5

Generalization of Heuristics Learned in Genetics-Based Learning

Benjamin W. Wah, Arthur Ieumwananonthachai, and Yong-Cheng Li

Abstract In this chapter, we study methods for developing general heuristics in order to solve problems in knowledge-lean application domains with a large and possibly infinite problem space. Our approach is based on genetic learning that generates heuristics, tests each to a limited extent, and prunes unpromising ones from further consideration. We summarize possible sources of anomalies in performance evaluation of heuristics along with our methods for coping with them. Based on the heuristics learned, we propose and study methods for generalizing heuristics to unlearned problem domains. Our method uses a new statistical measure called probability of win, which assesses the performance of heuristics in a distribution-independent manner. To validate our approach, we show experimental results on generalizing heuristics learned for sequential circuit testing, VLSI cell placement and routing, and branch-and-bound search. We show that generalization can lead to new and robust heuristics that perform well across problem instances of different characteristics in an application domain.

5.1 Introduction

The design of problem solving algorithms for many applications generally relies on the expertise of designers and the amount of domain knowledge available. The design is difficult when there is little domain knowledge or when the environment under consideration is different from which the algorithm is applied. In designing efficient algorithms for these knowledge-lean applications, there are two important problems to be considered: (1) automated design of problem solving heuristics,

and (2) systematic generalization of the heuristics learned. In this chapter, we focus on the second problem and present new methods for generalizing heuristics learned.

A *problem solver* can be optimal or heuristic. An optimal problem solver is a realization of an optimal algorithm that solves the problem for which it was designed 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 chapter 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 [14]. Newell et al. defined heuristics as "A process that may solve a given problem, but offers no guarantees of doing so" [12]. Pearl defined heuristics as "Strategies using readily accessible though loosely applicable information to control problem-solving processes in human being and machines" [14]. In this chapter, we define a *heuristic method (HM)* to mean a problem-solving procedure in a problem solver. Without loss of generality, a HM can be considered as a collection of interrelated *heuristic decision elements (HDE)* or *heuristics decision rules*. As illustrated in Figure 5.1, a heuristic problem solver takes a problem instance (or test case) and generates a possibly suboptimal solution.

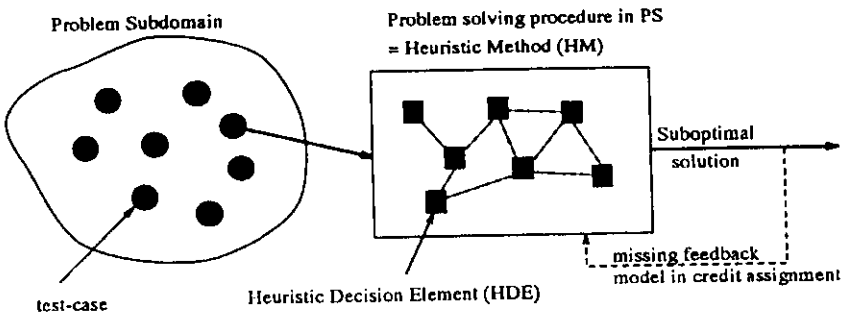

FIGURE 5.1
A heuristic method applied to a problem instance in a knowledge-lean application.

Heuristics are usually designed by human experts with strong expertise in the target application domain, or by automated systems using machine learning techniques. Both approaches focus on explaining the relation between heuristics and their performance, and on generating "good" heuristics based on observed information or explained relations. Regardless of the acquisition methods, heuristics in most cases are obtained by experimenting on a subset of test cases that cover a small portion of the problem space. For many applications, the problem space to be traversed is large, and there may be different regions that can best be solved by

different heuristics. For this reason, we study in this chapter methods for generalizing heuristics learned for a knowledge-lean application to regions of its problem space not seen during learning.

Designing heuristics for knowledge-lean applications is usually based on a generate-and-test paradigm. In the following, we present issues involved in this process and our assumptions made in this chapter.

5.1.1 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 heuristics to be designed. In a knowledge-rich domain, 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 Figure 5.1) to decide how new heuristics should be generated or how existing heuristics should be modified. Operators for composing new HMs for knowledge-lean domains are generally model-free, domain-independent, and syntactic in nature. A popular learning method using these operators is genetics-based learning.

Genetics-based learning is 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 applying genetic algorithms [15] to machine learning problems and is most suitable for learning performance-related HMs. (Performance-related HMs aim to improve the performance of solutions for an application problem, whose constraints are trivially satisfied.) This is true because genetic operators do not rely on domain knowledge in generating heuristics and are not able to generate (correctness-related) heuristics for applications whose constraints are hard to satisfy. Examples of genetics-based methods for learning performance-related heuristics include population-based learning [23] and genetic programming [9].

As our focus in this chapter is on generalization, we assume that a set of "good" heuristics for some subsets of the problem space of an application has been obtained. Regardless of the learning method used, it is clear that learning can only focus on a small portion of the problem space, and heuristics learned this way are restricted to domains studied in learning.

5.1.2 Testing of Heuristics and Evaluating Their Performance

To evaluate HMs in a knowledge-lean application domain, they must be tested on a set of problem instances (or test cases). Performance in this case is generally assessed in a statistical sense, without requiring exhaustive tests of all test cases.

In this chapter, we are interested in two types of problem domains: (1) those with a large number of test cases and possibly an infinite number of deterministic HMs for solving them, and (2) 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 a HM is nondeterministic, requiring multiple evaluations of the HM on different test cases (type 1) or multiple evaluations of the HM on the same test case (type 2). 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 a HM can be evaluated statistically. Moreover, we must also deal with conditions in which performance values of HMs can have different ranges within a subset of test cases and have entirely different distributions across multiple subsets. We present in Section 5.2 issues on selecting appropriate methods for coping with anomalies[1] in performance evaluation of heuristics.

5.1.3 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 developed to test cases not studied during learning. Generalization is difficult when HMs do not perform consistently or have different ranges of performance across different test cases. This issue has been somewhat ignored in the literature on genetic algorithms and must be addressed in order to find general and efficient HMs for solving a wide range of problem instances of an application. This issue is the focus of this chapter and is presented in Section 5.3.

In short, we study in this chapter methods for generalizing performance-related heuristics for knowledge-lean applications. We assume that performance of a HM is represented by one or more statistical metrics and is based on multiple evaluations of test cases (noisy evaluations). The major issues we study include methods to cope with inconsistencies in performance evaluation of heuristics (Section 5.2.3), and generalization of learned heuristics to unlearned domains (Section 5.3). Experimental results on several real-world applications are shown in Section 5.4. Heuristics for these applications were learned using TEACHER, a genetics-based learning system we had developed earlier [23].

[1] An anomaly means that one HM is better than another using one evaluation method, but worse using another evaluation method.

5.2 Performance Evaluation and Anomalies

In problem solving, a problem solver applies a sequence of decisions defined in HDEs of a 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 that is 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 to reach the final state.

The *performance* of a 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* (respectively, *cost*) of a solution with respect to an input test case to be one or more measures of how good the final state is (respectively, how expensive it is to reach the final state) when the test case is solved, and be independent of 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 examples of *performance measures* of the application.

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

5.2.1 Example Applications

In problem solving, a problem solver applies some general domain-independent algorithms, that rely on information provided by domain-dependent heuristics to make decisions and to change the current state of the application environment. Table 5.1 shows examples of a few practical applications and their domain-dependent heuristics. We use the first three applications to test our generalization strategy in this chapter. Results on the generalizing HMs in process mapping [23], load balancing [11], and stereo vision [18] have been presented elsewhere.

The first application we have studied is based on two genetic-algorithm packages (CRIS [17] and GATEST [16]) for generating test patterns in VLSI circuit testing. Both packages use a domain-independent genetic algorithm [15] that continuously evolves test patterns by analyzing mutated vectors on their ability to identify (or cover) more faults in a circuit. There are a lot of possible domain-dependent HMs; however, in our experiments, we chose the domain-dependent HMs for CRIS as

Table 5.1 Examples of Knowledge-Lean Applications and Their Learnable Domain-Dependent Heuristics

Application	Objective(s)	Domain-dependent parametric-heuristics	Heuristic element(s)	Example(s) of element
Genetic search of the best VLSI test sequence [16, 17]	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, 4, 3.2, 100), H(•)
Simulated annealing:Timber Wolf [19]	Minimize area of layout with fixed maximum number of layers	If ([acceptance ratio] > [threshold]), then reduce temperature to next lower value	Numerical threshold value, cost function, temperature function	0.9, C(•), T(•)
Branch-and-bound search for finding a minimum-cost tour in a graph	Minimize cost of tour, satisfy 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
Process mapping for placing a set of processes on a multicomputer [6]	Minimize overall completion time, 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 [11]	Minimize completion time of an incoming job	If (average WL[•] > [threshold]), then migrate this process	Workload function WL, numeric threshold value	WL(•), 2.0
Stereo vision for depth perception [18]	Minimize error in range estimation	Marr and Poggio's iterative algorithm	(Channel width, low edge-detection threshold, high threshold)	(0.6, 2.0, 5.0)
Designing a blind equalizer	Minimize convergence time, accumulated errors and cost, maximize S/N ratio	Objective (error) function for gradient descent	Symbolic formula of the error function	E(•)

a set of seven numeric parameters and for GATEST [16], one of the four fitness functions for computing the performance of test patterns. Our results here extend the performance results we have found earlier for CRIS [23] and show the ability of our learning and generalization procedures to result in higher fault coverages than the original packages.

The second application we consider is TimberWolf (Version 6), [19, 20] a software package based on simulated annealing for placing and routing a set of VLSI circuit components. We illustrate that, by tuning a set of six numeric parameters in TimberWolf, we can reduce the area of the chip as well as the time needed to find the layout. Results on extending our results on TimberWolf's cost and temperature-control functions will be shown in a future publication.

The last application is a software package WISE [21] that implements a branch-and-bound search to find optimal solutions of three combinatorial optimization problems (vertex cover, asymmetric traveling salesman, and knapsack packing). In this case, the branch-and-bound search is domain-independent, and we chose as domain-dependent heuristics the decomposition HM in the search algorithm. The decomposition HM is used to pick an attribute to decompose a subproblem in a search tree into descendants. For instance, in a vertex-cover problem, the goal is to find the minimum number of nodes of a graph so that each edge is emanating from one of the covered nodes. In this case, a subproblem represents a set of nodes in the graph to cover partially the edges in the graph, and the decomposition HM picks the next node to be included in the covered set. The HM is represented as a symbolic formula of parameters that can be obtained in the search tree.

5.2.2 Problem Subspace and Subdomain

Within an application domain, different regions of the problem space may have different characteristics, each of which can best be solved by a unique HM [15]. 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 a problem subspace and a 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 application problems. It requires knowing one or more attributes to classify test cases and is driven by a set of decision rules that identify the subspace to which a test case belongs. Such partitioning is important when test cases in an application have vastly different behavior. However, in some cases, it may not be possible to define the attributes needed for partitioning, or the number of attributes is too large. When these happen, nonparametric clustering methods, such as those based on neural networks, may have to be used. Another possibility is to always apply multiple HMs for each test case, resulting in a higher computational cost for a better solution.

For instance, consider solving a vertex-cover problem described in the last subsection. In designing a decomposition HM to decide 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 (in an *ad hoc* fashion) into a small number of subspaces based on graph connectivities and learned independently. As another example, in generating test patterns for VLSI circuits, 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. However, we are not able to partition the subspace of sequential circuits into smaller subspaces as it is not clear which attributes should be used in this partitioning.

Given a subspace of test cases, we next define a subdomain. A *problem subdomain* in this chapter 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. Recall that in our definition, test cases belong to the same subspace when they can be solved by the same HM, but this does not imply that their performance can be compared directly. 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, a subdomain is a maximal partitioning of test cases in a subspace so that different HMs can be compared quantitatively based on their aggregate metrics. It is important to point out that performance values may need to be normalized before aggregated statistically. We discuss issues related to normalization in the next subsection.

In the same way that test cases are partitioned into subspaces, minimal domain knowledge should be used in knowledge-lean applications to partition test cases into subdomains. Further, the requirement of the statistical metric for quantitative performance comparison must be satisfied. For instance, performance values need to be independent and identically distributed (i.i.d.) when the average metric is computed.

Continuing with the example on the vertex-cover problem, a problem subdomain can be defined as random graphs with a certain degree of connectivity. The performance of a decomposition HM can be defined as the number of subproblems expanded by a branch-and-bound search normalized with respect to that of a baseline HM. Proper normalization (to be discussed in the next subsection) allows meaningful statistical comparison of HMs within a subdomain. A possible goal in learning for a subdomain is, therefore, to develop a rank ordered set of HMs, based on the average fraction of vertices to cover all the edges of random graphs generated in this subdomain. As another example, in generating test patterns for VLSI circuits, we may have to treat each circuit as an individual subdomain as we do not know the best set of attributes to classify circuits.

It should now be clear that, in general, there are infinitely many subdomains in an application problem, and learning can only be performed on a small number of them. Consequently, generalization of HMs learned for a small number of subdomains to other subdomains in the same subspace is critical. Informally, generalization entails finding a good HM for solving a randomly chosen test case in a subspace so that this HM has a high probability of performing better than other competing HMs for solving this test case. In some situations, multiple HMs may have to be identified and applied together at a higher cost to find a solution of higher quality. We discuss generalization in detail in Section 5.3.

To illustrate the concepts presented in this chapter, we show in Figure 5.2 the average symmetric speedups of three decomposition HMs used in a branch-and-bound search to solve vertex-cover problems. (The use of symmetric speedup is defined in Equation 5.2 later.) 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 find the five best HMs for each of three subdomains with connectivities 0.1, 0.35, and 0.6. Figure 5.2 shows the performance of the best HMs learned in each subdomain across all subdomains. We have also identified a single generalized HM among the 15 HMs learned using the method discussed in Section 5.3 and show its performance in Figure 5.2. We found that the generalized HM is not the top HM learned in each subdomain, indicating that the best HM in each subdomain may be too specialized to the subdomain. We have also found that generalization is possible in terms of average performance. We need to point out that the average performance should not be used as the sole indicator, as the variances of performance may differ from one subdomain to another.

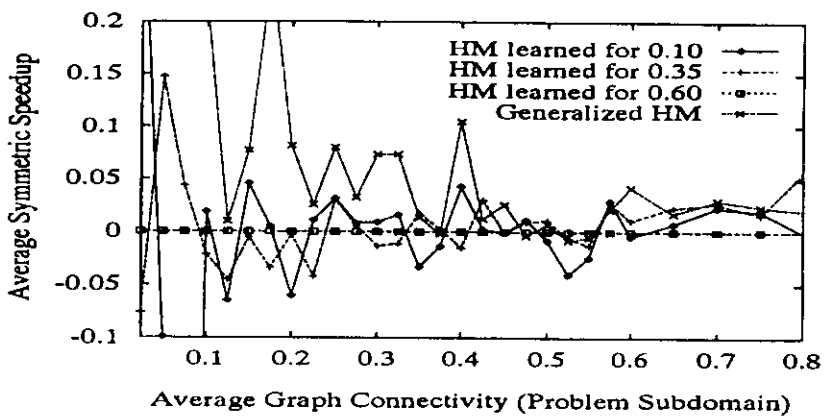

FIGURE 5.2

Average speedups (over 15 test cases) of three decomposition HMs for the vertex-cover problem, where subdomains are manually selected based on the connectivity of a graph. The HM learned for 0.6 connectivity is the same as the baseline HM. (Details of experiments are shown in Section 5.4.3)

5.2.3 Anomalies in Performance Evaluation

To design a good and general HM for an application, we must be able to compare HMs in terms of their performance across multiple problem subdomains within the same problem subspace. There are two steps in accomplishing this task. First, we must be able to compare HMs in terms of their performance within a single subdomain. Second, we must be able to compare the performance of HMs across multiple subdomains. Accomplishing the first step is necessary before we can deal with generalization in the second step. We have previously studied parts of the problem of evaluating performance of HMs within a single problem subdomain [22]. In this section, we summarize issues involved in the first step before presenting issues in the second step.

5.2.3.1 Anomalies within a Subdomain

Recall that HMs studied in this chapter have nondeterministic performance, implying the need to evaluate each HM multiple times in a subdomain. Further, performance may be made up of multiple measures (such as cost and quality) that are interrelated: higher quality is generally associated with higher cost.

To compare performance of different HMs, it is necessary to combine performance values before comparing them. This is, however, difficult, as the objectives of a 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 [22]:

- 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 [4].
- 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 the HMs are compared using either different user-defined objective functions or the same objective function with different parameters.

We have proposed before [23] three solutions to cope with these difficulties:

- Identify a *reference* or baseline HM upon which all other HMs are compared. A good choice of a reference method for a given application problem is the best existing HM.
- 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.

- Compare two HMs based on individual normalized performance measures, not on a single parametric function of the measures.

In this section, we extend the anomalies found earlier [22] and classify all the anomalies into three classes. Note that anomalies happen because there is more than one dimension of performance variations.

a) Inconsistencies in Performance of HMs across Different Test Cases When a HM is evaluated on a set of test cases, we must determine (1) the number of tests to be made and (2) the evaluation method (or metric) for aggregating performance values (such as mean, maximum, median, 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.

For example, when performance is evaluated by the average metric, the ranking of HMs may change as more tests are performed. Similar observations have been found when HMs are evaluated by the average-rank metric.

In this chapter, we assume that all HMs are tested on the same set of test cases in a subdomain when evaluating generalizability, that performance values of tests in a subdomain are i.i.d., and that the average metric is used as the primary method for comparing HMs. In addition, we evaluate the actual distribution of performance values to avoid HMs that have good average behavior but have large spread in performance. When none of the HMs is a clear winner, our system will propose alternative HMs so that users can decide the appropriate HM(s) to use.

b) Multiple Objectives with Unknown Trade-Offs The performance of a 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, as improved quality is often associated with increased cost. The problem, generally known as a *multiobjective optimization problem* [4], involves trade-offs among all objectives of a design.

We have found before that anomalies may happen when HMs are optimized with respect to a *single* parametric function of the objectives of an application [7, 22]. The performance of a HM (as defined by the parametric objective function) may change drastically when minor changes are made on the parameters of the objective function. In fact, it is possible to show that one HM is better than another by finding a new parametric objective function of performance measures.

To avoid inconsistencies due to multiple objectives, we must evaluate HMs based on individual performance measures and not combine multiple measures into a single parametric function [22]. During learning, the learning system should constrain all but one measure and optimize the single, unconstrained measure.

A HM is pruned from further testing when one of its performance constraints is violated [23]. If a good HM satisfying the constraints can be found, then the constraints are further refined and learning is repeated. The difficulty with this approach is on setting constraints. We have studied the case when there are two performance measures [23]. However, the general case when there are more than two performance measures is still open at this time.

The applications we have studied in this chapter have only one performance measure to be optimized; hence, we do not need to deal with the issue of multiple objectives. (For GATEST, CRIS, and TimberWolf, the performance measure to be optimized is the quality of the result; for branch-and-bound search, the measure to be optimized is the cost of finding the optimal solution.)

c) Inconsistencies in Normalization Normalization involves choosing a baseline HM and computing relative performance values of a new HM on a set of tests 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 1 as discussed in Section 5.1) and is not needed when nondeterminism in performance is due to randomness in the problem solver (type 2 as discussed in Section 5.1). In the former case, performance values 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 the 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 or when normalization compresses/decompresses the range of performance values. These anomalies are explained as follows.

Inconsistencies in evaluation may occur when using multiple normalization methods as compared to using one normalization method. To illustrate these inconsistencies, consider with two HMs that have completion times {1475, 1665, 1381} for HM_1 and {1269, 1513, 1988} for HM_2 (Example A). Using HM_1 as the baseline for normalization, we can compute the average normalized speedup of HM_2 using one of the following methods:

$$\bar{q}_2^n = \sum_{j=1}^{3} \frac{t_{1,j}}{t_{2,j}} = 0.986; \qquad \bar{Q}_2^n = \sum_{j=1}^{3} \frac{t_{1,j} - 1200}{t_{2,j} - 1200} = 1.900 \tag{5.1}$$

where $t_{i,j}$ is the completion time of HM i on test case j. 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 conventional 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). As another example (Example B), suppose the speedups of a 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. In this case, both the average speedups and average slowdowns are greater than one.

In general, when normalizing performance values, it is important to recognize that the ordering of HMs may change when using a different normalization method, and that the spread of (normalized) 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 outlyers 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 degradation. To simplify understanding, we describe this symmetric normalization method using the speedup measure. We define *symmetric speedup* as

$$Speedup_{symmetric} = \begin{cases} Speedup - 1 & \text{if } Speedup \geq 1 \\ 1 - \frac{1}{Speedup} & \text{if } 1 > Speedup \geq 0 \end{cases} \tag{5.2}$$

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 5.2 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

$$\begin{aligned} Slowdown_{symmetric} &= \begin{cases} Slowdown - 1 & \text{if } Slowdown \geq 1 \\ 1 - \frac{1}{Slowdown} & \text{if } 1 > Slowdown \geq 0 \end{cases} \\ &= -Speedup_{symmetric} \end{aligned} \tag{5.3}$$

It is also 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. (The average symmetric slowdown is 0.059.) 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 illustrate the difference between speedups and symmetric speedups, we show in Figure 5.3 the distributions of speedups as well as symmetric speedups of a HM to solve the vertex-cover problem.

Table 5.2 Inconsistent Behavior of HMs in Different Subdomains

Circuit		Heuristic method			Fault coverages (%)	
		Random seeds used in HM				
ID	ID	61801	98052	15213	Maximum	Average
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

5.2.3.2 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 the three HMs shown in Figure 5.2. 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. As another example, consider the HMs learned for CRIS (described in Section 5.2) [17]. A HM in this case is a vector of seven parameters and a random seed; by varying the random seed, we get different performance of the HM. Since it is difficult to characterize circuits with respect to their HMs, we treat the subspace to be all possible circuits, and each circuit as a subdomain for learning. The goal of generalization is to find one single set of parameters (HM) that gives high fault coverages for all circuits. In Table 5.2, we show two HMs and their fault coverages. With respect to circuit $s444$, HM_{101} has worse fault coverages and a wider distribution of coverage values than HM_{535}. On the other hand, HM_{101} performs better for circuit $s1196$.

The major difficulty in handling multiple problem subdomains is that performance values from different subdomains cannot be aggregated statistically. For instance, it is not meaningful to find the average of two different distributions. 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 Section 5.2.3.1.c. 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

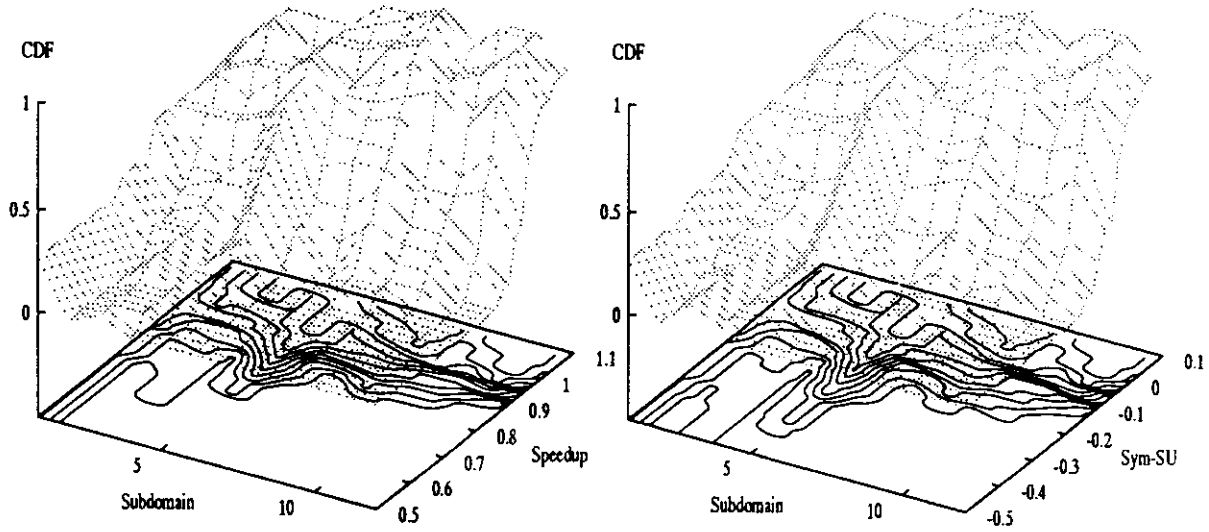

FIGURE 5.3
Distribution of performance values of one HM on 15 test cases based on speedup and symmetric speedup for solving the vertex-cover problem.

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 5.3 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 a HM in one subdomain is better than the true mean performance of another randomly selected HM in the same subdomain. The advantage of using probabilities of win is that they are in the range between zero and one, independent of the number of HMs evaluated and the range and distribution of performance values.

5.3 Generalization of Heuristic Methods Learned

The ability to generalize a learned HM to test cases in new problem subdomains is one of the key reasons for learning. 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 the baseline HM (if it exists). To achieve this goal, four issues are apparent here.

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

5.3.1 Probability of Win within a Subdomain

There are many ways to address issue (1) raised in this section, and the solutions to the remaining problems depend on this solution. As discussed at the end of the last section, ranking, scaling, and normalization do not work well. In this section, we propose a metric called probability of win to select good HMs across multiple subdomains.

P_{win}, the *probability-of-win* of HM h_i within a subdomain, is defined as the

probability that the true mean of h_i (with respect to one performance measure) is better than the true mean of HM h_j randomly selected from the pool. When h_i is applied on test cases in subdomain d_m, we have

$$P_{win}(h_i, d_m) = \frac{1}{|s| - 1} \sum_{j \neq i} P\left[\mu_i^m > \mu_j^m \left| \hat{\mu}_i^m, \hat{\sigma}_i^m, n_i^m, \hat{\mu}_j^m, \hat{\sigma}_j^m, n_j^m \right.\right] \quad (5.4)$$

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 h_i in d_m.

Since we are using the average performance metric, it is a good approximation to use the normal distribution as a distribution of the sample average. The probability that h_i is better than h_j in d_m can now be computed as follows.

$$P\left(\mu_i^m > \mu_j^m \left| \hat{\mu}_i^m, \hat{\sigma}_i^m, n_i^m, \hat{\mu}_j^m, \hat{\sigma}_j^m, n_j^m \right.\right) = 1 - \Phi\left[\frac{\hat{\mu}_i^m - \hat{\mu}_j^m}{\sqrt{\hat{\sigma}_i^{m2}/n_i^m + \hat{\sigma}_j^{m2}/n_j^m}}\right] \quad (5.5)$$

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

Table 5.3 Probabilities of Win of Four HMs

h_1	$\hat{\mu}_i$	$\hat{\sigma}_i$	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

To illustrate the concept, we show in Table 5.3 probabilities of win of four HMs tested to various degrees. Note that the probability of win is not directly related to sample mean, but rather depends on sample mean, sample variance, and number of tests performed.

P_{win} defined in Equation 5.4 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.

5.3.2 Probability of Win across Subdomains

The use of probability of win in a subdomain leads to two ways to solve issue (2) posted earlier in this section, namely, how to define the notion that one HM performs better than another HM across multiple subdomains. We present below two ways to define this notion.

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.

$$P_{win}(h, \Pi_p) = \frac{\sum_{d \in \Pi_p} P_{win}(h, d)}{|\Pi_p|} \tag{5.6}$$

where Π_p is the pth partition of subdomains in the subspace. The HM picked is the one that maximizes Equation 5.6. HMs picked this way usually win with a high probability across most of the subdomains in Π_p, but occasionally may not perform well in a few subdomains.

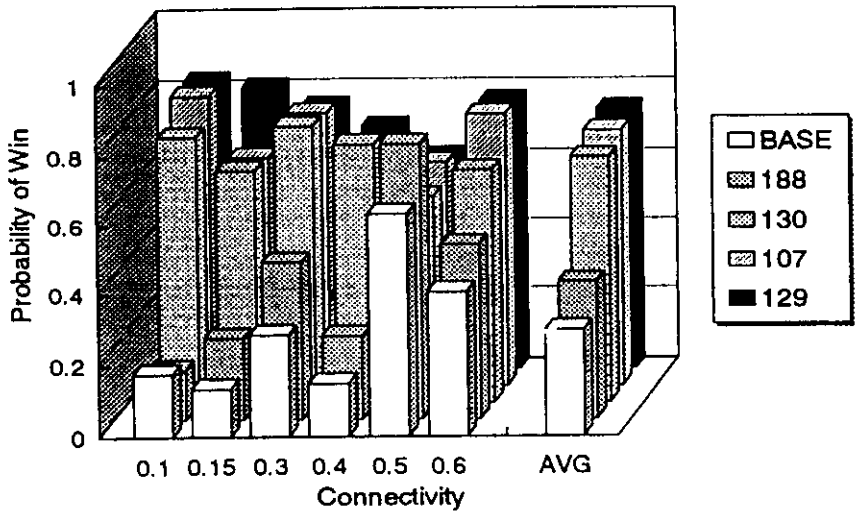

FIGURE 5.4
Histogram showing probabilities of win of four HMs generalized across six subdomains and those of the baseline HM. (HM_{129} is picked if Equation 5.6 is used as the selection criterion; HM_{107} is selected if Equation 5.7 is used as the criterion.)

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 5.2.3, evaluating HMs based on a combined objective function (such as the

average probability of win in Equation 5.6) 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 $m \in \Pi_p$.

$$P_{win}(h, m) \geq \left(q^m_{win} - \delta\right) \quad \text{for every } m \in \Pi_p \tag{5.7}$$

In this formulation, δ may need to be 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 5.2. Assume that learning had been performed on six subdomains (using graphs with connectivities 0.1, 0.15, 0.3, 0.4, 0.5, and 0.6, respectively), and that the five best decomposition HMs from each subdomain were reported. After full evaluation of the 30 HMs across all six subdomains, we computed the probability of win of each HM in every subdomain. Figure 5.4 shows the probabilities of win of several of these HMs. If we generalize HMs based on Equation 5.6, then HM_{129} will be picked since it has the highest average P_{win}. In contrast, if we generalize using Equation 5.7, then HM_{107} will be picked because it has the smallest deviation from the maximum P_{win} in each subdomain. Note that either HM is a reasonable generalized HM to be applied across all subdomains. To decide on the single generalized HM to use, further evaluations on the spread of performance values would be necessary (see Section 5.4).

Using probabilities of win to assess HMs across subdomains, we can now address issues 3 and 4 raised earlier in this section, which deal with the selection of multiple HMs. There are two ways that multiple HMs can be used: (1) using each HM in a nonoverlapping subset of subdomains in the subspace (issue 3), and (2) applying multiple HMs in solving any test case in the subspace (issue 4).

The issue on finding condition(s) under which a specific HM should be applied is a difficult open problem at this time. 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 a unique HM. This is possible in some applications that are characterized by a small number of well-defined attributes. For instance, in the vertex-cover problem discussed in Sections 5.2 and 5.4, the graph connectivity is a unique attribute that allows us to decompose the space of all random graphs into partitions. This is, however, difficult for applications that do not have a well-defined set of characteristic attributes. As a result, we cannot partition the subspace and assign a good HM to each. This happens in the CRIS test-pattern generation system [17] discussed in Sections 5.2 and 5.4.1 which does not have a set of characteristic attributes to classify circuits (e.g., number of gates, length of the longest path).

Finally, issue (4) raised earlier in this section is on trade-offs between cost and quality in generalization. Since it may be difficult in some cases to partition a subspace or to find a single HM that performs well across all test cases 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 practical 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 Equation 5.6. 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,

$$P_{win}^{\max}(\Omega, \Pi) = \frac{\max\limits_{\substack{H \subset \Omega \\ |H|=\text{constant}}} \sum\limits_{d \in \Pi} \max\limits_{h \in H} P_{win}(h, d)}{|\Pi|} \tag{5.8}$$

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

One way to find H in Equation 5.8 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 [3]: 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 complete, in general, and is solvable in polynomial time only when $|H|$ is two.

Fortunately, by applying Equation 5.7, 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 5.4.1.

5.3.3 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 Equation 5.4) of each HM in every subdomain learned or to be generalized.

2. Apply Equation 5.8 to select the necessary number of HMs for evaluating test cases in the subspace. Equation 5.7 can be used to restrict the set of HMs considered in the selection process.

5.4 Experimental Results

To illustrate the generalization procedure described in Section 5.3.3, we present in this section results on generalization for the three applications discussed in Section 5.2.1.

5.4.1 Heuristics for Sequential Circuit Testing

The first application is based on CRIS [17] and GATEST [16], two genetic-algorithm software packages for generating patterns to test sequential VLSI circuits. In our experiments, we used sequential circuits from the ISCAS89 benchmarks [1] plus several other larger circuits. Since these circuits are from different applications, it is difficult to classify them by some common features. Consequently, we treat each circuit as an individual subdomain. As we like to find one common HM for all circuits, we assume that all circuits are from one subspace.

5.4.1.1 CRIS

CRIS [17] is based on continuous mutations of a given input test sequence and on analyzing the mutated vectors for selecting a test set. Hierarchical simulation techniques are used in the system to reduce memory requirement, thereby allowing test generations for large VLSI circuits. The package has been applied successfully to generate test patterns with high fault coverages for large combinatorial and sequential circuits.

CRIS in our experiments is treated as a problem solver in a black box, as we have minimal knowledge in its design. A HM targeted for improvement is a set of eight parameters used in CRIS (see Table 5.4). 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 proper values for 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 manual tuning were exceedingly high and were not reported in the Reference [17].

Table 5.4 Parameters in CRIS Treated as a HM in Learning and in Generalization

Parameter	Type	Range	Step	Definition	Learned value
P_1	Integer	1–10	1	Related to the number of stages in a flip-flop	1
P_2	Integer	1–40	1	Related to the sensitivity of changes of state of a flip-flop (number of times a flip-flop changes its state in a test sequence)	12
P_3	Integer	1–40	1	Selection criterion—related to the survival rate of a candidate test sequence in the next generation	38
P_4	Float	0.1–10.0	0.1	Related to the number of test vectors concatenated to form a new test sequence	7.06
P_5	Integer	50–800	10	Related to the number of useless trials before quitting	623
P_6	Integer	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	Integer	Any	1	Seed for the random number generator	—

Note: The type, range, and step of each parameter were recommended to us by the designer of CRIS. The default parameters were not given to us as they are circuit-dependent.

Our goal 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.

5.4.1.2 GATEST

GATEST [16] is another test-pattern generation package based on genetic algorithms. It augments existing techniques in order to reduce execution times and to improve fault coverages. The genetic-algorithm component evolves candidate test vectors and sequences, using a fault simulator to compute the fitness of each candidate test. To improve performance, the designers manually tuned various genetic-algorithm parameters in the package (including alphabet size, fitness function, generation gap, population size, and mutation rate) as well as selection and crossover schemes. High fault coverages were obtained for most of the ISCAS89

sequential benchmark circuits [1], and execution times were significantly lower in most cases than those obtained by HITEC [13], a deterministic test-pattern generator.

The entire GA process was divided into four phases, each with its own fitness function that had been tuned by the designers manually. The designers also told us that phase 2 has the largest impact on performance and recommended that we improved it first. As a result, we treat GATEST as our problem solver, and the fitness function (a symbolic formula) in phase 2 as our HM. The original form of this fitness function is

$$fitness_2 = \#\_faults\_detected + \frac{\#\_faults\_propagated\_to\_flip\_flops}{(\#\_faults)\,(\#\_flip\_flops)} \tag{5.9}$$

In learning a new fitness function, we have used the following variables as possible arguments of the function: *#_faults_detected, #_faults_propagated_to_flip_flops, #_faults, #_flip flops, #_circuit nodes,* and *sequence_length.* The operators allowed to compose new fitness functions include $\{+, -, *, /\}$.

5.4.1.3 Experimental Results

In our experiments, we chose five circuits as our learning subdomains. In each of these subdomains, we used TEACHER [23] to test CRIS 1000 times with different HMs, each represented as the first seven parameters in Table 5.4. At the end of learning, we picked the top 20 HMs and evaluated them fully by initializing CRIS by ten different random seeds (P_8 in Table 5.4). We then selected the top five HMs from each subdomain, resulting in a total of 25 HMs supplied to the generalization phase. We evaluated the 25 HMs fully (each with ten random seeds) on the five subdomains used in learning and five new subdomains. We then selected one generalized HM to be used across all the ten circuits (based on Equation 5.8). The elements of the generalized HM found are shown in Table 5.4.

For GATEST, we applied learning to find good HMs for six circuits (s298, s386, s526, s820, s1196, and s1488 in the ISCAS89 benchmark). We then generalized the best 30 HMs (5 from each subdomain) by first evaluating them fully (each with ten random seeds) on the six subdomains and by selecting one generalized HM for all circuits (using Equation 5.8). The final fitness function we got after generalization is

$$fitness_2 = 2 \times \#\_faults\_propagated\_to\_flip\_flops - \#\_faults\_detected. \tag{5.10}$$

Table 5.5 shows the results after generalization for CRIS and GATEST. For each circuit, we present the average and maximum fault coverages (over ten random seeds) and the corresponding computational costs. These fault coverages are compared against the published fault coverages of CRIS [17] and GATEST [16] as well as those of HITEC [13]. Note that the maximum fault coverages reported in

Table 5.5 Performance of HMs in Terms of Computational Cost and Fault Coverage for CRIS and GATEST

Circuit ID	Total faults	Fault coverage				Cost		CRIS generalized HM			GATEST generalized HM		
		HITEC	CRIS	Avg GATEST	Max GATEST	HITEC	Avg GATEST	Avg FC	Max FC	Avg cost	Avg FC	Max FC	Avg cost
*s298	308	86.0	82.1	85.9	86.0	15,984.0	128.6	84.7	86.4	10.9	85.9	86.0	126.4
s344	342	95.9	93.7	96.2	96.2	4,788.0	134.8	96.1	96.2	21.8	96.2	96.2	133.3
s349	350	95.7	—	95.7	95.7	3,132.0	136.9	95.6	95.7	21.9	95.7	95.7	128.3
+s382	399	90.9	68.6	87.0	87.5	43,200.0	203.3	72.4	87.0	7.2	87.0	87.5	208.9
s386	384	81.7	76.0	76.9	77.9	61.8	67.6	77.5	78.9	3.5	78.6	79.3	78.6
*s400	426	89.9	84.7	85.7	86.6	43,560.0	229.3	71.2	85.7	8.4	85.7	86.6	215.1
s444	474	87.3	83.7	85.6	86.3	57,960.0	259.4	79.8	85.4	9.3	85.6	86.3	233.8
*s526	555	65.7	77.1	75.1	76.4	168,480.0	333.4	70.0	77.1	10.0	75.5	77.3	302.7
s641	467	86.5	85.2	86.5	86.5	1,080.0	181.2	85.0	86.1	19.5	86.5	86.5	195.0
+s713	581	81.9	81.7	81.9	81.9	91.2	219.9	81.3	81.9	23.0	81.9	81.9	256.5
s820	850	95.6	53.1	60.8	68.0	5,796.0	266.4	44.7	46.7	51.3	69.3	80.9	225.4
*s832	870	93.9	42.5	61.9	66.8	6,336.0	265.8	44.1	45.6	44.6	66.9	72.8	251.0
s1196	1,242	99.7	95.0	99.2	99.5	91.8	292.1	92.0	94.1	20.0	99.2	99.4	421.7
*s1238	1,355	94.6	90.7	94.0	94.4	132.0	380.5	88.2	89.2	23.0	94.0	94.2	585.2
s1488	1,486	97.0	91.2	93.7	96.0	12,960.0	512.3	94.1	95.2	85.6	94.3	96.5	553.4
+s1494	1,506	96.4	90.1	94.0	95.8	6,876.0	510.4	93.2	94.1	85.5	93.6	95.6	584.3
s1423	1,515	40.0	77.0	81.0	86.3	—	3,673.9	82.0	88.3	210.4	81.3	87.3	4,325.7
+s5378	4,603	70.3	65.8	69.5	70.1	—	9,973.3	65.3	69.9	501.8	69.6	71.9	8,875.7
s35932	39,094	89.3	88.2	89.5	89.7	13,680.0	184,316.0	77.9	78.4	4,265.7	89.4	89.7	184,417.0
am2910	2,573	85.0	83.0	—	—	—	—	83.7	85.2	307.6	—	—	—
+div16	2,147	72.0	75.0	—	—	—	—	79.1	81.0	149.9	—	—	—
tc100	1,979	80.6	70.8	—	—	—	—	72.6	75.9	163.8	—	—	—

Note: Learned subdomains for CRIS are marked by * and generalized subdomains by +. Performance of HITEC is from the literature [13, 17]. Costs of our experiments are running times in seconds on a Sun SparcStation 10/512; costs of HITEC are running times in seconds on a Sun SparcStation SLC [16] (around four to six times slower than a Sun SparcStation 10/512).

Table 5.5 were based on ten runs of the underlying problem solver, implying that the computational cost is ten times of the average cost.

Table 5.6 summarizes the improvements of our learned and generalized HMs as compared to the published results of CRIS, GATEST, and HITEC. Each entry of the table shows the number of times our HM wins, ties, and loses in terms of fault coverages with respect to the method(s) in the first column. Our results show that our generalized HM based on CRIS as the problem solver is better than the original CRIS in 16 out of 21 circuits in terms of the maximum fault coverage, and 11 out of 21 circuits in terms of the average fault coverage. Further, our generalized HM based on GATEST as the problem solver is better than the original GATEST in 7 out of 19 circuits in terms of both the average and maximum fault coverages. Our results show that our generalization procedure can discover good HMs that work better than the original HMs.

Finally, we plot the distributions of symmetric fault coverages of our generalized HMs normalized with respect to average fault coverages of the original CRIS (Figure 5.5) and GATEST (Figure 5.6). These plots clearly demonstrate improvements over the original systems.

5.4.2 Heuristics for VLSI Placement and Routing

In our second application, we take TimberWolf [20] as our problem solver. This is a software package based on simulated annealing [8] 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 the paths. 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 in this subsection, 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 in a single experiment to control the running 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 for setting them optimally. In Table 5.7 we list the parameters we have focused in this subsection. Our goal is to illustrate the power of our learning and generaliza-

Table 5.6 Summary of Results Comparing the Performance of our Generalized HMs with Respect to That of HITEC, CRIS, and GATEST

Our HM wins/ties/loses with respect to the following systems	CRIS generalized HM			GATEST generalized HM		
	Total	Max FC	Avg FC	Total	Max FC	Avg FC
HITEC	22	6, 2, 14	4, 0, 18	19	5+2, 2, 10	4+2, 1, 12
Original CRIS	21	16, 1, 4	11, 0, 10	18	18, 0, 0	17, 0, 1
Original GATEST	19	4, 3, 12	3, 0, 16	19	7+2, 7, 3	7+2, 8, 2
Both HITEC and CRIS	21	5, 2, 14	3, 0, 18	18	5+2, 1, 10	3+2, 1, 13
Both HITEC and GATEST	19	3, 3, 13	1, 0, 18	19	3+2, 4, 10	2+2, 2, 13
All HITEC, CRIS, and GATEST	18	2, 3, 13	1, 0, 17	18	3+2, 3, 10	1+2, 1, 14

Note: The first number in each entry shows the number of wins out of all applicable circuits, the second, the number of ties, and the third, the number of losses. A second number in the entry on wins indicates the number of circuits in which the test efficiency is already 100%. For these circuits, no further improvement is possible.

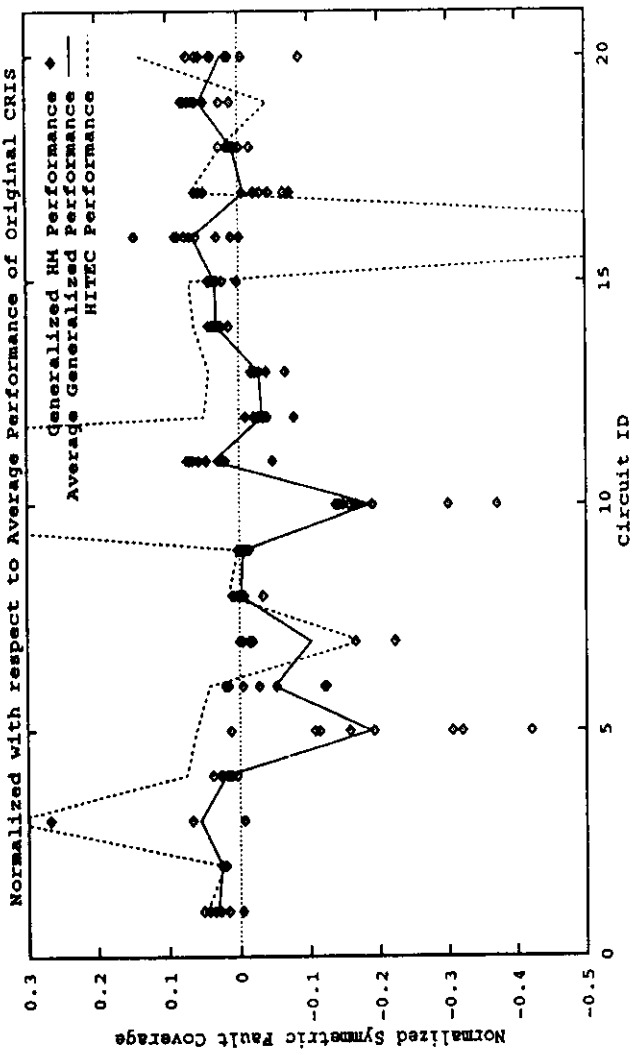

FIGURE 5.5
Distribution of normalized symmetric fault coverages of our generalized HM with respect to average fault coverages of the original CRIS on 20 benchmark circuits (s298, s344, s382, s386, s400, s444, s526, s641, s713, s820, s832, s1196, s1238, s1488, s1494, s1423, s5378, am2910, div16, and tc100 in that order).

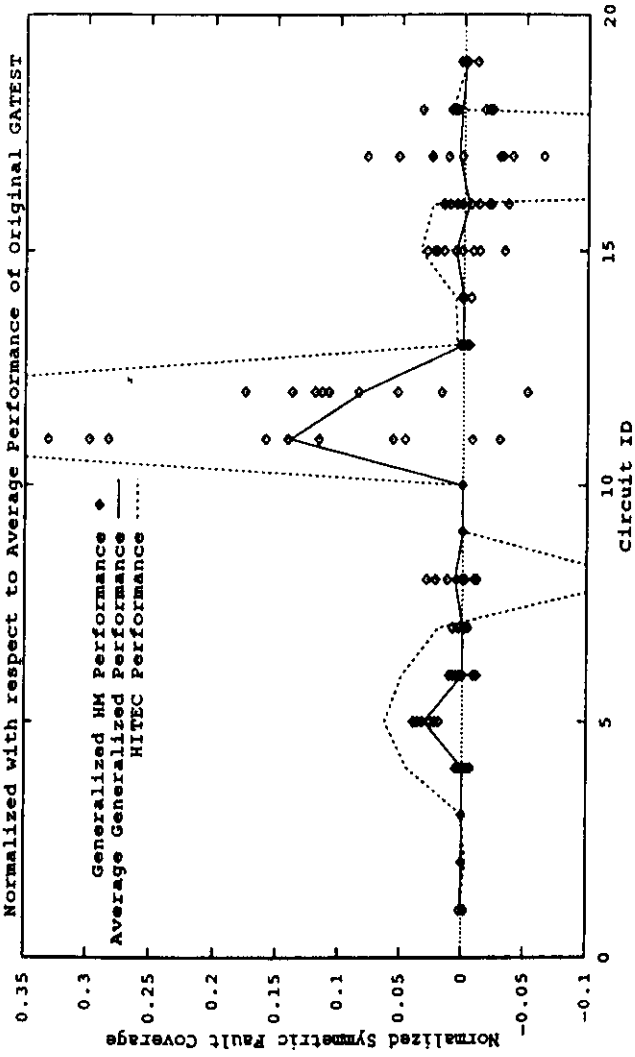

FIGURE 5.6
Distribution of normalized symmetric fault coverages of our generalized HM with respect to average fault coverages of the original GATEST on 19 benchmark circuits (s298, s344, s349, s382, s386, s400, s444, s526, s641, s713, s820, s832, s1196, s1238, s1488, s1494, s1423, s5378, and s35932 in that order).

Table 5.7 Parameters in TimberWolf (Version 6) Used in Our HM for Learning and Generalization

Parameter	Range	Step	Meaning	Default	Learned
P_1	0.1–2.5	0.1	Vertical path weight for estimating the cost function	1.0	0.1954624
P_2	0.33–2.0	0.1	Range limiter window change ratio	1.0	1.004637
P_3	18.0–28.0	1.0	High temperature finishing point	23.0	24.88345
P_4	71.0–91.0	1.0	Intermediate temperature finishing point	81.0	71.0
P_5	0.29–0.59	0.01	Critical ratio that determines acceptance probability	0.44	0.29
P_6	0.01–0.12	0.01	Temperature for controller turnoff	0.06	0.01

Table 5.8 Benchmark Circuits Used in Our Experiments

Cell name	Cells	Nets	Pins	Implicit feedthru
fract	124	163	454	0
s298	133	138	741	98
s420	211	233	1,488	112
primary1	766	1,172	5,534	0
struct	1,888	1,920	5,407	0
primary2	3,014	3,817	12,014	0
industrial1	2,271	2,597	—	—

From LayoutSynth92, *Int. Workshop on Layout Synthesis,* ftp site: mcnc.mcnc.org under /pub/benchmark, 1992.

tion 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 and in generalization.

In our experiments, we used seven benchmark circuits [10] whose specifications are shown in Table 5.8. Here, we have only studied the application of TimberWolf to standard-cell placement, though other kinds of placement (such as 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. Each learning experiment involved 1000 applications of TimberWolf. Based on the best 30 HMs (10 from each subdomain), we applied our generalization procedure to obtain one generalized HM.

The default and generalized HMs are shown in Table 5.7. Table 5.9 compares

Table 5.9 Comparison of Cost-Quality between Our Generalized HM and the Default HM on Seven Benchmark Circuits and Three *fast-n* Values over Ten Runs

Circuit	Performance measure	*fast-n* = 10		*fast-n* = 5		*fast-n* = 1	
		Default	Generalized	Default	Generalized	Default	Generalized
s298	Avg quality	655,706	0.019	0.008	0.040	0.020	0.045
	Max quality	640,668	0.044	0.038	0.052	0.032	0.058
	Avg cost	21.84	−0.062	0.454	0.412	3.525	3.282
s420	Avg quality	858,110	0.012	0.012	0.032	0.036	0.062
	Max quality	850,662	0.033	0.030	0.056	0.046	0.075
	Avg cost	30.69	−0.033	0.414	0.430	3.267	3.067
fract	Avg quality	87,524	0.061	0.054	0.049	0.099	0.168
	Max quality	77,248	0.166	0.162	0.152	0.183	0.229
	Avg cost	22.48	0.031	0.314	0.349	2.481	2.086
primary1	Avg quality	3,542,722	0.095	0.072	0.141	0.160	0.200
	Max quality	3,413,835	0.127	0.106	0.190	0.212	0.230
	Avg cost	227.10	−0.035	0.305	0.235	2.365	2.076
struct	Avg quality	2,029,258	0.405	0.225	0.622	0.658	1.000
	Max quality	1,894,847	0.528	0.293	0.755	0.781	1.119
	Avg cost	638.31	−0.414	0.538	0.280	2.497	2.179
primary2	Avg quality	17,944,576	0.100	0.150	0.225	0.273	0.315
	Max quality	17,111,600	0.173	0.205	0.282	0.315	0.337
	Avg cost	1,528.79	0.093	0.462	0.409	2.025	1.937
industry1	Avg quality	16,329,438	0.019	0.031	0.049	0.064	0.077
	Max quality	15,855,015	0.034	0.052	0.067	0.088	0.092
	Avg cost	1,321.97	0.109	0.200	0.165	1.987	1.702

Note: All performance values except the defaults with *fast-n*=10 are in the form of symmetric improvements with respect to the default performance values of *fast-n*=10. Percentage improvement in quality means percentage decrease in chip area.

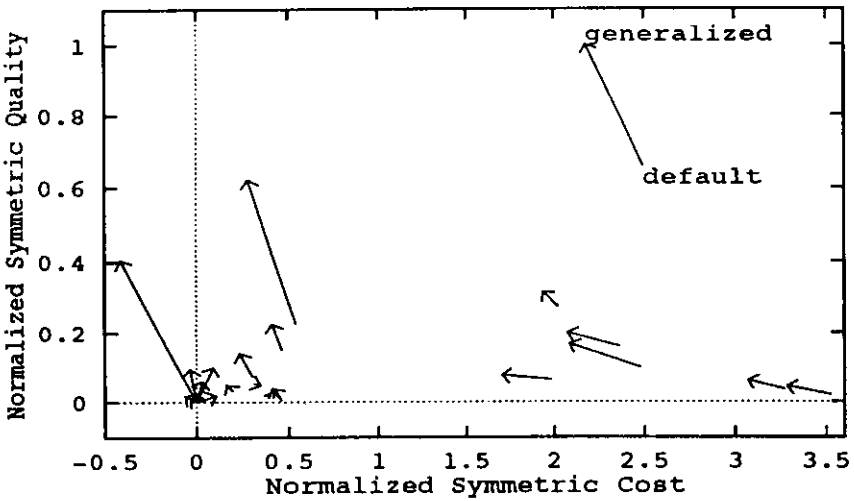

FIGURE 5.7
Comparison of average performance between the default and the generalized HMs.

the quality (average/maximum area of chip) and cost (average execution time) between the generalized HM and the default HM on all seven circuits with *fast-n* of 1, 5, and 10, respectively. (The cost for finding the minimum area is ten times the average cost.) These results are also plotted in Figure 5.7, where each arrow 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 one instance, and has worse cost in 5 out of 21 cases. Similarly, we see in Figure 5.7 that most of the arrows point in a left-upward direction, implying improved quality and reduced cost. Since we have only addressed a few parameters in TimberWolf, we expect to see more improvement as we learn other functions and parameters in TimberWolf.

5.4.3 Branch-and-Bound Search

A branch-and-bound 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 the 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

Table 5.10 Generation of Test Cases for Learning and Generalization of Decomposition HMs in a Branch-and-Bound Search (Each Has 12 Subdomains)

Application	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 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

search node into descendants 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 subsection, we apply learning to find only new *decomposition HMs*; preliminary results on learning of selection and pruning HMs can be found in Reference [2]. 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 two-dimensional plane (TSP), vertex-cover problems (VC), and knapsack problems (KS). The second problem can be solved by a polynomial-time approximation algorithm with guaranteed performance deviations from optimal solutions, and the last can be solved by a pseudo-polynomial-time approximation algorithm. Hence, we expect that improvements due to learning are likely for the first two problems and not likely for the last. Table 5.10 shows the parameters used in generating a *test case* in each application. We further assume that all subdomains belong to one problem subspace.

The problem solver here is a branch-and-bound algorithm, and a test case is considered solved when its optimal solution is found. Note that the decomposition HM studied is only a component of the branch-and-bound algorithm. We use well-known decomposition HMs developed for these applications as our baseline HMs (see Table 5.12). The normalized cost of a candidate decomposition HM is defined in terms of its *average symmetric speedup* (see Equation 5.2), which is related to the number of nodes expanded by a branch-and-bound 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 branch-and-bound search look for the optimal solution.

In our experiments, we selected six subdomains in each application for learning. We performed learning in each of these subdomains using 1600 tests, selected the top five HMs in each subdomain, fully verified them on all the learned subdomains, and selected one final HM to be used across all the subdomains. Table 5.11 summarizes the learning, generalization, and validation results. In the learning results, 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 specifically for a subdomain has around 1 to 35% improvement in its average symmetric speedups and 3000 to 16,000 times in learning costs.

Table 5.11 also shows the average symmetric speedups of the generalized HMs. We picked six subdomains randomly for learning. After learning and full verification of the five top HMs in each subdomain, we applied Equation 5.8 to identify one top HM to be used across all the 12 subdomains. Our results show that we have between 0 and 8% improvement in average symmetric speedups using the generalized HMs. Note that these results are worse than those obtained by learning, and that the baseline HM is the best HM for solving the knapsack problem.

The third part of Table 5.11 shows the average symmetric speedups when we validate the generalized HMs on larger test cases. These test cases generally require 10 to 50 times more nodes expanded than those used earlier. Surprisingly, our results show better improvement (9 to 23%). It is interesting to point out that 6 of the 12 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 5.12 shows the new decomposition HMs learned for the three applications. We list the variables that we fed 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 include 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.

Finally, Figure 5.8 plots the distribution of symmetric speedups of the generalized HM for VC with respect to the original HM using test cases in our generalization database. It shows the performance improvement of each individual test case with respect to that of the original HM. It further shows that performance is fairly evenly distributed above and below the average value without unnatural compression of ranges. This observation confirms that symmetric speedup is a proper normalization measure in this case. This plot also shows the difference in performance distribution across different subdomains, indicating the need to use a range-independent measure such as the probability of win.

Table 5.11 Results of Learning and Generalization for VC, TSP, and KS

Type	Application	Performance measure	Subdomain 1	2	3	4	5	6	7	8	9	10	11	12	Average
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	26,343.5	23,570.9	21,934.1	12,951.6	11,034.3	12,414.4	5,871.0	8,093.3	6,878.0	5,051.2	3,826.2	3,107.3	11,756.3
	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	2,846.6	1,543.9	2,077.7	2,207.7	2,314.9	1,865.6	1,889.9	1,847.5	2,509.7	1,947.0	1,445.4	1,958.8	2,037.9
	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	25,707.7	32,587.9	9,671.6	26,408.1	24,903.6	22,309.1	3,648.1	7,943.1	8,114.7	6,476.2	772.9	10,684.4	14,935.6
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

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

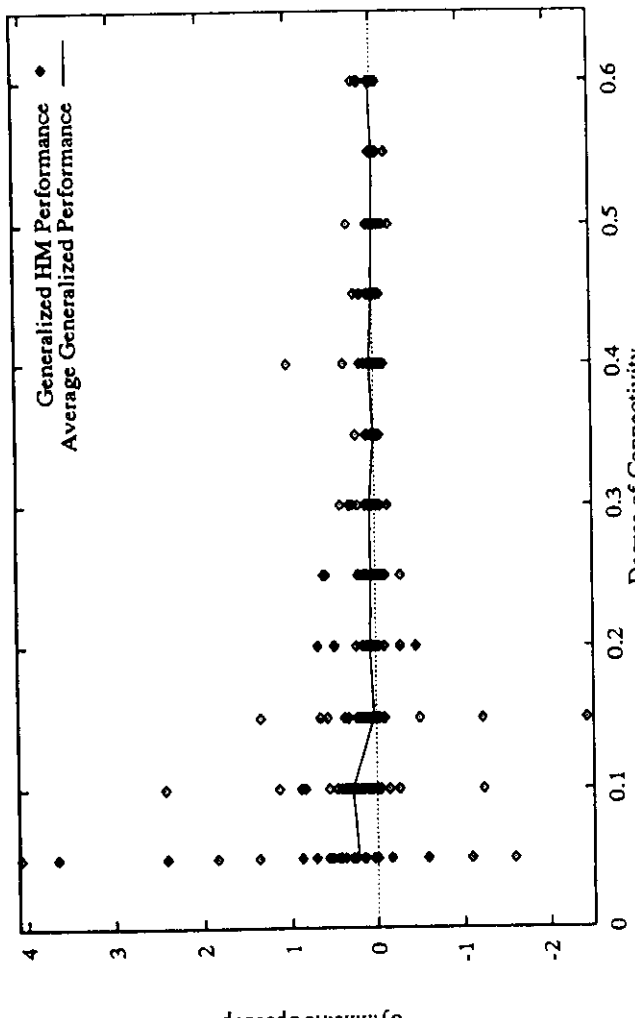

FIGURE 5.8
Distribution of performance values of the generalized decomposition HM for VC normalized on a case-by-case basis with respect to those of the original HM (using generalization test database).

Table 5.12 Original and Generalized Decomposition HMs Used in a Branch-and-Bound Search

Application	Variables used in constructing HMs	Original HM	Generalized HM
VC	l = live degree of vertex (uncovered edges) d = dead degree of vertex (covered edges) n = average live degree of all neighbors of vertex Δl = difference between l from parent node to current node	l	$1000\,l + n - \Delta l$ (l is the primary key, $n - \Delta l$ is the secondary key)
TSP	c = length of current partial tour m = min length to complete current partial tour a = avg length to complete current partial tour l = number of neighbor cities not yet visited d = number of neighbor already visited	c	$m * c$
KS	p, w = profit/weight of object s = weight slack = weight limit − current weight p_{max}, p_{min} = max/min profit of unselected objects w_{max}, w_{min} = max/min weight of unselected objects	p/w	p/w

In short, our results show that reasonable improvements can be obtained by generalization of learned HMs. 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 one subdomain but not in others.

5.5 Conclusions

In this chapter, we have studied automated generalization of performance-related heuristics for knowledge-lean applications. 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 methods to generalize learned heuristics to unlearned domains. To this end, we have proposed and evaluated a range-independent measure called probability of win for ranking heuristics in a problem subdomain. This allows heuristics across problem subdomains to be compared 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.

3. We have found better heuristics for generating patterns in circuit testing, placement and routing of VLSI components, and branch-and-bound search. Due to space limitation, new heuristics for process mapping [7] and blind equalization in communication are not shown.

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

1. One of the open problems that has not been studied 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 chapter. One such metric 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 trade between improved quality and higher cost.

3. Finally, we plan to carry out learning and generalization on more applications. The merits of our system, of course, lie in finding better heuristics for real-world applications, which may involve many contradicting objectives. Our generalization procedure needs to be extended in order to cope with applications with multiple objectives.

Acknowledgments

The authors are grateful to the following people for providing us with application programs used in testing our generalization procedure: Daniel Saab for his CRIS system [17], Elizabeth Rudnick and Janak Patel for their GATEST system [16], and Lon-Chan Chu for his WISE system for evaluating branch-and-bound search [21].

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] Brglez, F., Bryan, D., and Kozminski, K., Combinatorial profiles of sequential benchmark circuits, *Int. Symp. Circuits Syst.*, p. 1929, May 1989.

[2] Chu, L.-C., Algorithms for Combinatorial Optimization in Real Time and Their Automated Refinement by Genetic Programming, Ph.D. thesis, Department of Electrical and Computer Engineering, University of Illinois, Urbana, May 1994.

[3] Edmonds, J. and Karp, R.M., Theoretical improvements in algorithmic efficiency for network flow problems, *J. ACM*, 19(2), 248, 1972.

[4] Gembicki, F.W., Vector Optimization for Control with Performance and Parameter Sensitivity Indices, Ph.D. thesis, Case Western Reserve University, Cleveland, OH, 1974.

[5] Goldberg, D.E. and Holland, J.H., Genetic algorithms and machine learning, in *Machine Learning*, Vol. 3, No. 2/3, Kluwer Academic Publishing, Boston, MA, October 1988, 95.

[6] Ieumwananonthachai, A., Aizawa, A. N., Schwartz, S. R., Wah, B. W., and Yan, J.C., Intelligent mapping of communicating processes in distributed computing systems, in *Proc. Supercomputing 91*, ACM/IEEE, Albuquerque, NM, November 1991, 512.

[7] Ieumwananonthachai, A., Aizawa, A., Schwartz, S. R., Wah, B. W., and Yan, J.C., Intelligent process mapping through systematic improvement of heuristics, *J. Parallel Distributed Comput.*, 15, 118, 1992.

[8] Kirkpatrick, S., Gelatt, C. D., Jr., and Vecchi, M. P., Optimization by simulated annealing, *Science*, 220(4598), 671, 1983.

[9] Koza, J. R., *Genetic Programming*, MIT Press, Cambridge, MA, 1992.

[10] LayoutSynth92, *Workshop on Layout Synthesis*, ftp site: mcnc.mcnc.org under/pub/benchmark, 1992.

[11] Mehra, P. and Wah, B. W., *Load Balancing: An Automated Learning Approach*, World Scientific Publishing, Singapore, 1995.

[12] Newell, A., Shaw, J. C., and Simon, H. A., Programming the logic theory machine, *Prof. 1957 West. Joint Comput. Conf.*, IRE, 230, 1957.

[13] Niermann, T.M. and Patel, J. H., HITEC: a test generation package for sequential circuits, *Eur. Design Automation Conf.*, p.214, 1991.

[14] Pearl, J., *Heuristics–Intelligent Search Strategies for Computer Problem Solving*, Addison-Wesley, Reading, MA, 1984.

[15] Ramsey, C. L. and Grefenstette, J.J., Case-based initialization of genetic algorithms, in *Proc. 5th Int. Conf. on Genetic Algorithms*, pp. 84–91, International Society for Genetic Algorithms, Champaign, IL, June 1993.

[16] Rudnick, E.M., Patel, J.H., Greenstein, G.S., and Niermann, T.M., Sequential circuit test generation in a genetic algorithm framework, in *Proc. Design Automation Conf.*, ACM/IEEE, Los Alamitos, CA, June 1994.

[17] Saab, D.G., Saab, Y.G., and Abraham, J.A., CRIS: a test cultivation program for sequential VLSI circuits, in *Proc. Int. Conf. on Computer Aided Design*, IEEE, Santa Clara, CA, November 8–12, 1992, 216.

[18] Schwartz, S.R. and Wah, B.W., Automated parameter tuning in stereo vision under time constraints, in *Proc. Int. Conf. on Tools for Artificial Intelligence*, IEEE, Los Alamitos, CA, November 1992, 162.

[19] Sechen, C. and Sangiovanni-Vicentelli, A., The TimberWolf placement and routing package, *J. Solid State Circuits*, 20(2), 510, 1985.

[20] Sechen, C., *VLSI Placement and Global Routing Using Simulated Annealing*, Kluwer Academic Publishers, Boston, MA, 1988.

[21] Wah, B.W. and Chu, L.-C., Combinatorial search algorithms with meta-control: modeling and implementations, *Int. J. Artif. Intelligence Tools*, 1(3), 369, 1992.

[22] Wah, B.W., Population-based learning: a new method for learning from examples under resource constraints, *Trans. Knowledge Data Eng.*, 4(5), 454, 1992.

[23] Wah, B.W., Ieumwananonthachai, A., Chu, L.-C., and Aizawa, A., Genetics-based learning of new heuristics: rational scheduling of experiments and generalization, *IEEE Trans. Knowledge Data Eng.*, 7(5), 763–785, October 1995.