832	42.5	93.9	6336.0	45.6	59.4	56526.2	44.1	45.6	44.6
s1196	95.0	99.7	91.8	93.9	95.2	22444.4	92.0	94.1	20.0
*s1238	90.7	94.6	132.0	90.6	91.7	26961.4	88.2	89.2	23.0
s1488	91.2	97.0	12960.0	94.4	96.3	87314.4	94.1	95.2	85.6
+s1494	90.1	96.4	6876.0	93.7	95.4	88649.9	93.2	94.1	85.5
s1423	77.0	40.0	–	84.6	89.3	241983.	82.0	88.3	210.4
+s5378	65.8	70.3	–	71.5	73.7	560312.	65.3	69	501.8
am2910	83.0	85.0	–	85.4	86.5	345253.	83.7	85.2	307.6
+div16	75.0	72.0	–	80.6	82.6	167876.	79.1	81.0	149.9
tc100	70.8	80.6	–	75.8	76.8	244061.	72.6	75.9	163.8

Learned subdomains are marked by "" and generalized subdomains by "+." The costs and fault coverages of HITEC are from the literature. Costs of our experiments are running times in seconds on a Sun SparcStation 10/512, whereas costs of HITEC are running times in seconds on a Sun SparcStation SLC (around 4-6 times slower than a Sun SparcStation 10/512.*

TABLE XIV
SUMMARY OF COMPARISON OF OUR GENERALIZED HMS WITH RESPECT TO HITEC AND CRIS

Our HM wins/ties with respect to the following	Generalized HM		Learned HM	
	Max. FC	Avg. FC	Max. FC	Avg. FC
HITEC	6, 1	4, 0	7, 2	6, 0
CRIS	16, 1	11, 0	20, 0	15, 0
Both HITEC and CRIS	5, 2	3, 0	7, 2	5, 0

The first number in each entry shows the number of wins out of 20 circuits, and the second, the number of ties.

Table XIV summarizes the improvements of our learned and generalized HMs as compared to the published results of CRIS and HITEC. Each entry of the table shows the number of times our HM wins (and ties) as compared to the method(s) in the first column. Our results show that our generalized HM wins five out of 20 circuits as compared to the best fault coverages of HITEC and CRIS. Learning can improve this further to result in seven wins (and two ties) out of 20 circuits (albeit much higher computational costs as compared to those of HITEC and CRIS).

Our results are significant in the following aspects:

1) new faults detected by our generalized HMs were not discovered by previous methods;
2) only one HM (rather than many circuit-dependent HMs) was found for all the circuits.

D. Heuristics for VLSI Placement and Routing

In our last application, we use TimberWolf [27] as our problem solver. This is a software package based on simulated annealing to place and route various components (transistors, resistors, capacitors, wires, etc.) on a piece of silicon. Its goal is to minimize the chip area needed while satisfying constraints such as the number of layers of poly-silicon for routing and the maximum signal delay through any path. Its operations can be divided into three steps: placement, global routing, and detailed routing.

The placement and routing problem is NP-hard; hence, heuristics are generally used. Simulated annealing (SA) used in TimberWolf is an efficient method to randomly search the space of possible placements. Although in theory SA converges asymptotically to the global optimum with probability one, the results generated in finite time are usually suboptimal. As a result, there is a trade-off between quality of a result and cost (or computational time) of obtaining it. In TimberWolf version 6.0, the version we have studied, there are two parameters to control the running time (which indirectly control the quality of the result): *fast-n* and *slow-n*. The larger the *fast-n* is, the shorter time SA will run. In contrast, the larger the *slow-n* is, the longer time SA will run. Of course, only one of these parameters can be used at any time.

TimberWolf has six major components: *cost function*, *generate function*, *initial temperature*, *temperature decrement*, *equilibrium condition*, and *stopping criterion*. Many parameters in these components have been tuned manually. However, their settings are generally heuristic because we lack domain knowledge to set them optimally. In Table XV, we list the parameters we have studied. Our goal is to illustrate the power of our learning and generalization procedures and to show improved quality and reduced cost for the placement and routing of large circuits, despite the fact that only small circuits were used in learning.

In our experiments, we used seven benchmark circuits [17] (s298, s420, fract, primary1, struct, primary2, industrial1). Here, we have only studied the application of TimberWolf to standard-cell placement, although other kinds of placement (gate-array placement and macro/custom-cell placement) can be studied in a similar fashion. In our experiments, we used *fast-n* values of 1, 5, and 10, respectively. We first applied TEACHER to learn good HMs for circuits *s298* with *fast-n* of 1, *s420* with *fast-n* of 5, and *primary1* with *fast-n* of 10, each of which was taken as a learning subdomain. We used a fixed sequence of 10 random seeds (P_{11} in Table XV) in each subdomain to find the statistical performance of an HM. Each learning experiment involved 1,000 applications of TimberWolf divided into ten generations. Based on the best 30 HMs (10 from each subdomain), we applied our generalization procedure to obtain one generalized HM. This generalized HM as well as the default HM are shown in Table XV.

TABLE XV
PARAMETERS IN TIMBERWOLF (VERSION 6) USED IN OUR HM FOR LEARNING AND FOR GENERALIZATION

Param.	Range	Step	Meaning	Default	Learned
p_1	0.1-2.5	0.1	vertical path weight for estimating the cost function	1.0	0.9584
p_2	0.1-2.5	0.1	vertical wire weight for estimating the cost function	1.0	0.2315
p_3	3-10	1	orientation ratio	6	10
p_4	0.33-2.0	0.1	range limiter window change ratio	1.0	1.2987
p_5	10.0-35.0	1.0	high temperature finishing point	23.0	10.0416
p_6	50.0-99.0	1.0	intermediate temperature finishing point	81.0	63.6962
p_7	100.0-150.0	1.0	low temperature finishing point	125.0	125.5509
p_8	130.0-180.0	1.0	final iteration temperature	155.0	147.9912
p_9	0.29-0.59	0.01	critical ratio that determines acceptance probability	0.44	0.3325
p_{10}	0.01-0.12	0.01	temperature for controller turn off	0.06	0.1124
p_{11}	integer	1	seed for the random number generator	-	-

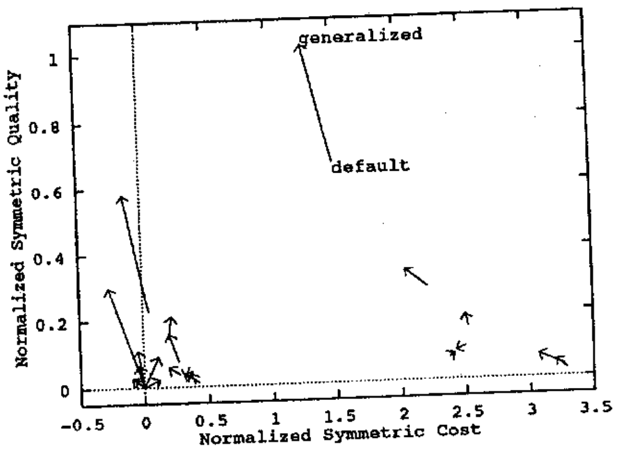


Fig. 9. Comparison of normalized average performance between the default and the generalized HMs. The plots are normalized with respect to the performance of the baseline HM on each circuit using *fast-n* = 10. (See (2)).

Fig. 9 plots the quality (higher quality in the *y*-axis means reduced chip area averaged over 10 runs using the defined random seeds) and cost (average execution time of TimberWolf) between the generalized HM and the default HM on all seven circuits with *fast-n* of 1, 5, and 10, respectively. Note that all performance values in Fig. 9 are normalized with respect to those of *fast-n* of 10, and that the positive (resp., negative) portion of the x-axes shows the fractional improvement (resp., degradation) in computational cost with respect to the baseline HM using *fast-n* of 10 for the same circuit. Each arrow in this figure points from the average performance of the default HM to the average performance of the generalized HM.

Among the 21 test cases, the generalized HM has worse quality than that of the default in only two instances (both for circuit fract), and has worse cost in four out of 21 cases. We see in Fig. 9 that most of the arrows point in a left-upward direction, implying improved quality and reduced cost. Note that these experiments are meant to illustrate the power of our generalization procedure. We expect to see more improvement as we learn other functions and parameters in TimberWolf.

VII. Conclusions

In this paper, we have studied automated learning of performance-related heuristics for knowledge-lean applications using genetics-based learning methods. To summarize, we have derived the following results.

1) We have found inconsistencies in performance evaluation of heuristics due to multiple tests, multiple learning objectives, normalization, and changing behavior of heuristics across problem subdomains. We have proposed methods to cope with some of these anomalies.
2) We have studied strategies to schedule resources for tests in learning. An improvement over previous strategies is that our strategy is nonparametric and does not rely on the underlying performance distribution of heuristics. We have also proposed a scheduling strategy to cope with one or more learning objectives. Our results show that scheduling is important when tests are expensive and test results are noisy.
3) We have studied methods to find good HMs that can generalize to unlearned domains. Using a range-independent measure called probability of win, we can compare and rank heuristics across problem subdomains in a uniform manner. In case that there are trade-offs between cost and quality, our learning system will propose alternative heuristics showing such trade-offs.
4) We have found better heuristics for process mapping, branch-and-bound search, test-pattern generation in circuit testing, and VLSI cell placement and routing..

There are several areas that we plan to study in the future.

1) One of these areas is the identification of problem subdomains for learning and subspaces for generalization. Since such demarcation is generally vague and imprecise, we plan to apply fuzzy sets to help define subdomains and subspaces. Fuzzy logic can also help identify heuristics that can be generalized, especially when there are multiple objectives in the application.
2) We plan to study metrics for performance evaluation besides the average metric studied in this paper. One of such metrics is the maximum metric that is useful when a heuristic method can be applied multiple times in order to generate better results at higher costs. This is also related to better generalization procedures that trades between improved quality and higher cost.
3) Finally, we plan to carry out learning on more applications. The merits of our system, of course, lie in finding better heuristics for real-world problems, which may involve many contradicting objectives. Our experience in this paper is on an application with two objectives. To cope with applications with many objectives, we need to extend our scheduling and generalization strategies.

Acknowledgments

The authors are indebted to Prof. C. V. Ramamoorthy who processed the original submission in 1991 and the revised submission in 1994. The authors would like to thank Dr. Jerry Yan for letting us use his PostGame Analysis package described in Section VI.A. The authors like to acknowledge Mr. Yong-Cheng Li for interfacing TEACHER to TimberWolf and for collecting some preliminary results in Section VI.D. The authors are grateful to Mr. Steve Schwartz who co-developed the first version of TEACHER in 1991.

This research was supported in part by National Science Foundation Grants MIP 92-10584 and MIP 88-10584 and National Aeronautics and Space Administration Grants NCC 2-481, NAG 1-613, and NGT 50743 (NASA Graduate Fellowship Program).

References

[1] D.H. Ackley, *A Connectionist Machine for Genetic Hill-Climbing.* Boston, Mass.: Kluwer Academic, 1987.

[2] A.K. Aizawa and B.W. Wah, "Scheduling of genetic algorithms in a noisy environment," *Evolutionary Computation*, vol. 2, no. 2, pp. 97-122, 1994.

[3] A.N. Aizawa and B.W. Wah, "A sequential sampling procedure for genetic algorithms," *Computers and Mathematics with Applications*, vol. 27, no. 9/10, pp. 77-82, May 1994.

[4] R.E. Bechhofer, "A single-sample multiple-decision procedure for ranking means of normal populations with known variances," *Ann. Math. Statist.*, vol. 25, no. 1, pp. 16-39, Mar. 1954.

[5] R.E. Bechhofer, A.J. Hayter, and A.C. Tamhane, "Designing experiments for selecting the largest normal mean when the variances are known and unequal: Optimal sample size allocation," *J. Statistical Planning and Inference*, vol. 28, pp. 271-289, 1991.

[6] L.B. Booker , D.E. Goldberg, and J.H. Holland, "Classifier systems and genetic algorithms," *Machine Learning: Paradigm and Methods*, J. Carbonell, ed., MIT Press, 1990.

[7] L.-C. Chu and B.W. Wah, "Optimization in real time," *Proc. Real Time Systems Symp.*, pp. 150-159, Nov. 1991.

[8] L.-C. Chu, "Algorithms for combinatorial optimization in real time and their automated refinement by genetic programming," PhD thesis, Dept. of Electrical and Computer Eng., Univ. of Illinois, Urbana, Ill., May 1994.

[9] J.L. Devore, *Probability and Statistics for Engineering and the Sciences.* Monterey, Calif.: Brooks/Cole, 1982.

[10] J.M. Fitzpatrick and J.J. Grefenstette, "Genetic algorithms in noisy environments," *Machine Learning*, vol. 3, no. 2/3, pp. 101-120, Oct. 1988.

[11] C.M. Fonseca and P.J. Fleming, "Genetic algorithms for multiobjective optimization: Formulation, discussion, and generalization," *Proc. Fifth Int'l Conf. Genetic Algorithms*, pp. 416-423, June 1993.

[12] F.W. Gembicki, "Vector optimization for control with performance and parameter sensitivity indices," PhD thesis, Case Western Reserve Univ., Cleveland, 1974.

[13] J.J. Grefenstette, C.L. Ramsey, and A.C. Schultz, "Learning sequential decision rules using simulation models and competition," *Machine Learning*, vol. 5, pp. 355-381, 1990.

[14] A. Ieumwananonthachai, A.N. Aizawa, S.R. Schwartz, B.W. Wah, and J.C. Yan, "Intelligent mapping of communicating processes in distributed computing systems," *Proc. Supercomputing '91*, pp. 512-521, Albuquerque, N.M., Nov. 1991.

[15] A. Ieumwananonthachai, A. Aizawa, S.R. Schwartz, B.W. Wah, and J.C. Yan, "Intelligent process mapping through systematic improvement of heuristics," *J. Parallel and Distributed Computing*, vol. 15, pp. 118-142, June 1992.

[16] J.R. Koza, *Genetic Programming*. Cambridge, Mass.: MIT Press, 1992.

[17] LayoutSynth92, *International Workshop on Layout Synthesis*, ftp site: mcnc.mcnc.org in directory /pub/benchmark, 1992.

[18] M.B. Lowrie and B.W. Wah, "Learning heuristic functions for numberic optimization problems," *Proc. Computer Software and Applications Conf.*, pp. 443-450, Chicago, Oct. 1988.

[19] P. Mehra and B.W. Wah, *Load Balancing: An Automated Learning Approach*. World Scientific Publishing Co. Pte. Ltd., 1995.

[20] A. Newell, J.C. Shaw, and H.A. Simon, "Programming the logic theory machine," *Proc. 1957 Western Joint Computer Conf.*, pp. 230-240, 1957.

[21] T.M. Niermann and J.H. Patel, "HITEC: A test generation package for sequential circuits," *European Design Automation Conf.*, pp. 214-218, 1991.

[22] J. Pearl, *Heuritstics—Intelligent Search Strategies for Computer Problem Solving*. Reading, Mass.: Addison-Wesley, 1984.

[23] C.L. Ramsey and J.J. Grefenstette, "Case-based initialization of genetic algorithms," *Proc. Fifth Int'l Conf. Genetic Algorithms*, pp. 84-91, June 1993.

[24] D.G. Saab, Y.G. Saab, and J.A. Abraham, "CRIS: A test cultivation program for sequential VLSI circuits," *Proc. Int'l Conf. Computer Aided Design*, pp. 216-219, Santa Clara, Calif., 1992.

[25] S.R. Schwarta and B.W. Wah, "Automated parameter tuning in stereo vision under time constraints," *Proc. Int'l Conf. Tools for Artificial Intelligence*, pp. 162-169, Nov. 1992.

[26] C. Sechen and A. Sangiovanni-Vincentelli, "The TimberWolf placement and routing package," *J. Solid-State Circuits*, vol. 20, no. 2, pp. 510-522, 1985.

[27] C. Sechen, *VLSI Placement and Global Routing Using Simulated Annealing*. Boston: Kluwer Academic, 1988.

[28] R.S. Sutton, "Temporal credit assignment in reinforcement learning," PhD thesis, Univ. of Massachusetts, Amherst, Mass., Feb. 1984.

[29] C.-C. Teng and B.W. Wah, "An automated design system for finding the minimal configuration of a feed-forward neural network," *Int'l Conf. Neural Networks*, vol. 3, pp. 1,295-1,300, June 1994.

[30] Y.L. Tong and D.E. Wetzell, "Allocation of observations for selecting the best normal population," *Design of Experiments: Ranking and Selection*, T.J. Santner and A.C. Tamhane, eds., pp. 213-224. New York: Marcel Dekker, 1984.

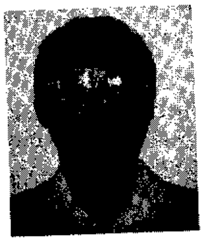
[31] B.W. Wah, "Population-based learning: A new method for learning from examples under resource constraints," *IEEE Trans. Knowledge and Data Engineering*, vol. 4, no. 5, pp. 454-574, Oct. 1992.

[32] J.C. Yan and S.F. Lundstrom, "The Post-Game analysis framework—developing resource management strategies for concurrent systems," *IEEE Trans. Knowledge and Data Engineering*, vol. 1, no. 3, pp. 293-309, Sept. 1989.

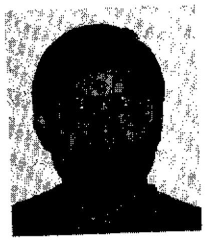
[33] C.F. Yu and B.W. Wah, "Learning dominance relations in combinatorial search problems," *IEEE Trans. Software Engineering*, vol. 14, no. 8, pp. 1,155-1,175, Aug. 1988.

Benjamin W. Wah received his PhD degree in computer science from the University of California, Berkeley, in 1979. He is currently a professor in the Department of Electrical and Computer Engineering and the Coordinated Science Laboratory of the University of Illinois at Urbana-Champaign. Previously, he served on the faculty of Purdue University (1979-85), as a Program Director at the National Science Foundation (1988-89), as Fujitsu Visiting Chair Professor of Intelligence Engineering, University of Tokyo (1992), and McKay Visiting Professor of Electrical Engineering and Computer Science, University of California, Berkeley (1994) In 1989, he was awarded a University Scholar of the University of Illinois. His current research interests are in the areas of parallel and distributed processing, data and knowledge base management, artificial intelligence, and nonlinear optimization.

Dr. Wah is editor-in-chief of *IEEE Transactions on Knowledge and Data Engineering*, and serves on the editorial boards of *Journal of Parallel and Distributed Computing*, *Information Sciences*, *International Journal on Artificial Intelligence Tools*, and *Journal of VLSI Signal Processing*. He has chaired a number of conferences and will chair the 1996 International Conference on Neural Networks. He served the IEEE Computer Society as a member of its Governing Board, and is currently serving on the IEEE-CS Publications Board, Press Activities Board, and Fellows Committee, and as a society representative to the IEEE Neural Network Council.



Arthur Ieumwananonthachai received his BS degree in electrical engineering and computer science from the University of Washingtion, Seattle, in 1986; and his MS degree in computer science from the University of California, Los Angeles, in 1988. Since then, he has been working toward his PhD degree in electrical and computer engineering at the University of Illinois, Urbana-Champaign, under the supervision of Prof. Benjamin Wah. His research interests include computer networks, distributed systems, and machine learning.



Lon-Chan Chu received his BS degree in electrical engineering from the National Taiwan University in 1985, and his MS and PhD degrees in electrical and computer engineering from the University of Illinois at Urbana-Champaign in 1991 and 1994, respectively. He joined Microsoft Corporation in 1993. His research interests include real-time intelligent systems and real-time embedded systems.

Akiko N. Aizawa received her BS degree in 1985, MS degree in 1987, and PhD degree in 1990, all in electrical engineering from the University of Tokyo. Since 1990, she has been with NACSIS (National Center for Science Information System), Japan. During 1990-92, she was a visiting researcher at the University of Illinois at Urbana-Champaign. Her research interests include stochastic optimization methods, knowledge engineering, networked information, and communication protocols.