29. J. Lienig, "Channel and Switchbox Routing with Minimized Crosstalk – A Parallel Genetic Approach", *Proceedings of the 10th International Conference on VLSI Design*, pp.27–31, Jan. 1997.
30. J. Lienig and K. Thulasiraman, "A Genetic Algorithm for Channel Routing in VLSI Circuits," *Evolutionary Computation*, vol. 1, no. 4, pp. 293-311, 1994.
31. J. Lienig and K. Thulasiraman, "GASBOR: A Genetic Algorithm for Switchbox Routing in Integrated Circuits," *Progress in Evolutionary Computation*, X. Yao, ed., Lecture Notes in Artificial Intelligence, vol. 956, Berlin: Springer Verlag, pp. 187-200, 1995.
32. Y.-L. Lin, Y.-C. Hsu and F.-S. Tsai, "SILK: A Simulated Evolution Router," *IEEE Trans. on Computer-Aided Design*, vol. 8, no. 10, pp. 1108-1114, Oct. 1989.
33. S. Mohan and P. Mazumder, "Wolverines: Standard Cell Placement on a Network of Workstations," *IEEE Trans. on Computer-Aided Design*, vol. 12, no. 9, pp. 1312-1326, Sept. 1993.
34. B. T. Preas, "Benchmarks for Cell-based Layout Systems," *Proc. of the ACM-IEEE Design Automation Conference*, pp. 319-320, 1987.
35. *Proc. of the ACM-IEEE Design Automation Conference*, 1984.
36. *Proc. of the ACM-IEEE Design Automation Conference*, 1987.
37. Y. Saab and V. Rao, "An Evolution-Based Approach to Partitioning ASIC Systems," *Proc. of the ACM-IEEE Design Automation Conference*, pp. 767-770, 1989.
38. C. Sechen, *VLSI Placement and Global Routing Using Simulated Annealing*, Boston, MA: Kluwer Academic Publishers, 1988.
39. K. Shahookar and P. Mazumder, "GASP - A Genetic Algorithm for Standard Cell Placement," *Proc. of the European Design Automation Conference*, pp. 660-664, 1990.
40. K. Shahookar and P. Mazumder, "A Genetic Approach to Standard Cell Placement using Meta-Genetic Parameter Optimization", *IEEE Trans. on Computer-Aided Design*, vol. 9, no. 5, pp. 500-511, May 1990.
41. K. Shahookar, W. Khamisani, P. Mazumder and S. M. Reddy, "Genetic Beam Search for Gate Matrix Layout," *Proc. of the 6th International Conference on VLSI Design*, pp. 208-213, Jan. 1993.
42. J. M. Varanelli and J. P. Cohoon, "Population-Oriented Simulated Annealing: A Genetic/Thermodynamic Hybrid Approach to Optimization," *Proc. of the Sixth International Conference on Genetic Algorithms*, pp. 174-181, 1995.

Statistical Generalization of Performance-Related Heuristics for Knowledge-Lean Applications

Arthur Ieumwananonthachai and Benjamin W. Wah

Department of Electrical and Computer Engineering and the Coordinated Science Laboratory, University of Illinois at Urbana-Champaign, 1308 West Main Street, Urbana, IL 61801

Summary. In this chapter, we present new results on the automated generalization of performance-related heuristics learned for knowledge-lean applications. By first applying genetics-based learning to learn new heuristics for some small subsets of test cases in a problem space, we study methods to generalize these heuristics to unlearned subdomains of test cases. Our method uses a new statistical metric called probability of win. By assessing the performance of heuristics in a range-independent and distribution-independent manner, we can compare heuristics across problem subdomains in a consistent manner. To illustrate our approach, we show experimental results on generalizing heuristics learned for sequential circuit testing, VLSI cell placement and routing, branch-and-bound search, and blind equalization. We show that generalization can lead to new and robust heuristics that perform better than the original heuristics across test cases of different characteristics.

1. Introduction

Heuristics or heuristic methods (HMs), in general terms, are "Strategies using readily accessible though loosely applicable information to control problem-solving processes in human being and machines" [12]. They exist as problem solving procedures in problem solvers to find (usually) suboptimal solutions for many engineering applications. Since their design depends on user experience and is rather ad hoc, it is desirable to acquire them automatically by machine learning.

We make the following assumptions in this chapter. First, we assume that the applications are *knowledge-lean*, implying that domain knowledge for credit assignment is missing. In this class of applications, we are interested to learn and generalize *performance-related* HMs whose goal is to find solutions with the best numerical performance. Examples of targeted HMs and applications include symbolic formulae for decision making in a branch-and-bound search and a set of numerical parameters used in a simulated annealing package for placement and routing of VLSI circuits. (See Section 4.).

Second, we assume that performance of a HM is characterized by one or more statistical metrics and is obtained by evaluating multiple test cases (noisy evaluations). We further assume that a HM may have different performance distributions across different subsets of test cases in the problem space, thereby disallowing the use of performance metrics such as the average.

For example, given two heuristic methods HM_1 and HM_2 and two subsets of test cases TC_1 and TC_2, assume that cost is the performance measure of HMs. Suppose after testing HM_1 on the two subsets of test cases, we found its average costs be 10 and 100 units, respectively. Similarly, we got 150 and 5 units for HM_2 on the two subsets of test cases. It will be difficult to say whether HM_1 is better than HM_2 in terms of cost, and which HM should be used as a general HM for all test cases in the problem domain.

Third, we assume that heuristics used in generalization are learned by a genetics-based learning method [18, 8]. This is a form of learning by induction that involves applying genetic algorithms [3] to machine learning problems. There are two steps involved in this learning method:

- *Generation* and *selection* of HMs that can better solve test cases used in learning, as compared to the best existing (*baseline*) HMs;
- *Generalization* of the selected HMs to test cases not seen in learning with the same high level of performance as compared to that of the baseline HMs.

As illustrated in Figure 1.1, these two steps are generally separated in genetics-based learning.

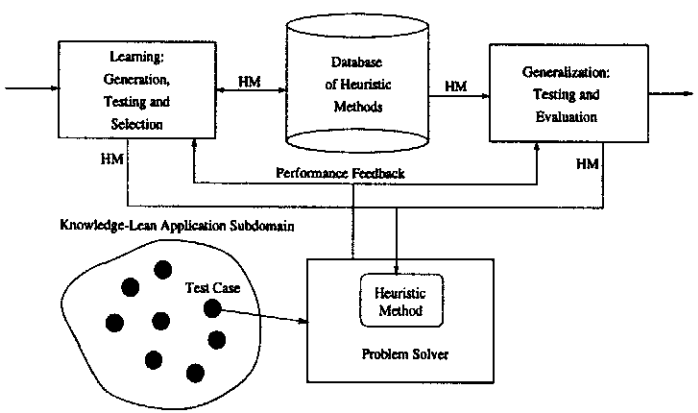

Fig. 1.1. Learning and generalization in knowledge-lean applications is based on evaluating a heuristic method on a test case and on observing its performance feedback

In this chapter, we study *statistical generalization of HMs across test cases of an application with different performance distributions*. The problem is illustrated in Figure 1.2 in which we show three heuristic methods and three subsets (or subdomains) of test cases in an application domain. Let $p_{i,j}$ be the performance of HM_i on Subdomain j, in which we assume that the

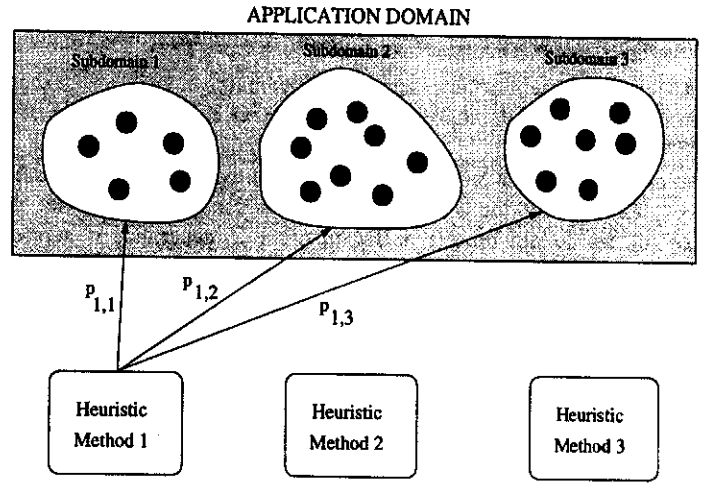

Fig. 1.2. Performance of a heuristic method may vary significantly across different subsets of test cases in an application domain, making it difficult to combine these numbers into a single performance number

performance of an HM in a subdomain can be aggregated into a single value. When an HM behaves differently across different subdomains of test cases, it will not be possible to aggregate its performance values across subdomains into a single number. Further, when one HM performs better than another HM in one subdomain but worse in another, we need to develop a method to differentiate HMs with high performance from those with low performance across all test cases in the application.

Generalization is important because learning time is often limited, and only a small set of test cases can be evaluated during learning. Generalization in many existing genetics-based learning systems [8, 3, 2] is a post-learning verification phase that simply verifies the generalizability of the learned HMs by evaluating them on a new set of test cases. This approach is suitable when test cases used in learning are representatives of *all* the test cases targeted by the HM. When test cases used in generalization have different characteristics, the HMs learned cannot be generalized.

To compare HMs bearing different performance distributions across different subsets of test cases in an application, we need to develop a performance metric that is independent of the actual distributions. We propose in this chapter a new metric called *probability of win* that measures the probability that a particular HM is better than another randomly chosen HM from a set of learned HMs for a given subset of test cases. Since probabilities are between 0 and 1, we eliminate the dependence of HMs on actual performance distributions. Using this metric, we can verify whether a HM is generalizable

across test cases of different performance distributions. Our approach can be summarized as follows:

- Partition the domain of test cases into *subdomains* in such a way that performance values in a subdomain are *independent and identically distributed* (i.i.d.).
- Develop conditions under which a HM can be considered to perform well across multiple subdomains. In contrast to studies in artificial intelligence [6], we do not modify a HM in order to generalize it across subdomains. Rather, we test certain conditions to see if a HM is generalizable.

This chapter is divided into five sections. Section 2. defines problem space and its partitioning into subdomains. We propose in Section 3. a new metric called probability of win and a new generalization strategy. Section 4. reports our experimental results on four real-world applications — circuit testing, VLSI cell placement and routing, branch-and-bound search and blind equalization. Conclusions are drawn in Section 5.

2. Problem Domains and Subdomains

Given an application problem consisting of a collection of test cases, the first task in learning and generalization is to classify the test cases into *domains* such that a unique HM can be designed for each [13]. This classification step is domain specific and is generally carried out by experts in the area.

For instance, consider the problem of generating test patterns to test VLSI circuits. Previous experience shows that sequential circuits require tests that are different from those of combinatorial circuits. Consequently, we can consider combinatorial circuits and sequential circuits as two different problem domains.

In comparing the performance of HMs in a problem domain, 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 across different subsets of test cases in the domain. In this case, we need to decompose the domain into smaller partitions so that quantitative comparison of performance of HMs in a partition is possible. We define a *problem subdomain* as a partitioning of the domain of test cases such that performance values of a HM in a subdomain are i.i.d. Under this condition, it is meaningful to compute the average performance of test cases in a subdomain. It is important to point out that performance values may need to be normalized with respect to those of the baseline HM before aggregated.

We need to know the attributes of an application in order to classify its test cases, and a set of decision rules to identify the subdomain to which a test case belongs. For example, in learning new decomposition HMs in a branch-and-bound search for solving a traveling-salesman problem (Section 4.), we can treat graph connectivity as an attribute to classify graphs into subdomains.

In some applications, it may be difficult to determine the subdomain to which a test case belongs. This is true because the available attributes may not be well defined or may be too large to be useful. For instance, in test-pattern generation for sequential circuits, there are many attributes that can be used to characterize circuits (such as length of the longest path and maximum number of fan-in's and fan-out's). However, none of these attributes is a clear winner.

When we do not know the attributes to classify test cases into subdomains, we can treat each test case as a subdomain by itself. This works well when the HM to be learned has a random component: by using different random seeds in the HM, we can obtain statistically valid performance values of the HM on a test case. We have used this approach in the two circuit-related applications discussed in Section 4. and have chosen each circuit as an independent subdomain for learning.

After applying learning to find good HMs for each subdomain, we need to compare their performance across subdomains. This may be difficult because test cases in different subdomains of a domain may have different performance distributions, even though they can be evaluated by a common HM. As a result, the performance of test cases cannot be compared statistically. For instance, we cannot use the average metric when performance values are dependent or have multiple distributions.

As an example, Table 2.1 shows the average and maximum fault coverages of two HMs used in a test-pattern generator to test sequential circuits. The data indicate that we cannot average their fault coverages across the two circuits as the performance distribution of HM_{101} across the two circuits is not the same as that of HM_{535}.

Table 2.1. Maximum and average fault coverages of two HMs used in a test-pattern generator with different random seeds

Circuit	HM	Maximum FC	Average FC
S444	101	60.3	28.5
	535	86.3	84.8
S1196	101	94.9	94.2
	535	93.6	93.1

It should now be clear that there can be many subdomains in an application, and learning can only be performed on a small number of them. Consequently, it is important to generalize HMs learned for a small number of subdomains to unlearned subdomains. In some situations, multiple HMs

may have to be identified and applied together at a higher cost to find a high quality solution.

3. Generalization of Heuristic Methods Learned

Since learning can only cover a small subset of a problem space, it is necessary to generalize HMs developed to test cases not studied in learning. When test cases used in learning have the same performance distribution as those used in generalization, generalization simply involves verifying the performance results. However, as illustrated in the last section, test cases used in generalization may have different performance distribution for two reasons: (a) A learned HM has different performance distributions across subdomains. (b) The baseline HM used in normalization has different performance distributions across subdomains. In either case, performance values after normalization will have different distributions across subdomains. This leads us to develop a generalization strategy that can compare HMs across different subdomains with different performance distributions.

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, two issues are apparent here.

- How to compare the performance of HMs within a subdomain in a range-independent and distribution-independent fashion? Here, we need to evaluate and generalize the performance of a HM in a single subdomain in a range-independent and distribution-independent way.
- How to define the notion that one HM performs well across multiple subdomains?

Our method to address these two issues involves a new metric called probability of win. Informally, *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. It is important to point out that the HMs used in computing the probability of win are found by learning; hence, they already perform well within a subdomain. Further, probabilities of win are in the range zero to one, independent of the number of HMs evaluated and the distribution of performance values.

3.1 Performance evaluation within a subdomain

There are many ways to address the first issue raised above, and solutions to the second issue depend on the solution to the first. For instance, scaling and normalization of performance values is a possible way to compare performance in a distribution-independent manner; however, this may lead to new inconsistencies [18]. Another way is to rank HMs by their performance values and use the average ranks of HMs for comparison. This does not work well because it does not account for actual differences in performance values, and two HMs with very close or very different performance may differ only by one in their ranks. Further, the maximum rank of HMs depends on the number of HMs evaluated, thereby biasing the average ranks of individual HMs. In this section, we propose a metric called probability of win to select good HMs within a subdomain.

$P_{win}(h_i, d_m)$, the *probability-of-win* of HM h_i in subdomain d_m, is defined as the probability that the true mean of h_i (on one performance measure[1]) is better than the true mean of HM h_j randomly selected from the pool. When h_i is applied on test cases in d_m, we have

$$P_{win}(h_i, d_m) = \frac{\sum_{j \neq i} P\left[\mu_i^m > \mu_j^m | \hat{\mu}_i^m, \hat{\sigma}_i^m, n_i^m, \hat{\mu}_j^m, \hat{\sigma}_j^m, n_j^m\right]}{|s| - 1}, \qquad (3.1)$$

where $|s|$ is the number of HMs under consideration, 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.

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

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

To illustrate the concept, we show in Table 3.1 the probabilities of win of four HMs tested to various degrees. Note that P_{win} is not only related to the sample mean but also depends on the sample variance and number of tests performed. Further, the probability that h_i is better than h_j and the probability that h_j is better than h_i are both counted in the evaluation. Hence, the average of P_{win} over all HMs in a subdomain ($= \sum_i P_{win}(h_i, d_m)/|s|$) will be 0.5.

P_{win} defined in (3.1) is range-independent and distribution-independent because all performance values are transformed into probabilities between 0 and 1 independent of the number of HMs evaluated and the distribution of performance values. It assumes that all HMs are i.i.d. and takes into account uncertainty in their sample averages (by using their variances); hence, it is better than simple scaling that only compresses performance averages into a

[1] Due to space limitation, we do not consider issues dealing with multiple performance measures in this chapter.

Table 3.1. Probabilities of win of four HMs in d_m

h_i	$\hat{\mu}_i$	$\hat{\sigma}_i$	n_i	$P_{win}(h_i, d_m)$
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

range between 0 and 1. It is also important to point out that the HMs used in computing P_{win} are found by learning; hence, they already perform well within a subdomain.

3.2 Performance evaluation across subdomains

One of the major difficulties in handling multiple subdomains is that it may be difficult to aggregate performance values statistically from different subdomains, and to define the notion that one HM performs better than another across multiple subdomains. For instance, it is not meaningful to find an average of random numbers from two different distributions. We address this problem using P_{win} defined in the last subsection.

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

$$P_{win}(h, \Pi_p) = \frac{\sum_{d \in \Pi_p} P_{win}(h, d)}{|\Pi_p|}, \tag{3.2}$$

where Π_p is the p'th partition of subdomains in the problem domain. The HM picked is the one that maximizes (3.2). When subdomains are not equally likely but with known relative weights, we can compute P_{win} as a weighted average instead of (3.2). HMs picked using 3.2 generally wins with a high probability across most of the subdomains in Π_p but occasionally may not perform well in a few subdomains.

Second, we consider the problem of finding a good HM across multiple subdomains in Π_p as a multi-objective optimization problem. In this case, evaluating HMs based on a combined objective function (such as the average P_{win} in (3.2) 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 P_{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 $m \in \Pi_p$.

$$P_{win}(h, m) \geq (q^m_{win} - \delta) \qquad \forall m \in \Pi_p \tag{3.3}$$

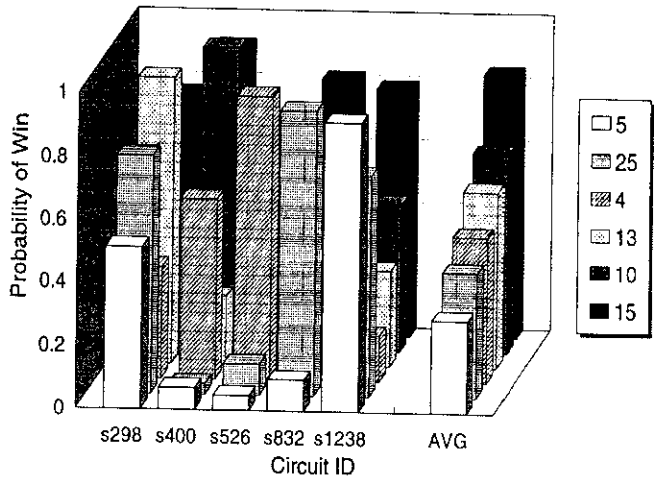

Fig. 3.1. P_{win} of six HMs across five subdomains in the test-pattern generation problem

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

To illustrate the generalization procedure, consider the test-pattern generation problem discussed in Section 2. Assume that learning had been performed on five circuits (subdomains), and that the six best HMs from each subdomain were reported. After full evaluation of the 30 HMs (initialized by ten random seeds) across all five subdomains, we computed P_{win} of each HM in every subdomain. Figure 3.1 shows the probabilities of win of six of these HMs. If we generalize HMs based on (3.2), then HM_{15} will be picked since it has the highest average P_{win}. Likewise, if we generalize using (3.3), we will also select HM_{15}. Note that in this example, no one HM is the best across all subdomains.

4. Experimental Results

To illustrate the generalization procedure described in Section 3., we present in this section results on generalization for two applications in VLSI design and branch-and-bound search. These results were obtained using TEACHER [18], a genetics-based learning system that implements our proposed generalization strategy. The parameters used during learning are shown in Table 4.1.

Table 4.1. Genetic-algorithm parameters used in our learning system. (*# HMs Verified at Termination* is the number of HMs selected for verification at the end of the last generation)

Application	CRIS	Timber-Wolf	Branch-and-Bound
Number of Generations	10	10	10
Duration of a Generation	100	100	160
# Active HMs in each Gen.	30	30	40
New HMs Generated in each Gen.	20	20	30
Crossover Rate	0.45	0.45	0.5
Mutation Rate	0.35	0.35	0.17
Random Generation Rate	0.20	0.20	0.33
# HMs Verified at Termination	20	20	20

4.1 HM for sequential circuit testing

The first application is based on CRIS [15], a genetic-algorithm software package for generating patterns to test sequential VLSI circuits. CRIS mutates an input test sequence continuously and analyzes the mutated vectors in selecting a test set. Since many copies of a circuit may be manufactured, it is desirable to obtain as high a fault coverage as possible, and computational cost is of secondary importance.

In our experiments, we used sequential circuits from the ISCAS89 benchmarks [1] plus several other larger circuits. We treat each circuit as an individual subdomain. Since we want one common HM for all circuits, we assume that all circuits are from one domain.

CRIS in our experiments is treated as a black-box problem solver, as we have minimal knowledge in its design. A HM targeted for improvement is a set of eight parameters used in CRIS (Table 4.2). 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.) 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. Note that in the original CRIS, the HM used for each circuit is unique and was tuned manually. The advantage of having one HM is that it can be applied to new circuits without further manual tuning.

In our experiments on CRIS, we chose five circuits as our learning subdomains. In each of these subdomains, we used TEACHER [18] to test CRIS 1000 times (divided into 10 generations) with different HMs. A HM in learning is represented as a tuple of the first seven parameters in Table 4.2. The majority of time was spent in testing the HMs generated, since the time to generate a HM is very small (involving the crossover or mutation of sets of seven parameters). At the end of learning, we picked the top twenty HMs

Table 4.2. Parameters of CRIS treated as a HM in learning and in generalization. (The type, range, and step of each parameter were given to us by the designer of CRIS. The default parameters were not given to us as they are circuit-dependent)

Parameter	Range	Step	Definition	New Value
P_1	1-10	1	related to the number of stages in a flip flop	1
P_2	1-40	1	sensitivity of state change of a flip flop	12
P_3	1-40	1	survival rate of a test sequence in next generation	38
P_4	0.1-10.0	0.1	number of test vec. concat. to form a new vec.	7.06
P_5	50-800	10	number of useless trials before quitting	623
P_6	1-20	1	number of generations	1
P_7	0.1-1.0	0.1	how genes are spliced in GA	0.1
P_8	Integer	1	seed for random number generator	-

in each subdomain and evaluated them fully by initializing CRIS using ten different random seeds (P_8 in 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 10 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 (3.2)). The HM found is shown in the last column in Table 4.2.

Table 4.3 shows the costs and qualities in applying our generalized HM learned for CRIS (see Table4.2) and compares them to the results of CRIS [15] and HITEC [10], the latter is a deterministic search algorithm that is often used as a benchmark algorithm. We do not have the cost figures of CRIS because they were not published. The designer of CRIS hand tuned the parameters for each circuit; hence, the time (or cost) for obtaining these parameters are very large. Note that the maximum fault coverages reported were based on ten runs of the underlying problem solver, implying that the computational cost is ten times of the average cost. Recall that we like to obtain the maximum coverage of a circuit, and that computational cost is a secondary issue in circuit testing. Table 4.4 summarizes the results shown in Table 4.3.

Our results show that our generalization procedure can discover new HMs that are better than the original HMs in 16 out of 22 circuits in terms of the maximum fault coverage, and in 11 out of 22 circuits in terms of the average fault coverage. Our results are significant in the following aspects:

- new faults detected by our generalized HMs were not discovered by previous methods;

Table 4.3. Performance of HMs in terms of computational cost and fault coverage for CRIS. (Learned subdomains for CRIS are marked by "*" and generalized subdomains by "+"). Performance of HITEC is from the literature [16, 11]. 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 [14] (around 4-6 times slower than a Sun SparcStation 10/51)

Circuit ID	Total Faults	Fault Coverage		Cost	CRIS Generalized HM		
		HITEC	CRIS	HITEC	Avg. FC	Max. FC	Avg. Cost
*s298	308	86.0	82.1	15984.0	84.7	86.4	10.9
s344	342	95.9	93.7	4788.0	96.1	96.2	21.8
s349	350	95.7	–	3132.0	95.6	95.7	21.9
+s382	399	90.9	68.6	43200.0	72.4	87.0	7.2
s386	384	81.7	76.0	61.8	77.5	78.9	3.5
*s400	426	89.9	84.7	43560.0	71.2	85.7	8.4
s444	474	87.3	83.7	57960.0	79.8	85.4	9.3
*s526	555	65.7	77.1	168480.0	70.0	77.1	10.0
s641	467	86.5	85.2	1080.0	85.0	86.1	19.5
+s713	581	81.9	81.7	91.2	81.3	81.9	23.0
s820	850	95.6	53.1	5796.0	44.7	46.7	51.3
*s832	870	93.9	42.5	6336.0	44.1	45.6	44.6
s1196	1242	99.7	95.0	91.8	92.0	94.1	20.0
*s1238	1355	94.6	90.7	132.0	88.2	89.2	23.0
s1488	1486	97.0	91.2	12960.0	94.1	95.2	85.6
+s1494	1506	96.4	90.1	6876.0	93.2	94.1	85.5
s1423	1515	40.0	77.0	–	82.0	88.3	210.4
+s5378	4603	70.3	65.8	–	65.3	69.9	501.8
s35932	39094	89.3	88.2	13680.0	77.9	78.4	4265.7
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 4.4. Summary of wins and losses in applying our generalized HM for CRIS on 22 circuits when compared to the performance of HITEC, CRIS, and the best of CRIS and HITEC. (Not all circuits were tested by HITEC and CRIS)

Our HM wins/ties with respect to the following	CRIS Generalized HM					
	Max. Fault Coverage			Avg. Fault Coverage		
	Wins	Ties	Losses	Wins	Ties	Losses
HITEC	6	2	14	4	0	18
CRIS	16	1	5	11	0	10
Best of HITEC and CRIS	5	3	14	3	0	9

– only one HM (rather than many circuit-dependent HMs in the original CRIS) was found for all circuits.

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 22 in term of the maximal fault coverage, and in 18 out of 22 in term of the average fault coverage). This happens because our generalized HM is bounded by the limitations in CRIS and our HM generator for CRIS. Such limitations cannot be overcome without generating more powerful HMs in our HM generator or using better test-pattern generators like HITEC as our baseline problem solver.

4.2 HM for VLSI placement and routing

In our second application, we use TimberWolf [17] as our problem solver. This is a software package based on simulated annealing (SA) [7] to place and route various circuit components on a piece of silicon. Its goal is to minimize the chip area needed while satisfying constraints such as the number of layers of poly-silicon for routing and the maximum signal delay through any path. Its operations can be divided into three steps: placement, global routing, and detailed routing.

The placement and routing problem is NP-hard; hence, heuristics are generally used. 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. Consequently, there is a trade-off between the quality of a result and the cost (or computational time) of obtaining it. In TimberWolf version 6.0, the version we have studied, there are two parameters to control the running time (which indirectly control the quality of the result): *fast-n* and *slow-n*. The larger the *fast-n* is, the shorter time SA will run. In contrast, the larger the *slow-n* is, the longer time SA will run. Of course, only one of these parameters can be used at any time.

TimberWolf has six major components: *cost function*, *generate function*, *initial temperature*, *temperature decrement*, *equilibrium condition*, and *stopping criterion*. Many parameters in these components have been tuned manually. However, their settings are generally heuristic because we lack domain knowledge to set them optimally. In Table 4.5, we list the parameters we have focused in this study. Our goal is to illustrate the power of our learning and generalization procedures and to show improved quality and reduced cost for the placement and routing of large circuits, despite the fact that only small circuits were used in learning.

In our experiments, we used seven benchmark circuits [9] (*s298*, *s420*, *fract*, *primary1*, *struct*, *primary2*, *industrial1*) that were mostly from ftp.mcnc.org in /pub/benchmark. We studied only the standard-cell placement problem, noting that other kinds of placement can be studied in a similar fashion. We used *fast-n* values of 1, 5, and 10, respectively.

We first applied TEACHER to learn good HMs for circuits *s298* with *fast-n* of 1, *s420* with *fast-n* of 5, and *primary1* with *fast-n* of 10, each of which was taken as a learning subdomain. We used a fixed sequence of ten

Table 4.5. Parameters of TimberWolf (Version 6) used in the original HM and after learning and generalization

Parameter	Range	Step	Meaning	Original	New
P_1	0.1 - 2.5	0.1	vertical path weight for estimating the cost function	1.0	0.958
P_2	0.1 - 2.5	0.1	vertical wire weight for estimating the cost function	1.0	0.232
P_3	3 - 10	1	orientation ratio	6	10
P_4	0.33 - 2.0	0.1	range limiter window change ratio	1.0	1.30
P_5	10.0 - 35.0	1.0	high temperature finishing point	23.0	10.04
P_6	50.0 - 99.0	1.0	intermediate temperature	81.0	63.70
P_7	100.0 - 150.0	1.0	low temperature finishing point	125.0	125.55
P_8	130.0 - 180.0	1.0	final iteration temperature	155.0	147.99
P_9	0.29 - 0.59	0.01	critical ratio that determines acceptance probability	0.44	0.333
P_{10}	0.01 - 0.12	0.01	temperature for controller turn off	0.06	0.112
P_{11}	integer	1	seed for the random number generator	-	-

random seeds (P_{11} in Table 4.5) in each subdomain to find the statistical performance of a HM. Each learning experiment involved 1000 applications of TimberWolf divided into ten generations. Based on the best 30 HMs (10 from each subdomain), we applied our generalization procedure to obtain one generalized HM. This generalized HM as well as the default HM are shown in Table 4.5.

Figure 4.1 plots the quality (higher quality in the y-axis means reduced chip area averaged over 10 runs using the defined random seeds) and cost (average execution time of TimberWolf) between the generalized HM and the default HM on all seven circuits with *fast-n* of 1, 5, and 10, respectively. Note that all performance values in Figure 4.1 are normalized with respect to those of *fast-n* of 10, and that the positive (resp., negative) portion of the x-axes shows the fractional improvement (resp., degradation) in computational cost with respect to the baseline HM using *fast-n* of 10 for the same circuit. Each arrow in this figure points from the average performance of the default HM to the average performance of the generalized HM.

The equation for computing the normalized symmetric cost is as follows. Let C_{new}, C_{base} and C_{sym}^{norm} be, respectively, the costs of the new HM, the cost of the baseline HM, and the normalized symmetric cost.

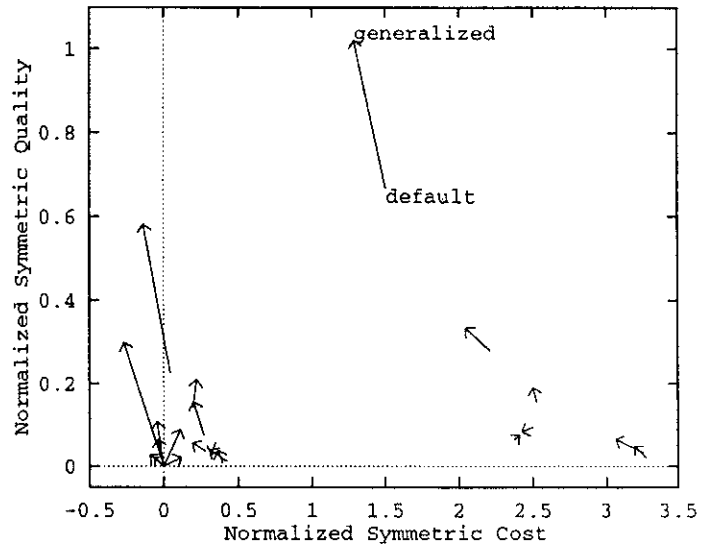

Fig. 4.1. Comparison of normalized average performance between the default and the generalized HMs. The plots are normalized with respect to the performance of applying the baseline HM on each circuit using *fast-n* = 10. (See (4.1))

$$C_{sym}^{norm} = \begin{cases} \frac{C_{new}}{C_{base}} - 1 & \text{if } C_{new} \geq C_{base} \\ 1 - \frac{C_{base}}{C_{new}} & \text{if } C_{new} < C_{base} \end{cases} \tag{4.1}$$

The reason for using the above equation is to avoid uneven compression of the ratio C_{new}/C_{base}. This ratio is between 0 and 1 when $C_{new} < C_{base}$, but is between 1 and ∞ when $C_{new} > C_{base}$. (4.1) allows increases in cost to be normalized in the range between 0 and ∞, and decreases to be normalized in the range between 0 and $-\infty$. The normalized symmetric quality in the y-axis is computed in a similar way.

Among the 22 test cases, the generalized HM has worse quality than that of the default in only two instances, and has worse cost in 4 out of 22 cases. We see in Figure 4.1 that most of the arrows point in a left-upward direction, implying improved quality and reduced cost. Note that these experiments are meant to illustrate the power of our generalization procedure. We expect to see more improvement as we learn other functions and parameters in TimberWolf. Further, improvements in TimberWolf are important as the system is actually used in industry.

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: (a) *Selection HM* for selecting a search node for expansion based on a sequence of selection keys for ordering search nodes; (b) *Decomposition HM* (or branching mechanism) for expanding a search node into descendants using operators to expand (or transform) a search node into child nodes; (c) *Pruning HM* for pruning inferior nodes in order to trim potentially poor subtrees; and (d) *Termination HM* for determining when to stop. In this subsection, we apply learning to find new *decomposition HMs* for expanding a search node into descendants.

We illustrate our method on three applications: traveling salesman problem (TSP) on incompletely connected graphs mapped on a two-dimensional plane, vertex-cover problem (VC), and knapsack problem (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 4.6 shows the parameters used in generating a *test case* in each application. We assume that each problem constitutes one domain.

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.

Table 4.6. 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 $(0.05 - 0.6)$ with step size 0.05 • Number of vertices is between 16 and 45
TSP	• Distributions of 8-18 cities ($U(0,100)$ on both X and Y axes, $N(50,12.5)$ on both axes, or $U(0,100)$ and $N(50,12.5)$ on different 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)} • σ^2 of profit/weight ratio is (1.05, 1.5, 10, 100) • 13-60 objects in the knapsack

Table 4.7. Original and generalized decomposition HMs used in a branch-and-bound search (l: number of uncovered edges or live degree of a vertex; n: average live degree of all neighbors; Δl: difference between l of parent node and l of current node; c: length of current partial tour; m: minimum length to complete current tour; p: profit of object; w: weight of object)

Application	Original HM	Generalized HM
VC	l	$1000\,l + n - \Delta l$
TSP	c	$m\,c$
KS	p/w	p/w

We have used well-known decomposition HMs developed for these applications as our baseline HMs (see Table 4.7). The normalized cost of a candidate decomposition HM is defined in terms of its *average symmetric speedup* (see Eq. (4.1) in Section 4.2), 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 subdomain using 1,600 tests, selected

Table 4.8. Results of generalization for VC, TSP, and KS. (In the results on generalization, numbers with "*" are the ones learned; only one common HM is generalized to all 12 subdomains)

Subdomain	Subdomain Performance (Sym-SU)					
	Learning			Generalization		
	VC	TSP	KS	VC	TSP	KS
1	0.218	0.072*	0.000*	0.070	0.417	0.000
2	0.283*	0.004	0.000*	0.638	0.036	0.000
3	0.031	0.082*	0.000	0.241	0.144	0.000
4	0.068*	0.225	0.000	0.078	0.155	0.000
5	0.054	0.005*	0.000	0.073	0.131	0.000
6	0.060*	0.061*	0.000*	0.020	0.364	0.000
7	0.017	0.139	0.000*	−0.013	1.161	0.000
8	0.049*	0.155	0.000	−0.004	0.101	0.000
9	0.016	−0.010	0.000*	−0.018	0.108	0.000
10	−0.000*	0.054	0.000	−0.000	0.008	0.000
11	−0.011	0.090*	0.000	−0.019	0.022	0.000
12	0.028*	0.083*	0.000*	−0.010	0.131	0.000
Average	0.068	0.080	0.000	0.088	0.231	0.00

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. (See (3.2).) Table 4.8 summarizes the generalization and validation results.

We show in our results 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 1-35% improvement in its average symmetric speedups and 3,000-16,000 times in learning costs.

Table 4.8 also shows the average symmetric speedups of the generalized HMs. We picked six subdomains randomly for learning. After learning and fully verifying the five top HMs in each subdomain, we applied (3.2) to identify one top HM to be used across all the twelve subdomains. Our results show that we have between 0-8% improvement in average symmetric speedups using the generalized HMs. Note that these results are worse than those obtained by learning, Moreover, the baseline HM is the best HM for solving the knapsack problem.

The second part of Table 4.8 shows the average symmetric speedups when we validate the generalized HMs on larger test cases. These test cases generally require 10-50 times more nodes expanded than those used earlier. Surprisingly, our results show better improvement (9-23%). It is interesting to point out that six of the twelve subdomains with high degree of connectivity in the vertex-cover problem have slowdowns. This is a clear indication that these subdomains should be grouped in a different domain and learned separately.

Table 4.7 shows the new decomposition HMs learned for the three applications that lists the variables used in the HMs. Note that we have included constants in our HMs in learning; an example of which 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 l's of two alternatives are different, then the remaining terms in the formula $(n - \Delta l)$ are insignificant. On the other hand, when the l's are the same, then we use $(n - \Delta l)$ as a tie breaker.

In short, our results show that reasonable improvements can be obtained by generalization of learned HMs. We anticipate further improvements by

- learning and generalizing new pruning HMs in a depth-first search,
- partitioning the problem space into a number of domains and learning a new HM for each, and
- identifying attributes that help explain why one HM performs well in one subdomain but not in others.

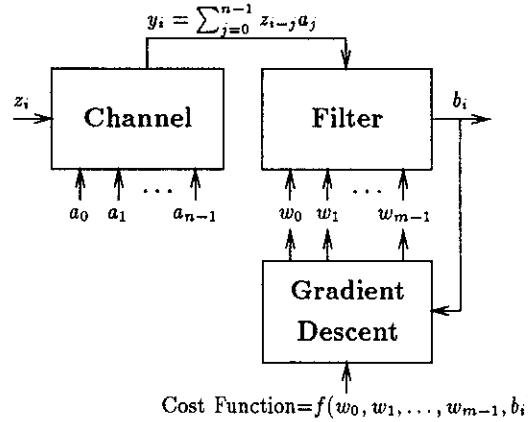

Fig. 4.2. Blind equalization process for recovering input data stream for n-th order channel and m-th order filter

Table 4.9. Summary of average symmetric improvements in terms of number of accumulated errors for the learned cost function over ten subdomains. (b_i in Figure 4.2 is the instantaneous value of b)

Average Symmetric Improvement				Original HM	New HM
Average	Std.Dev.	Maximum	Minimum		
0.153	0.395	0.694	−0.465	$b^3 - b$	$4b^3 - 2b^2 Sign(b) - b$

4.4 Blind equalization

Our last application is on applying genetic algorithms to learn a cost function in blind equalization. Our goal is to minimize the number of accumulated errors for a sequence of input data corrupted in transmission (Figure 4.2). The process is equivalent to adjusting the weights of an FIR filter using gradient descent in order to minimize the value of a cost function, which is defined in term of the weights of the filter and its current output.

In this application, we define a test case as multiple random sequences of data of fixed length passing through a fixed channel and a blind equalizer with given random initial weights. We further define a subdomain to be all test cases with the same channel specification. In our experiments, we attempt to cover all possible third-order channels: from relatively easy ones ($|a_i| > \sum_{i \neq j} |a_j|$ where a_i is the i-th weight of the channel) to the hardest one ($a_i = a_j$ for all i and j).

Table 4.9 shows the average symmetric improvements in terms of number of accumulated errors for HM_{base} (CMA 2-2) [4] and the new HM found after learning and generalization.

5. Conclusions

In this chapter, we have presented a method for generalizing performance-related heuristics learned by genetics-based learning for knowledge-lean applications. We have focused on a class of heuristic methods (HMs) whose performance is evaluated statistically by applying them on multiple test cases. Due to a lack of domain-knowledge for improving such heuristics, we have used a genetics-based learning paradigm (a generate-and-test method) to learn new HMs.

One of the major problems in performance evaluation of heuristics is that a HM may have different performance distributions across different sets of test cases in an application. This renders it impossible to use statistical metrics, such as average, to compare their performance.

We have proposed in this chapter a new metric called probability of win to characterize the performance of heuristics. This metric evaluates the probability that the mean performance of a HM is better than the mean performance of another randomly chosen HM in a set of learned HMs on a common set of test cases. The only requirement on the choice of test cases in evaluating probabilities of win is that each HM, when evaluated on the test cases, produces a set of independent and identically distributed performance results. We define such a set of test cases as a subdomain. Since probabilities of win are between 0 and 1, we can compare them across subdomains in generalizing HMs.

We have developed TEACHER [5], an integrated system that incorporates the learning and generalization method presented in this chapter. The system is relatively easy to use: the design of an interface between an application program and TEACHER usually takes less than two weeks to complete.

We have applied TEACHER [18], a genetics-based learning system that incorporates our generalization method, on four engineering applications and found very good improvements over existing HMs. These applications are hard to improve because they have been studied and tuned extensively by many others before. In each case, we have found very good improvements over existing HMs for these applications. These demonstrate that learning and generalization is important in refining heuristics used in many application problem solvers.

Acknowledgments

The authors would like to thank Mr. Yong-Cheng Li for interfacing TEACHER to TimberWolf and for collecting some preliminary results in Section 4.. This 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.

References

1. F. Brglez, D. Bryan, and K. Kozminski, "Combinatorial profiles of sequential benchmark circuits," in *Int'l Symposium on Circuits and Systems*, pp. 1929–1934, May 1989.
2. C. M. Fonseca and P. J. Fleming, "Genetic algorithms for multiobjective optimization: Formulation, discussion, and generalization," in *Proc. of the Fifth Int'l Conf. on Genetic Algorithms*, (Morgan Kaufman), pp. 416–423, Int'l Soc. for Genetic Algorithms, June 1993.
3. D. E. Goldberg and J. H. Holland, "Genetic algorithms and machine learning," *Machine Learning*, vol. 3, pp. 95–100, Oct. 1988.
4. S. Haykin, *Blind Deconvolution*. Englewood Cliffs, NJ: Prentice Hall, 1994.
5. A. Ieumwananonthachai and B. W. Wah, "TEACHER – an automated system for learning knowledge-lean heuristics," Tech. Rep. CRHC-95-08, Center for Reliable and High Performance Computing, Coordinated Science Laboratory, Univ. of Illinois, Urbana, IL, March 1995.
6. C. Z. Janikow, "A knowledge-intensive genetic algorithm for supervised learning," *Machine Learning*, vol. 13, no. 2-3, pp. 189–228, 1993.
7. S. Kirkpatrick, J. C. D. Gelatt, and M. P. Vecchi, "Optimization by simulated annealing," *Science*, vol. 220, pp. 671–680, May 1983.
8. J. R. Koza, *Genetic Programming*. Cambridge, MA: The MIT Press, 1992.
9. LayoutSynth92, *International Workshop on Layout Synthesis*. ftp site: mcnc.mcnc.org in directory /pub/benchmark, 1992.
10. T. M. Niermann and J. H. Patel, "HITEC: A test generation package for sequential circuits," in *European Design Automation Conference*, pp. 214–218, 1991.
11. T. M. Niermann and J. H. Patel, "HITEC: A test generation package for sequential circuits," in *European Design Automation Conference*, pp. 214–218, 1991.
12. J. Pearl, *Heuristics–Intelligent Search Strategies for Computer Problem Solving*. Reading, MA: Addison-Wesley, 1984.
13. C. L. Ramsey and J. J. Grefenstette, "Case-based initialization of genetic algorithms," in *Proc. of the Fifth Int'l Conf. on Genetic Algorithms*, (Morgan Kaufman), pp. 84–91, Int'l Soc. for Genetic Algorithms, June 1993.
14. E. M. Rudnick, J. H. Patel, G. S. Greenstein, and T. M. Niermann, "Sequential circuit test generation in a genetic algorithm framework," in *Proc. Design Automation Conf.*, ACM/IEEE, June 1994.
15. D. G. Saab, Y. G. Saab, and J. A. Abraham, "CRIS: A test cultivation program for sequential VLSI circuits," in *Proc. of Int'l Conf. on Computer Aided Design*, (Santa Clara, CA), pp. 216–219, IEEE, Nov. 1992.
16. D. G. Saab, Y. G. Saab, and J. A. Abraham, "CRIS: A test cultivation program for sequential VLSI circuits," in *Proc. of Int'l Conf. on Computer Aided Design*, (Santa Clara, CA), pp. 216–219, IEEE, Nov. 8-12 1992.
17. C. Sechen, *VLSI Placement and Global Routing Using Simulated Annealing*. Boston, MA: Kluwer Academic Publishers, 1988.
18. B. W. Wah, A. Ieumwananonthachai, L. C. Chu, and A. Aizawa, "Genetics-based learning of new heuristics: Rational scheduling of experiments and generalization," *IEEE Trans. on Knowledge and Data Engineering*, vol. 7, pp. 763–785, Oct. 1995.