Chapter 4

Teacher: A Genetics-Based System for Learning and for Generalizing Heuristics

B. W. Wah and A. Ieumwananonthachai

In this chapter, we present the design of Teacher (an acronym for TEchniques for the Automated Creation of HEuRistics), a system for learning and for generalizing heuristics used in problem solving. Our system learns knowledge-lean heuristics whose performance is measured statistically. The objective of the design process is to find, under resource constraints, improved heuristic methods (HMs) as compared to existing ones. Teacher addresses five general issues in learning heuristics: (1) *decomposition* of a problem solver into smaller components and *integration* of HMs designed for each together; (2) *classification* of an application domain into subdomains so that performance can be evaluated statistically for each; (3) *generation* of new and improved HMs based on past performance information and heuristics generated; (4) *evaluation* of each HM's performance; and (5) *performance generalization* to find HMs that perform well across the entire application domain. Teacher employs a genetics-based machine learning approach and divides the design process into four phases. In the classification phase, the application domain is divided into subspaces (based on user requirements) and problem subdomains (based on the performance behavior of HMs). In the learning phase, HMs are generated and evaluated under resource constraints with a goal of discovering improved HMs. In the performance-verification phase, good HMs from the learning phase are further evaluated to acquire more accurate and more complete performance information. Finally, in the performance-generalization phase, HMs most likely to provide the best performance over the entire application domain are selected. We conclude the chapter by showing some experimental results on heuristics learned for two problems used in circuit testing.

4.1 Process of Designing Heuristics

A *heuristics-design process* is a process for *learning* improved heuristic methods (HMs) for a problem solver. These HMs, when used to solve an application problem, can provide better and/or less costly solutions.

The design of heuristics is an important issue in machine learning. Today, many application problems are too complex for us to find optimal algorithms analytically. Rather, we rely on heuristics designed by experts to find high quality solutions. The effectiveness of these heuristics, however, depends on the domain knowledge and past experience of these experts. When little domain knowledge is available, as in the applications we study in this chapter, it is important to develop automated learning methods for generating improved heuristics systematically, evaluating their performance on realistic test cases, and generalizing their performance to test cases not seen in learning.

The goal of this chapter is to present, for a given application domain, a system that can *find improved HMs over existing HMs with respect to some average objective measures.* The applications we are interested in have the following characteristics:

- a large number, and possibly infinitely many, test cases;
- a knowledge-lean application domain with little domain knowledge to relate the controls of HMs to their performance;

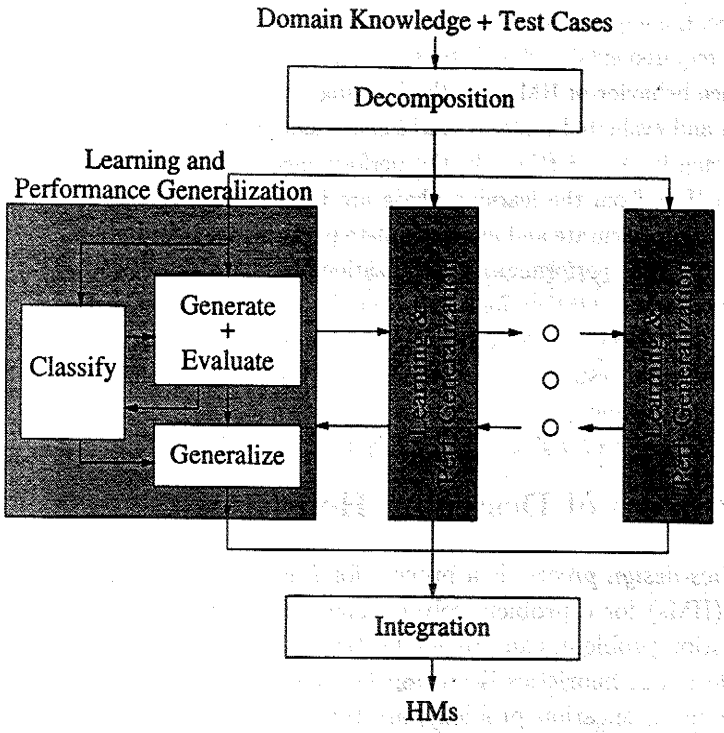


Figure 4.1: A general model of the heuristics-design process.

- performance-related HMs whose performance can be evaluated by the cost of applying them and the quality of the solutions found;
- HMs that are too costly to be tested extensively; and
- a large (and possibly infinite) pool of possible HMs.

There are five major issues in developing HMs for these applications as outlined in Figure 4.1.

(1) **Decomposition and integration of problem-solver components.** For a complex problem solver with multiple heuristic components, there are many combinations of heuristics that can be developed. Oftentimes, it is not possible to find improved heuristics for all the components simultaneously.

A feasible approach is to improve these heuristics in a piecewise fashion. This involves decomposing the problem solver into a number of groups with a smaller number of components in each, designing improved heuristics for each group, and integrating improved heuristics from each group into the problem solver.

There are several difficulties with this approach. First, it is hard to find the best way to group heuristics components in order to have the best overall performance. Second, heuristics in a problem solver may interact with each other, and the design of heuristics for one group may depend on the heuristics designed for others. Without the domain knowledge on the effects of heuristics in one group on others, it is hard to design heuristics for these groups independently.

One simple approach that minimizes the interactions is to design heuristics for different groups sequentially. In this approach, we first design an improved HM_A for group A. This is followed by the design of HM_B for Group B after HM_A is in place. The process is then repeated until HMs for all groups have been designed.

(2) **Classification of application domain.** When the performance of an HM is *nondeterministic* and the cost for evaluating it on a test case is expensive, we can only afford to evaluate each HM on a small number of test cases. In general, it is valid to estimate the true performance of an HM based on a subset of tests when the performance values are independent and identically distributed (IID).

We have found that HMs can behave differently on different subsets of test cases of an application [29]. When this happens, it is necessary to partition the application domain into smaller subsets (called subdomains) so that each can be evaluated independently.

In this chapter, we define a *problem subdomain* as a subset of the application domain so that performance values of HMs, when evaluated on test cases in the subdomain, satisfy the IID property.

Sometimes, it may be more efficient to have different groups of subdomains solved by different HMs. In this case, subdomains may need to be partitioned into groups and an HM be learned for each. We refer to this partitioning of the application domain as the *classification problem.*

(3) **Generation of heuristics.** This refers to the generation of improved HMs based on HMs generated already. This issue has been studied extensively in machine learning. Section 4.2 overviews the pre-

vious work in this area.

The generation of improved HMs is difficult for knowledge-lean problem solvers as there are no models that relate the specifications of an HM to its performance. In this case, the problem solver must be treated as a black box, and heuristics generation have to rely on weak (domain-independent) methods.

(4) **Evaluation of Heuristics.** To obtain improved HMs, we must be able to compare their performance. The performance of HMs is generally obtained by evaluating them on one or more test cases. When the performance of an HM is nondeterministic, this process may need to be repeated multiple times.

The key issue is to be able to compare the performance of different heuristics using a small number of tests. This is especially important in our research since we are dealing with heuristics that are expensive to evaluate.

(5) **Generalization of Performance on Heuristics Learned.** When the performance of an HM is nondeterministic and varies across different test cases, only a subset of test cases are usually used in evaluating its performance. However, we expect the HM selected to be generalizable; that is, it must perform well not only on test cases tested during learning but also on test cases not seen in learning. We call this the *performance-generalization problem* in this chapter.

Performance generalization is difficult when the application domain is large, and HMs have different (and possibly inconsistent) performance behavior across different regions (or subdomains) in the application domain.

The next section presents an overview of the work in automated learning of heuristics, which has been studied extensively in artificial intelligence. There has also been some work on the evaluation and generalization of heuristics in genetics-based learning [31, 33, 71]. The remaining issues on decomposition/integration and classification have been mostly ignored in the literature since they exist only in complicated application problems. Unfortunately, many real-world applications may fit in this category, and it is highly desirable to have some solutions to address these issues.

The approach presented in this chapter addresses all these issues except decomposition and integration. We currently require the decomposition/integration process to be performed manually by designers, who develop improved heuristics for each group sequentially. This is necessary because decomposition and integration cannot be studied until all the other issues have been addressed and a good system for designing improved heuristics for each component has been developed. We plan to address this issue in our future work.

We present in Section 4.4 the design of Teacher, a system that implements our heuristics-design process. Our system has four phases of operation, each of which is isolated to deal with a unique design issue. The first phase dealing with the classification issue is discussed in Section 4.5. Section 4.6 presents the second phase that addresses the generation of good heuristics for a subdomain and their evaluation. The operation in this phase is the one referred to by most researches as the learning process. Our approach in this phase is based on the genetics-based machine learning paradigm presented in Section 4.3. Section 4.7 discusses the third phase that deals with the verification of performance of heuristics. Section 4.8 presents the final phase that deals with the performance-generalization issue. In Section 4.9, we report our experience on learning HMs for two problems used in circuit testing.

4.2 Background on Heuristics Learning

In this section, we present a brief survey on the automated learning of heuristics. We first present a classification of the approaches in heuristics learning. This is followed by examples on the strengths and weaknesses of various learning methods. Finally, we summarize how our work is related to previous approaches and the type of application problems we have studied.

4.2.1 Classification of heuristics-learning methods

Learning is carried out for application domains that are either knowledge-rich or knowledge-lean. In a *knowledge-rich domain*, there is domain knowledge on a world model that relates the controls/inputs of an HM to its performance. Such a model can reduce learning time tremendously because new HMs can be derived with very little or no testing of previous HMs. In contrast, when the domain is *knowledge-lean*, there

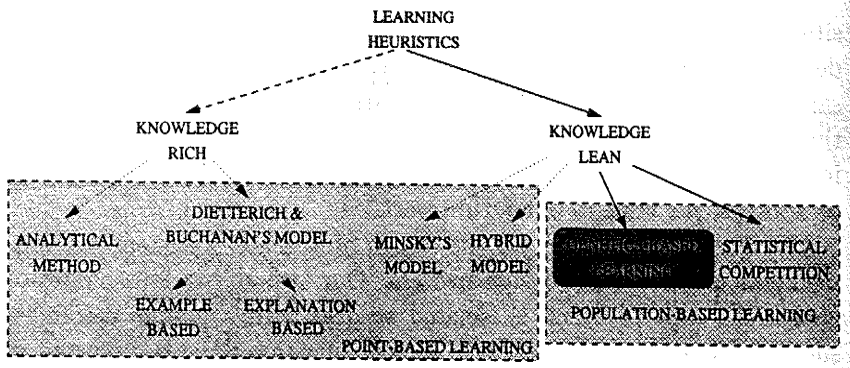


Figure 4.2: Classification of methods for learning heuristics.

is no world model, and heuristics derived must be tested to find their performance. These are the domains that we study in this chapter.

Figure 4.2 shows a classification of the methods for learning heuristics. We classify these methods based on the number of heuristics maintained by each during the design process—whether it is *point based* or *population based* [1, 69].

Point-Based Learning Paradigm. In this paradigm, a learning system maintains one incumbent HM that is modified in place by the learning system. Since each modification of the HM destroys the original HM, there must be high confidence that the new HM is better than the old one. This learning paradigm works well for learning knowledge-rich heuristics because the world model can be used to guide the generation of improved HMs [41, 45].

There are three models in traditional machine-learning studies that fit in this paradigm. Fundamental work in this area was addressed by Mitchell [46, 59], Minsky [42], and Dietterich and Buchanan [19]. The basic principle is based on a generate-and-test paradigm that generates plausible HMs, performs limited tests, and modifies the HMs according to the feedback signals obtained. Each of these models are described briefly in this section. Many existing machine-learning systems fit in one of these models.

A general point-based learning paradigm used by the three point-based models is shown in Figure 4.3 [69]. The general model includes the Learning Performance Database and Preprocessor, Credit Assign-

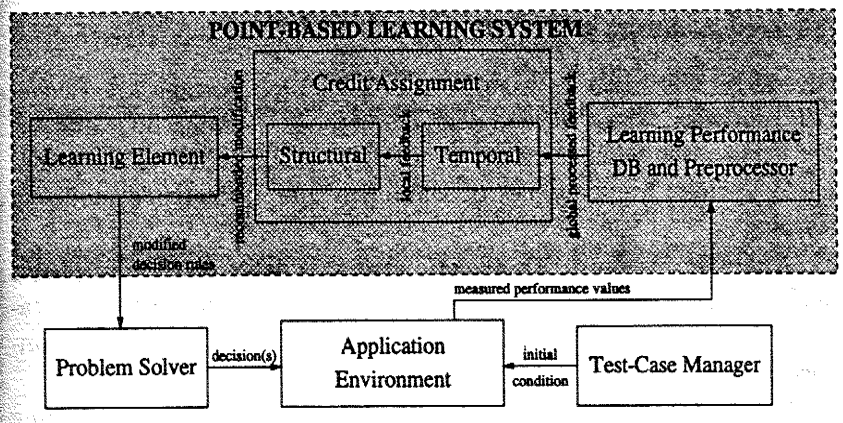


Figure 4.3: A general point-based learning model.

ment unit, and Learning Element. The problem solver and its initial conditions are shown as components outside the learning system.

The *Learning Performance Database and Preprocessor* captures the effects of decisions made by the problem solver on the application environment. The preprocessed data are used for *credit assignments* that include both *temporal* and *structural* credit assignments. The *Learning Element* then modifies the HM based on recommended actions from the Credit Assignment unit.

Population-Based Learning Paradigm. In contrast to the point-based approach, a population-based paradigm maintains multiple competing HMs and tries to find the best HM within the pool. During learning, new HMs are added to the pool and poor ones removed. This paradigm is useful for learning knowledge-lean heuristics, as the weak methods used in generating new HMs do not have to depend on a good world model [12, 26]. Moreover, it is not necessary for every new HM to perform well since a number of alternative HMs are generated.

One important characteristic of the population-based approach is the potential need for scheduling the available computational resources among competing HMs. Related issues on scheduling are presented in Sections 4.3 and 4.6.

4.2.2 Learning knowledge-rich heuristics

Several general methods have been proposed for learning knowledge-rich heuristics. These include analytic methods, learning by examples, and explanation-based learning. Learning by examples and explanation-based learning are examples of Dietterich and Buchanan's model of learning.

Analytic Method. An *analytic method* is based on comprehensive analysis on the problem solver with respect to its particular representation [18,20]. This approach is knowledge-intensive and very application-specific.

Dietterich and Buchanan's Model. This point-based learning model [8, 19, 69] and similar models proposed by Smith et al. [60] and Langley [37] belong to a class of models for learning HMs of target problems with well-defined objectives. They learn by supervised learning with *prescriptive feedback* that carries explicit information to guide the modification of the HM tested. This learning model fits within the framework of the general point-based model shown in Figure 4.3.

Learning by examples narrows the scope of possible heuristics by specialization and generalization [45]. *Explanation-based learning* exercises domain knowledge to explain the relationship between an HM and its performance [43, 44].

In learning knowledge-rich heuristics, extensive domain knowledge must be available. As our focus is on learning heuristics without such knowledge, methods developed for learning knowledge-rich heuristics cannot be used.

4.2.3 Learning knowledge-lean heuristics

Several general methods have been proposed for learning knowledge-lean heuristics. These include Minsky's learning model, hybrid point-based learning model, genetics-based learning, and statistical methods.

Minsky's Model. This model [42, 69] is an older and perhaps less restricted model of point-based learning systems. It applies well for learning HMs of target applications with undefined objectives in knowledge-lean environments, which tend to produce evaluative and possibly delayed feedback signals.

An *evaluative feedback* carries only implicit information about the desired behavior but explicit evaluation of the observed behavior. Such behavior is *a posteriori*, being measured or generated after the behavior has occurred. It requires a critic [73] that has some prior knowledge of the objective function and can assess the goodness of external states or sequences thereof. Scalar evaluative feedback signals are called *reinforcements* [42] and learning from such signals, *reinforcement learning*.

In order to simplify temporal credit assignment, the Markovian property has to be satisfied, namely, the current state of the system depends only on the last action performed by the HM and not on the sequence of actions before it. This property is generally hard to satisfy in complex applications. Examples of systems in this class include Klopf's drive reinforcement model [35] and Sutton and Barto's reinforcement model [64].

Hybrid Learning Model. This point-based learning model [69] combines aspects of Dietterich and Buchanan's model and Minsky's model. It uses an approximate temporal model instead of the Markovian model. It is intended for dealing with a knowledge-lean learning environment with an ill-defined objective, evaluative feedback, and a non-Markovian temporal scope.

Examples of this type of learning systems include EURISKO [38], Samuel's Checker Player [56, 57], Williams' REINFORCEMENT model [74], classifier system (the Michigan approach) [28], and the truck-backer-upper problem of Widrow et al. [48]. This type of learning does not address the lack of domain knowledge for structural credit assignment.

Genetics-Based Machine Learning. This is a population-based approach based on generate-and-test and the application of genetic algorithms [22, 23] to machine-learning problems. In generate-and-test methods, new heuristics to be tested are generated by applying operators to existing heuristics that perform well [12, 26]. The new heuristics are potentially better as they were generated from heuristics that perform well. The reproduction operators applied include crossover and mutation. More details about this approach are presented in Section 4.3.

Examples of genetics-based machine learning include genetic programming [36] and the Pittsburgh approach to classifier systems [26].

Statistical Competition. One form of statistical competition uses

statistics to translate data into concepts so that concept learning can be applied [52]. Another form uses statistics to decide which heuristics to test more, given a pool of candidate heuristics [14,30]. This method is especially useful for learning heuristics whose exact performance cannot be determined by a limited number of tests.

This approach is limited in its usefulness as it only tests a a fixed pool of heuristics and excludes the introduction of new heuristics based on past evaluations.

4.2.4 Summary

In learning knowledge-lean heuristics, genetics-based machine learning is the most suitable approach. The two point-based learning models for learning knowledge-lean heuristics are too restricted and cannot be applied for the type of applications we address in our research. Minsky's model requires a Markovian temporal model that is hard to satisfy in general, and the hybrid model requires some domain knowledge for structural credit assignment. The statistical competition approach can only handle a fixed pool of heuristics and does not allow incremental improvements through mutations and crossovers.

We have adopted a genetics-based learning approach that can operate in a knowledge-lean environment and can generate new and potentially improved heuristics based on past performance information. Since tests are expensive, we have incorporated some aspects of statistical competition to improve resource scheduling in our learning framework. In the next section, we examine the genetics-based approach in more details and identify some key issues to be studied. Section 4.6 presents the architecture of a population-based learning system.

4.3 Background on Evolutionary Computing

Genetics-based machine learning, the approach we have selected for our heuristics-design process, is a part of a bigger field called evolutionary computing [27,62].

Evolutionary computing (EC) include *genetic algorithms (GAs), evolutionary programming (EP), evolution strategies (ES), classifier systems (CFSs), genetic programming (GP)*, and several other problem-

solving strategies. They are based on the means of natural selection, the survival of the fittest and the theories of evolution [27]. They provide the evolution of individual structures through the processes of selection, mutation, and reproduction. These processes depend on the perceived performance of the individual structures as defined by an environment [62].

In this section, we first present a brief overview of genetic algorithm (GA) before proceeding to genetics-based machine learning, an extension of genetic algorithms to machine-learning problems.

4.3.1 Genetic algorithms

Genetic algorithms (GAs) are adaptive methods that may be used to solve search and optimization problems. Since the development of GA by Holland [28], GAs have been extensively studied [15,22,25]. They are based on the genetic processes of biological organisms described first by Charles Darwin in *The Origin of Species*. Populations of competing individuals evolve over many generations according to the principle of natural selection and "survival of the fittest."

Genetic algorithms work with a *population* of "individuals" and a set of biologically based operators (such as mutation and crossover). Each individual represents a possible solution to a given problem. Based on the theory of evolution, only the most suited elements in a population are likely to survive and generate offsprings [22].

In GAs, each individual is represented as a string of binary values (0 or 1). This uniform representation is application-independent, allowing various studies to focus on the general approach.

A *fitness level* is assigned to each individual based on how good the problem has been solved. Those with high fitness levels will reproduce by recombining with other individuals in the population. New offsprings generated share some features taken from each parent. Those with low fitness levels are less likely to be selected for reproduction and so die out.

A new generation is, therefore, produced by selecting the best individuals from the current *generation* and mating them to produce a new set of offsprings. Over many generations, good characteristics are spread throughout the population. This process allows the most promising ar-

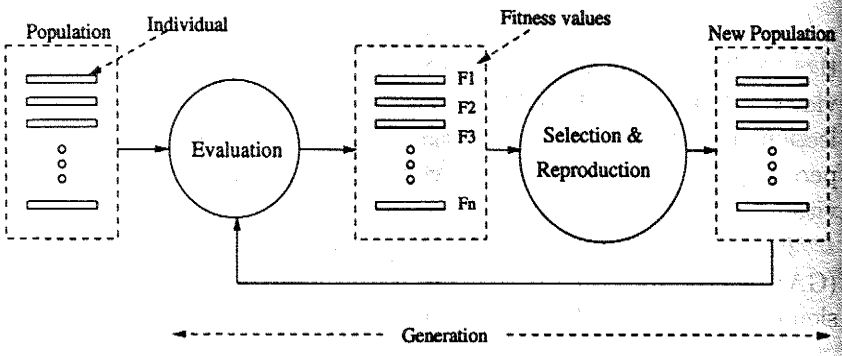


Figure 4.4: General model of the evolution process in a genetics-based approach.

eas of the search space represented by the population to be explored [9].

Figure 4.4 shows the overall process within GAs. The process can be viewed as iterating over two different steps: (1) evaluation of individuals of a population in the current generation, and assigning fitness levels to each individual, and (2) generation of a new population for a new generation by selecting existing individuals based on their fitness values and by using selected individuals to reproduce.

In most traditional GAs, the fitness of each individual is exact and can be computed with negligible costs. However, there are cases in which there are "noises" in the evaluation process, resulting in multiple evaluations and in uncertainties over the fitness of each individual [4, 21, 24]. This condition can significantly increase the amount of fitness evaluations performed during each generation. Such is the case in the applications we have studied. Here, the performance of an HM on a test case is governed by a statistical distribution, and testing the HM once is equivalent to drawing a random value from the distribution.

There are many issues in genetic algorithms that have been and continue to be studied, including issues on representation, population size, fitness evaluation, selection of individuals for reproduction, and reproduction methods [9, 10, 16, 62].

4.3.2 Genetics-based machine learning

Genetics-based machine learning is an extension of genetic algorithms (GAs) to solve machine-learning problems [22]. The term is used in this

chapter to cover applications of the idea behind genetic algorithms in order to develop HMs used in a problem-solving process.

Genetics-based machine learning is based on the same idea of evolution and natural selection as in genetic algorithms (see Figure 4.4). Such a system maintains a population of individuals, evaluates their fitness values, and generates a new population by selecting existing individuals for reproduction based on their fitness values. However, the representation of HMs is generally more complex [22, 36]. Some example representations of an individual include an if-then rule [22], a set of rules [34], a Lisp expression [36], and a vector of numbers [70].

Genetics-based learning applied to learn improved HMs are more complex computationally as compared to genetic algorithms. First, because the structure of each individual can be complex, reproduction operators, such as mutation and crossover, can also be more complex. In addition, more domain knowledge may be needed in order to create knowledge-intensive reproduction [10, 22], such as those used in GIL [34]. Second, since the evaluation of an HM on a test case is non-deterministic (or noisy), evolutionary computing applied to learn improved HMs must also deal with "noisy" conditions. This means that the fitness of each individual may not be exact, and that multiple evaluations of each individual may be necessary.

Existing work in genetics-based machine learning can be divided into two approaches: (a) treating the entire population as the HM to be learned and (b) treating each individual as a complete HM.

(A) *Population as HM.* This approach, known as the Michigan approach in the genetic algorithm community [15, 75], treats each individual in the current population as a component contributing to the entire population. It requires all components of a solution to be homogeneous across the population. To achieve better performance for the entire population, there must be cooperation among individuals. This is similar to the hybrid point-based learning paradigm discussed earlier. It requires credit assignments to apportion credits or debits to various contributing individuals in an episode. The most common credit assignment strategy is the *bucket brigade* algorithm [22].

Examples of this approach include most classifier systems such as CS-1 [22] and CFS-C [53]. This approach can be applied in online learning to improve a problem solver during the problem-solving process.

(B) *Individual as HM.* This approach, known as the Pittsburgh approach in the genetic algorithm community [15,75], treats each individual as a solution that competes with other individuals in the population. In this case, each individual can be entirely different from one another, leading to more complex systems as compared to the Michigan approach.

This approach does not require credit assignments to apportion credits or debits, since each performance feedback is directed to only one individual. However, feedbacks usually come less often to each individual, leading to more evaluations in order to reach a final result.

Examples of this type of approach include the Pittsburgh approach [15,75] to classifier system (such as LS-1 [61], GABIL [17], and GIL [34]) and genetic programming (GP) [36].

This approach is more suitable for our system because HMs in our applications are usually nonuniform with different structures and do not have good models of interactions among their components to allow credit assignments.

There are some systems that use a hybrid of the Michigan and the Pittsburgh approaches. In this case, each individual is treated as a potential solution with components that can contribute to the problem-solving process. It is also necessary to use credit assignment within each individual to assign credits/debits to each component. An example of this hybrid approach is SAMUEL [26].

In summary, genetics-based learning can be used to learn knowledge-lean HMs in the applications that we study in this chapter. A population of HMs can be maintained, each will be tested, evaluated, mutated, and crossed over with other HMs to form new HMs in the next generation. Existing genetics-based methods, however, do not address the issues when performance data of HMs do not belong to one common statistical distribution and when tests are expensive to conduct. When HMs behave differently across different subsets of test cases of an application, it is possible for an HM to perform well in one subset but poorly in another. Hence, the generalization of test results to test cases not evaluated must be considered. The problem is further complicated when tests are expensive to carry out. In this case, the learning system must decide how many HMs to be tested and the amount of testing to be carried out on each. These issues are addressed in Teacher described in the next section.

4.4 The Teacher System

In this section, we discuss Teacher, a genetics-based learning system we have developed in the last six years [69]. Its objective is to learn, under limited computational resources, good HMs for solving application problems and to generalize the HMs learned to unlearned subdomains. We choose to 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. Our learning system is capable of proposing more than one HMs, showing trade-offs among these objectives.

4.4.1 Key characteristics of Teacher

Our present prototype is unique as compared to other genetics-based learning studies because it combines the following three features.

- Our learning environment is noisy so that the performance of each HM may have to be evaluated using multiple tests.
- We consider applications in which HMs behave differently in different subdomains (Section 4.1). 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. In the applications presented in Section 4.9, a fast workstation takes a few days to perform one to two thousand tests. Hence, it is not feasible to perform millions of tests as assumed in other genetics-based learning systems [36]. For simplicity, we consider logical time in this chapter in which one unit of time is needed for each test of an HM.

These conditions are more realistic for complicated real-world applications for which we want to design improved heuristics.

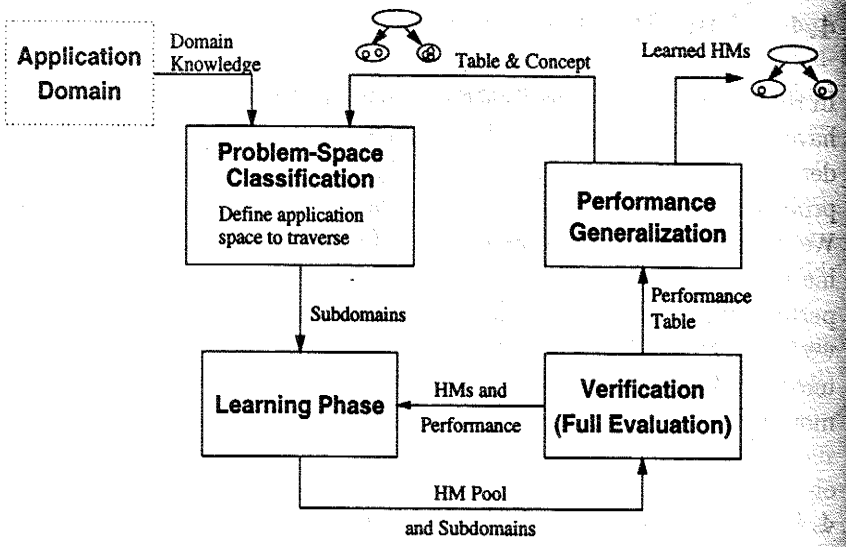


Figure 4.5: Organization of the heuristics-design process in Teacher.

4.4.2 Four phases of Teacher

The operations of Teacher are divided into four phases: classification, learning, performance verification, and performance generalization. Each phase is designed to independently deal with a separate issue discussed in Section 4.1.

Currently, the classification phase is performed manually while the other three phases are automated [32]. We have plans to incorporate strategies for automated classification, decomposition and integration of HMs in Teacher in the future. Figure 4.5 shows the overall design process in Teacher. We describe the objectives and the key issues of each phase in this section and our solutions to these issues in the following sections.

Classification phase

This first phase of the design process partitions test cases in an application into distinct subsets. Its goal is to partition a target application domain into smaller subsets in order to (a) identify different regions within the application domain that may require different HMs, and (b)

make sure that performance values from test cases used in learning are representative and can be used for statistical estimation of the true performance of unseen test cases.

There are two steps in this phase.

(a) *Subspace classification.* Within an application domain, different regions may have different characteristics, each of which can best be solved by a unique HM [51]. Hence, regions should be identified whenever possible so that unique HMs can be developed. This involves partitioning the application domain into a small number of distinct subspaces so that improved HMs are learned for each.

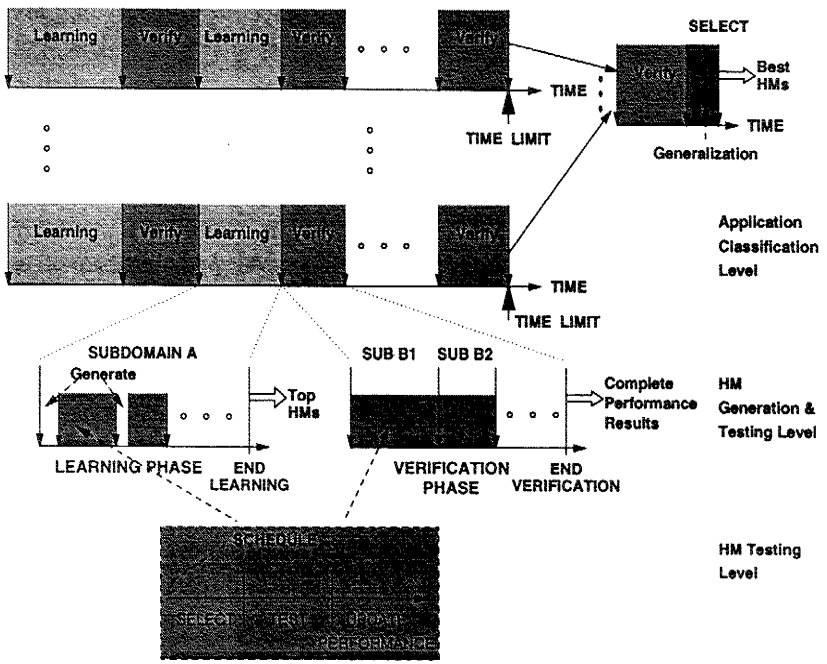
We define an *application subspace* as a user-defined partition of an application domain so that HMs for one subspace can be learned independently of HMs in other subspaces. Subspace partitioning is important when test cases in an application have vastly different behavior.

(b) *Subdomain classification.* A *problem subdomain* in this chapter is defined as a subset of the application domain (or application subspace) so that the performance of HMs in each subdomain can be estimated statistically based on a subset of test cases in this subdomain. In other words, the performance values of an HM in a subdomain are *independent and identically distributed* (IID), but may not be IID across subdomains. Since the performance distribution of an HM may be different across different subdomains, the performance of HMs cannot be compared or combined across subdomains in a learning experiment.

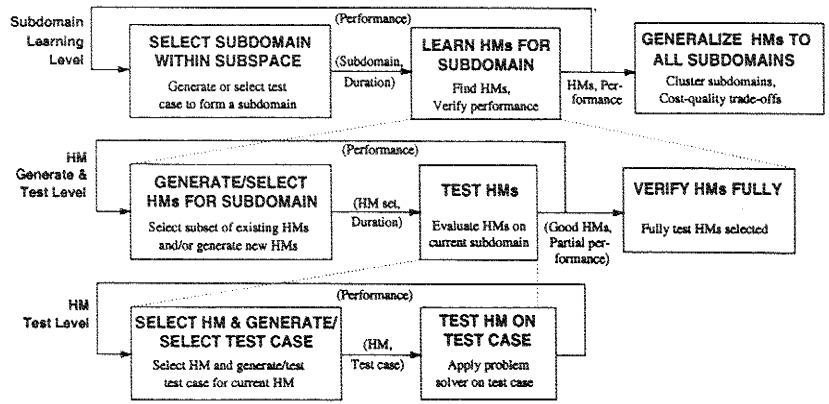
Learning phase

The goal in this phase is to find effective HMs for each of a limited set of subdomains. Besides being the most complicated phase in the design process, it is the only phase in which new HMs are introduced and tested under resource constraints. Figure 4.6 shows the tasks in the learning, performance-verification, and performance-generalization phases.

To perform learning, the system first selects a subdomain, generates good HMs (or uses existing HMs from users or previous learning experiments) 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 found during learning



(a) The learning, performance-verification and performance-generalization process.



(b) The actions in each phase of Teacher.

Figure 4.6: The process and actions in each phase in Teacher.

may not be tested adequately. Note that learning is performed on one subdomain at a time.

There are three key issues in this phase.

(a) *Heuristics generation.* This entails the generation of good HMs, given the performance information of "empirically good" HMs. As discussed in Section 4.2, we use weak genetics-based operators here [12,36].

(b) *Performance evaluation.* This is related to the performance evaluation of HMs during learning, given that there may be multiple performance measures, that there is no defined relationship among them, and that HMs may have different performance across different subdomains.

(c) *Resource scheduling.* Given the performance information of HMs under consideration, resource scheduling entails the selection of HMs for further testing, the termination of the current generation, and the initiation of the next generation. 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 (i) the number of tests on each HM, (ii) the number of competing HMs to be maintained at any time, and (iii) the number of problem subdomains to be used for learning and for generalization. We have studied two related issues in resource scheduling in genetics-based learning algorithms: *sample allocation* and *duration scheduling* [2,4,6,30,70].

Performance-verification phase

The goal of this phase is to fully evaluate the set of HMs with good performance found at the end of previous learning phases. As mentioned previously, the performance of HMs evaluated during learning is only estimated based on incomplete and possibly inadequate performance data. In order to select HMs that can be generalized to unlearned subdomains, we need to carry out full evaluation of each HM selected at the end of a learning phase. This involves evaluating each selected HM fully on all subdomains from all learning phases and any additional subdomains provided by users. The main potential issue in this phase is the scheduling of limited computational resources to test a given set of HMs.

Performance-generalization phase

The goal of the last phase is determine which HMs found in previous learning phases perform consistently well on all subdomains, including those not studied in learning. This notion of performance generalization is slightly different from that in point-based learning, as we generalize the performance of HMs found to unseen test cases rather than generalizing the HMs found to better HMs. There are two key issues to be studied here.

(a) *Performance of HMs across different subdomains.* As discussed in Section 4.1, HMs may have different performance behavior in different subdomains; hence, performance values of an HM from different subdomains cannot be combined directly. Oftentimes, an HM may perform well in some subdomains but worse in others. One approach we have studied is to find HMs that are consistently better than others with a high probability across all the subdomains [33, 71].

(b) *Cost-quality trade-offs.* This involves determining efficient HMs that perform well in an application. Should there be multiple HMs to be applied (at a higher total cost and better quality of results), or should there be one HM that is costly to run but generates high-quality results? Some results on these trade-offs are shown in Section 4.9.

4.5 Strategies in the Classification Phase

Recall from the last section that quantitative comparison of performance is difficult when test cases are of different behavior. Hence, before learning begins, the application domain should be broken into smaller subspaces and subdomains.

In subspace partitioning, the attributes needed for partitioning may not be defined, or the number of attributes may be too large. When this happens, non-parametric 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.

We show two examples to demonstrate the idea of application subspaces.

Example 1. Consider solving a vertex-cover problem that finds the minimum number of nodes to cover all the edges in a graph. 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 those for sparsely connected ones. Consequently, the application domain of all graphs can be partitioned (in an ad hoc fashion) into a small number of subspaces based on graph connectivities. □

Example 2. As another example, in generating test patterns for testing VLSI circuits, previous experience shows that sequential circuits require tests that are different from those of combinatorial circuits. Hence, we can partition the application domain into two subspaces, one for sequential circuits and another for combinatorial circuits. However, we are not able to partition the subspace of sequential circuits into smaller subspaces as it is not clear which attributes (like the length of the longest path, the number of flip flops, etc.) should be used in this partitioning. □

In our current implementation, subspace partitioning is 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. When such attributes cannot be identified, we simply assume that the entire application domain is in one subspace.

In classifying test cases in a subspace into subdomains, some domain knowledge, such as previous experience on similar problems, may be required. After subdomains have been classified, it is important to test the HMs in each subdomain to make sure that their performance data are IID. Examples of methods for testing for identical distributions are Kolmogorov-Smirnov two-sample test [29, 47, 50], Mann-Whitney test [68] and Wald-Wolfowitz two-sample runs test [47]. On the other hand, testing for independence is difficult, if not impossible [68]. Currently, there do not exist methods to guarantee that all performance values are independent. However, it is possible to evaluate the randomness of *a given sequence of test cases*, which is a necessary condition for data to be independent. For instance, test of randomness can be found by computing the total number of runs up and down [47, 72], the total number of runs above and below the median [68, 72], and the total num-

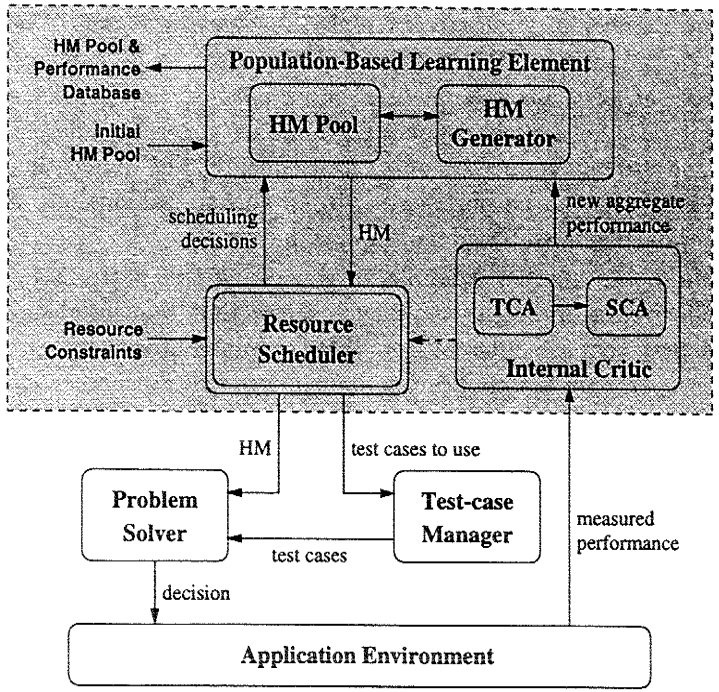


Figure 4.7: Architecture of population-based learning for one subdomain.

ber of runs above and below the mean [47]. Other randomness tests include Kendall's Rank Correlation Coefficient Test [72] and Circular Serial Correlation Coefficient Test [72]. Results on applying these tests are discussed in the references [29].

Continuing with Example 1 on the vertex-cover problem, a problem subdomain can be defined as a collection of random graphs with a certain degree of connectivity. As another example, in generating test patterns for testing VLSI circuits (Example 2), each circuit may be treated as an individual subdomain, as we do not know the best set of attributes to group different circuits into subdomains.

4.6 Learning Heuristics in One Subdomain

In this section, we present our approach to learn improved HMs under resource constraints for a single subdomain. Figure 4.7 shows the architecture of our resource-constrained learning system for one subdomain [69]. This population-based learning system is based on the genetics-based machine-learning paradigm. There are five main components in the system:

(a) *Resource Scheduler* that determines the best way to use the available computational resources,

(b) *Internal Critic* that provides feedback, based on measured performance, to indicate how well a particular HM has performed,

(c) *Population-Based Learning Element* that generates new HMs and maintains a pool of existing ones and their past performance,

(d) *Test-Case Manager* that generates and maintains a database of test cases used in HM evaluation, and

(e) *Problem Solver* that evaluates an HM using a test case.

In this research, we assume that the application-specific Problem Solver and Test-Case Manager are user-supplied. The remaining three components are designed to deal with the three key issues in Section 4.4: heuristics generation, performance evaluation, and resource scheduling.

Problem Solver

This component is simply the target problem solver whose heuristics we want to improve. The performance of applying a problem solver on a test case is in terms of the quality of the solution found and the cost of the problem-solving process.

In our learning strategy, the problem solver accepts (a) the specification of the HM to be used in problem solving, and (b) the test case to be solved. It also has a mechanism to return the measured performance of the problem-solving process as feedback to the learning system.

Test-Case Manager

This provides test cases to be used in learning. These test cases are either generated randomly or retrieved from a database.

In our current implementation, each HM is evaluated on a predefined sequence of user-specified test cases. When a test case is requested for testing a particular HM, the Test-Case Manager returns the first test case in the sequence that has not been evaluated by the chosen HM. This strategy allows performance data of two HMs to be normalized against each other and is useful when performance data have large variances.

Population-Based Learning Element

The *Population-Based Learning Element* maintains a pool of active HMs. At the end of each generation, a new set of HMs are generated to replace existing HMs. Several top active HMs are usually retained along with the new HMs while other HMs are removed from the active pool.

The Population-Based Learning Element in Teacher generates new HMs using weak domain-independent operators, such as crossover, mutation, and hill-climbing. These are traditional operators used in genetic algorithms for generating new HMs [12, 25]. The process for selecting existing HMs for reproduction is also the same as in traditional genetics-based machine learning.

More advanced generation methods that require additional domain knowledge are left for future study. They are currently not necessary because our application domains are knowledge lean.

Internal Critic

In general, this performs credit assignment [63] that apportions credit and blame on components of an HM using results obtained in testing (see Figure 4.3). 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. TCA divides feedback between the current and the past decisions. Methods for TCA depend on whether the state space is Markovian: non-Markovian representations often require more complex TCA procedures. The second stage is SCA that translates the (temporally local but structurally global) feedback associated with a decision point into modifications associated with various parameters of the decision process.

Since the knowledge-lean applications considered here does not have a world model that relates states, decisions, and feedback signals generated by the learning system or measured in the environment, credit assignment has a much weaker influence on performance improvement. Note that the lack of a world model for credit assignment is the main reason for maintaining competing HMs in our learning system.

In our current prototype, the Internal Critic normalizes the performance value of an HM on a test case against the performance value of the same test case evaluated by the baseline HM. It then updates the performance metrics of the candidate HM. This step is similar to updating fitness values in classifier-system learning.

We have chosen to use a fixed baseline HM during each learning phase and compare different HMs based on their estimated average normalized performance. This baseline HM is usually the best existing HM before learning begins.

Our approach in normalization may cause performance anomalies. For instance, different ways of normalization may lead to different ordering of HMs by their performance data. Anomalies in ordering may also happen when baselines are changed. Strategies to address some of these anomalies have been presented elsewhere [29, 69, 70].

Although anomalies may happen, it is not critical to have perfect ordering of HMs during learning, as the HMs will eventually be evaluated fully in the Performance Verification Phase.

Resource Scheduler

This schedules tests of HMs based on the available computational resources. It is critical when tests are computationally expensive. There are two problems in scheduling during each learning phase.

The *sample-allocation problem* involves the scheduling of tests of HMs in a generation, given a fixed number of tests in the generation and HMs to be tested. This problem is known in statistics as the (sequential) *allocation problem* [11, 67] and the scheduler, the *local scheduler*.

The *duration-scheduling problem* involves deciding when to termi-

nate an existing generation and to start a new one. The part of the resource scheduler that deals with this problem is known as the *global scheduler*.

These two problems, as well as the scheduling of tests under multiple performance objectives, are presented elsewhere [3,5,7,30,69,70]

4.7 Strategies in Performance Verification

In this phase, we like to find more complete performance information about the HMs we have generated during the learning phase(s). This is necessary for two reasons. First, the performance information obtained during each learning phase pertains to only one subdomain and is usually incomplete due to resource constraints. Second, the performance-generalization phase (to be described in the next section) requires performance information of each HM on every subdomain. Hence, we need to evaluate thoroughly the HMs selected at the end of learning.

The operations in this phase (see Figure 4.8) are very similar to those in the learning phase except for the following differences. (a) A fixed pool of HMs is maintained in this phase and no new HMs are generated. (b) More than one subdomains of test cases can be maintained by the Test-Case Manager. (c) The performance of HMs from different subdomains is evaluated separately and independently by the Internal Critic. (d) Resource scheduling in this phase has a different goal of minimizing uncertainties in the performance of all HMs across all subdomains. (e) Only temporal credit assignment is done in the Internal Critic since we do not use structural credit assignment to modify the HMs tested.

Currently, our prototype does not address the issue on resource scheduling in this phase. It evaluates each HM fully on all test cases in each problem subdomain. Such a strategy may be inefficient because it tests good HMs as well as poor HMs to the same extent. We plan to study resource scheduling in the future.

4.8 Strategies in Performance Generalization

As discussed in Section 4.4.2, an application domain (or application subspace) can have many subdomains, and HMs may behave differently

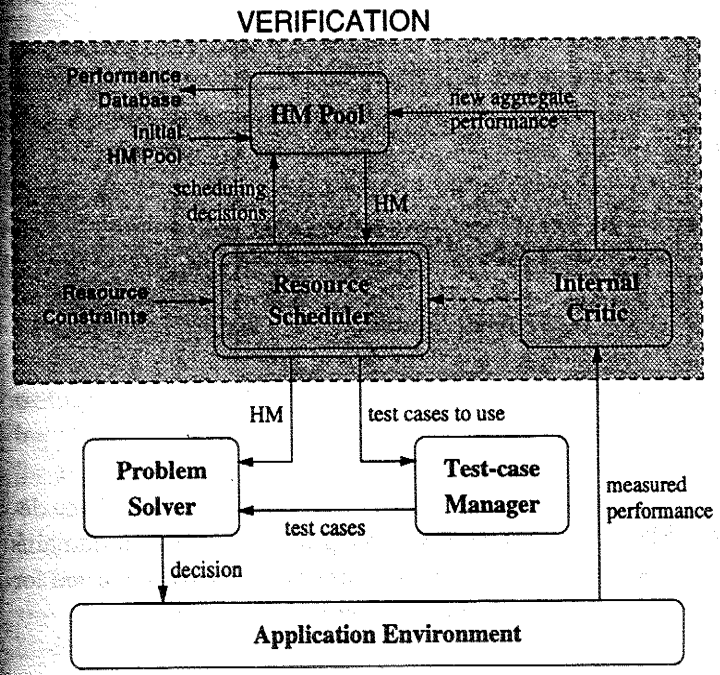


Figure 4.8: General model for performance verification of HMs learned.

in different subdomains. As a result, it is possible for an HM to perform well across all the subdomains, but its performance data from different subdomains cannot be aggregated statistically. Our objective in generalization is, therefore, to find one or more HMs in a given application domain that has a high probability of performing better than other competing HMs on a randomly chosen test case in the problem domain.

The performance-generalization process is difficult because (a) performance data from different subdomains must be treated separately and independently, and (b) there are usually many more subdomains than the ones we can test. Since we may not be able to characterize the subdomains we test to be representatives of the entire application domain, the process is heuristic in nature.

Example 3. To illustrate the difficulties in performance general-

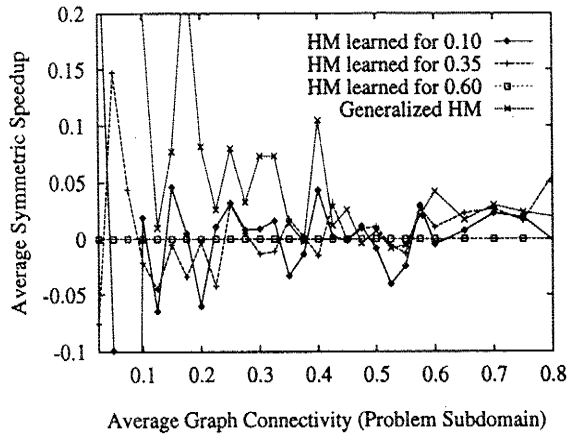


Figure 4.9: Average symmetric speedups (over 15 test cases) of three decomposition HMs to solve the vertex-cover problem, where subdomains are manually selected based on graph connectivity. The HM learned for 0.6 connectivity is the same as the baseline HM.

ization, we show in Figure 4.9 the average symmetric speedups[1] of four decomposition HMs used in a branch-and-bound search to solve vertex-cover problems. We treat all test cases as belonging to one subspace, and graphs with the same degree of connectivity are grouped into a subdomain. We apply genetics-based learning to find the five best HMs for each of three subdomains with connectivities 0.1, 0.35, and 0.6.

Figure 4.9 shows the performance of the best HMs learned in each subdomain across all the subdomains. We have also identified a single

[1]The symmetric-improvement measure, S^{sym+}, is defined as follows [29, 69, 70]:

$$S^{sym+} = \begin{cases} S^+ - 1 & \text{if} \quad S^+ \geq 1 \\ 1 - \dfrac{1}{S^+} & \text{if} \quad 0 \leq S^+ < 1 \end{cases},$$

where S^+ is the improvement ratio of the new HM with respect to the original baseline HM. Symmetric improvement ratios have the properties that they range from negative infinity to positive infinity, and that the baseline has a value of zero. On the other hand, improvement ratios range from zero to positive infinity, and the baseline HM has a value of one.

generalized HM among the fifteen HMs learned and show its performance in Figure 4.9. We find 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. We have also found that performance generalization is possible in terms of average performance. We must point out that the average performance should not be used as the sole indicator, as performance variances may differ from one subdomain to another. □

Our current work on generalization is to ensure that HMs generated in learning perform well across multiple subdomains. This is done by testing each HM on multiple subdomains in a generation and by selecting those that perform well across all the subdomains to be candidates for reproduction in the next generation. Our preliminary results indicate the need of a better fitness function to measure the merits of HMs that do not perform well across all the subdomains. Our current fitness function, based on the minimum average performance across all the tested subdomains, is too weak and may prune all the candidate HMs in a generation.

4.9 Experimental Results

Teacher has been applied in learning improved process-placement strategies on a network of workstations [40], more efficient process-placement strategies on distributed-memory multicomputers [30], more robust parameters in a stereo-vision algorithm [58], smaller feed-forward neural networks [65,66], improved heuristics for a branch-and-bound search [39, 76], better and less costly strategies in circuit testing [33,70,71] and cell placement and routing [33,70,71], and improved parameters in channel equalization [31,33].

In this section, we report our experience in applying Teacher to learn improved HMs in two problem solvers, CRIS [55] and GATEST [54], for circuit testing. These are two genetic-algorithm software packages for generating patterns to test sequential VLSI circuits. In our experiments, we used sequential circuits from the ISCAS89 benchmarks [13] 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.

4.9.1 CRIS

CRIS [55] is based on continuous mutations of a given input test sequence and on analyzing the mutated vectors to select a new test sequence. Hierarchical simulation techniques are used in the system to reduce memory requirements, thereby allowing its application to large VLSI circuits. The package has been applied successfully to generate test patterns that can detect nearly all the detectable faults (high fault coverage) 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. An HM targeted for improvement is a set of eight parameters used in CRIS (see Table 4.1). 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 from Table 4.2.)

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 tuning 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 [55].

Our goal is to develop one common HM that works 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 tuning.

4.9.2 GATEST

GATEST [54] is another test-pattern generator 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

Table 4.1: Parameters in CRIS treated as an HM in learning and in generalization. (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.)

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	-

Table 4.2: Sequence of random seeds used in learning experiments for CRIS, GATEST, and TimberWolf.

Sequence of Random Seeds				
61801	98052	15213	48823	55414
60203	43212	08540	94702	92715

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 [13], and execution times were significantly lower in most cases than those obtained by HITEC [49], a deterministic test-pattern generator.

The entire genetic-algorithm process was divided into four phases, each with its own fitness function that had been manually tuned by the designers. The designers also told us that Phase 2 of their package had the largest impact on performance and recommended that we improved it first. As a result, we treated 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

$$fitness_2 = \#\_faults\_detected + \frac{\#\_faults\_propagated\_to\_flip\_flops}{(\#\_faults)(\#\_flip\_flops)} \quad (4.1)$$

In learning a new fitness function, we have used the following variables as possible arguments of the function: *#_faults*, *#_faults_detected*, *#_circuit_nodes*, *#_flip_flops*, *#_faults_propagated_to_flip_flops*, and *sequence_length*. The operators allowed to compose new fitness functions include +, −, *, and /.

4.9.3 Experimental results

In our experiments, we chose five circuits as our learning subdomains. In each of these subdomains, we used Teacher to test CRIS 1000 times with different HMs, each represented as the first seven parameters in Table 4.1. 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 4.1 with values from Table 4.2). 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

ten circuits. Since there is no incumbent HM, we use the median performance value of each test case as the baseline performance for that test case. The elements of the generalized HM found are shown in Table 4.1.

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 (five from each subdomain) by first evaluating them fully (each with ten random seeds from Table 4.2) on the six subdomains and by selecting one generalized HM for all circuits. Since there is an incumbent HM, we use the performance of the incumbent HM as our baseline for improvement. The final fitness function we got after generalization is

$$fitness_2 = 2 \times \#\_faults\_propagated\_to\_flip\_flops - \#\_faults\_detected \quad (4.2)$$

Table 4.3 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 [55] and GATEST [54] as well as those of HITEC [49]. Note that the maximum fault coverages reported in Table 4.3 were based on ten runs of the underlying problem solver, implying that the computational cost is ten times the average cost.

Table 4.4 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 better than 11 out of 21 circuits in terms of the average fault coverage. Furthermore, 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 the maximum fault coverages. Note that the average fault coverages of our generalized HM are better than or equal to the original GATEST in all subdomains used in the heuristics-design process. Our results show that our generalization procedure can discover good HMs that work better than the original HMs.

Circuit ID	Total Faults	Fault Coverage				Cost		CRIS Gen. HM			GATEST Gen. 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	15984.0	128.6	84.7	86.4	10.9	85.9	86.0	126.4
s344	342	95.9	93.7	96.2	96.2	4788.0	134.8	96.1	96.2	21.8	96.2	96.2	133.3
s349	350	95.7	–	95.7	95.7	3132.0	136.9	95.6	95.7	21.9	95.7	95.7	128.3
+s382	399	90.9	68.6	87.0	87.5	43200.0	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	43560.0	229.3	71.2	85.7	8.4	85.7	86.6	215.1
s444	474	87.3	83.7	85.6	86.3	57960.0	259.4	79.8	85.4	9.3	85.6	86.3	233.8
*s526	555	65.7	77.1	75.1	76.4	168480.0	333.4	70.0	77.1	10.0	75.5	77.3	302.7
s641	467	86.5	85.2	86.5	86.5	1080.0	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	57960.0	266.4	44.7	46.7	51.3	69.3	80.9	225.4
*s832	870	93.9	42.5	61.9	66.8	63360.0	265.8	44.1	45.6	44.6	66.9	72.8	251.0
s1196	1242	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	1355	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	1486	97.0	91.2	93.7	96.0	129960.0	512.3	94.1	95.2	85.6	94.3	96.5	553.4
+s1423	1515	40.0	77.0	81.0	86.3	–	3673.9	82.0	88.3	210.4	81.3	87.3	4325.7
+s1494	1506	96.4	90.1	94.0	95.8	6876.0	510.4	93.2	94.1	85.5	93.6	95.6	584.3
+s5378	4603	70.3	65.8	69.5	70.1	–	9973.3	65.3	69.9	501.8	69.6	71.9	8875.7
s35932	39094	89.3	88.2	89.5	89.7	13680.0	184316.0	77.9	78.4	4265.7	89.4	89.7	184417.0
am2910	2573	85.0	83.0	–	–	–	–	83.7	85.2	307.6	–	–	–
+div16	2147	72.0	75.0	–	–	–	–	79.1	81.0	149.9	–	–	–
tc100	1979	80.6	70.8	–	–	–	–	72.6	75.9	163.8	–	–	–

Table 4.3: Performance of HMs in terms of computational cost and fault coverage for CRIS and GATEST. (Learned subdomains for CRIS are marked by "*" and generalized subdomains by "+"). Performance of HITEC is from the literature [49, 55]. Costs of our experiments are running times in seconds on a Sun SparcStation 10/51; costs of HITEC are running times in seconds on a Sun SparcStation SLC [54] (a computer around 4-6 times slower than a Sun SparcStation 10/51).

Table 4.4: Summary of results comparing the performance of our generalized HMs with respect to those of HITEC, CRIS, and GATEST. (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.)

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 & GATEST	19	3, 3, 13	1, 0, 18	19	3+2, 4, 10	2+2, 2, 13
HITEC, CRIS, & GATEST	18	2, 3, 13	1, 0, 17	18	3+2, 3, 10	1+2, 1, 14

Table 4.4 also indicates that HITEC is still better than our new generalized HM for CRIS in most of the circuits (in 14 out of 21 in terms of the maximal fault coverage and in 17 out of 21 in terms of the average fault coverage). The poor performance of our generalized HM as compared to HITECH is attributed to the limitations in CRIS and by our HM generator. Such limitations cannot be overcome without using HITECH as our problem solver or without using more powerful HM generator.

Finally, we plot the distributions of symmetric fault coverages of our generalized HMs normalized with respect to the average fault coverages of the original CRIS (Figure 4.10) and GATEST (Figure 4.11). These plots clearly demonstrate improvements over the original HMs.

4.10 Conclusions

In this chapter, we have studied the following five issues in learning improved knowledge-lean heuristics using a genetics-based machine learning approach:

- Decomposition of a problem solver into components so that heuristics can be learned for each, and integration of the heuristics

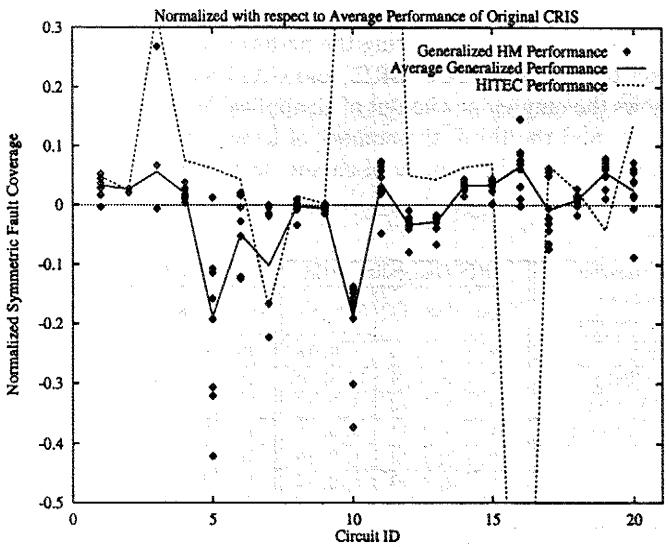


Figure 4.10: Distribution of the normalized symmetric fault coverages of our generalized HM with respect to the average fault coverages of the original CRIS on 20 benchmark circuits (s298, s344, s382, s386, s400, s444, s526, s641, s713, s820, s832, s1196, s1238, s1488, s1494, s1423, s5378, am2910, div16, and tc100 in that order).

learned into the problem solver,

- Partitioning of test cases in an application domain into subdomains so that performance data of heuristics within each subdomain are independent and identically distributed,
- Generation of new heuristics based on performance of existing heuristics evaluated in the past,
- Full evaluation of heuristics to verify their performance,
- Generalization of performance of heuristics to find heuristics that perform well across the entire application domain.

To address these issues, we have developed Teacher, a genetics-based system for learning knowledge-lean heuristics. Teacher has four phases in its operation: (a) classification of test cases into subdomains, (b)

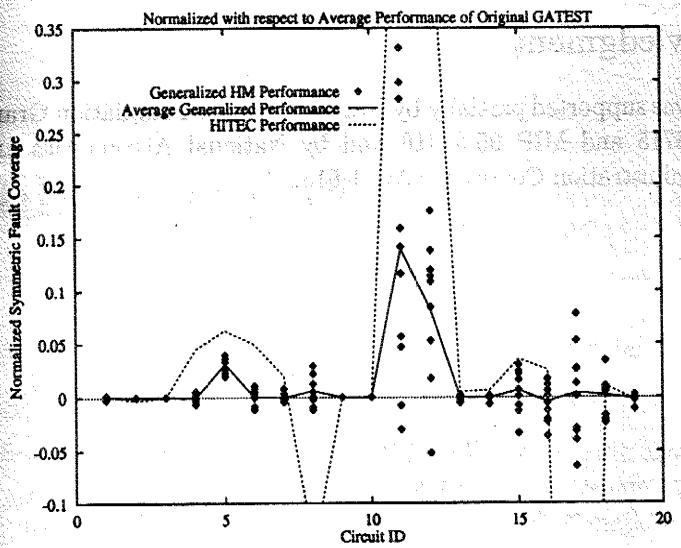


Figure 4.11: Distribution of the normalized symmetric fault coverages of our generalized HM with respect to the average fault coverages of the original GATEST on 19 benchmark circuits (s298, s344, s349, s382, s386, s400, s444, s526, s641, s713, s820, s832, s1196, s1238, s1488, s1494, s1423, s5378, and s35932 in that order).

learning improved heuristics for one subdomain at a time under resource constraints, (c) verification of performance of heuristics learned to augment partial evaluation results obtained during learning, and (d) finding heuristics whose performance behavior can be generalized to subdomains not studied in learning.

Finally, we have shown improved heuristics for two genetic algorithm packages used in VLSI test-pattern generation and have demonstrated that improved heuristics learned can be generalized to new circuits. Our approach allows designers to apply circuit-independent heuristics in these systems, eliminating the time-consuming process to find good heuristics for each circuit.

Acknowledgment

Research was supported partially by National Science Foundation Grants MIP 92-18715 and MIP 96-32316 and by National Aeronautics and Space Administration Contract NAG 1-613.

Bibliography

[1] D. H. Ackley, *A Connectionist Machine for Genetic Hillclimbing.* Boston, MA: Kluwer Academic Pub., 1987.

[2] A. Aizawa and B. W. Wah, "Scheduling of genetic algorithms in a noisy environment," in *Proc. Int'l Conf. on Genetic Algorithms*, (Morgan Kaufman), pp. 48–55, Int'l Soc. for Genetic Algorithms, July 1993.

[3] A. Aizawa and B. W. Wah, "Scheduling of genetic algorithms in a noisy environment," in *Proc. Int'l Conf. on Genetic Algorithms*, (Morgan Kaufman), pp. 48–55, Int'l Soc. for Genetic Algorithms, July 1993.

[4] 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.

[5] 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.

[6] A. N. Aizawa and B. W. Wah, "A sequential sampling procedure for genetic algorithms," *Computers and Mathematics with Applications*, vol. 27, pp. 77–82, May 1994.

[7] A. N. Aizawa and B. W. Wah, "A sequential sampling procedure for genetic algorithms," *Computers and Mathematics with Applications*, vol. 27, pp. 77–82, May 1994.