Learning Heuristics by Genetic Algorithms *

Rolf Drechsler

Department of Computer Science
Johann Wolfgang Goethe-University
60054 Frankfurt am Main, Germany
email: drechsler@kea.informatik.uni-frankfurt.de

Bernd Becker

Institute of Computer Science
Albert-Ludwigs-University
79110 Freiburg im Breisgau, Germany
email: becker@informatik.uni-freiburg.de

Abstract— In many applications of Computer Aided Design (CAD) of Integrated Circuits (ICs) the problems that have to be solved are NP-hard. Thus, exact algorithms are only applicable to small problem instances and many authors have presented heuristics to obtain solutions (non-optimal in general) for larger instances of these hard problems.

In this paper we present a model for Genetic Algorithms (GA) to learn heuristics starting from a given set of basic operations. The difference to other previous applications of GAs in CAD of ICs is that the GA does not solve the problem directly. Rather, it developes strategies for solving the problem. To demonstrate the efficiency of our approach experimental results for a specific problem are presented.

I. Introduction

During the last decades, the complexity of *Integrated Circuits* (ICs) has increased exponentially. In the 1970's a typical microprocessor such as the *Intel 8080* consisted of about 5,000 transistors while in 1993 Intel's state-of-the-art processor *Pentium* contains 3.1 million transistors. To handle the complexity of todays circuits the design engineers are totally dependent on *Computer Aided Design* (CAD), i.e., software tools. The capabilities and limitations of CAD tools have crucial impact on the performance and cost of the produced circuits as well as on the resources required to develop a circuit. Consequently, CAD for ICs is a very important and increasingly growing research area.

Most problems arising in CAD of ICs are very hard combinatorial optimization problems. Multiple, competing criteria have to be optimized subject to a large number of non-trivial constraints. To handle the complexity, many problems are artificially divided into a number of subproblems, which are then solved in sequence. Most of the subproblems are still both NP-hard and large, and furthermore, they are mutually dependent. The solution of a subproblem is based on estimations of the results of subsequent steps not yet performed. In other words, the cost functions used are not exact, but are based on the estimations.

Due to these reasons exact solutions can often not be determined (within reasonable time bounds). Consequently, in the last few years many authors developed heuristics to solve these problems. One promising method is the *Genetic Algorithm* (GA). GAs are often used in optimization and machine learning [7, 1]. In many applications they are superior to the classical optimization techniques, e.g. gradient-descent. Recently, GAs have succesfully been applied to several hard problems in CAD, like routing, placement, test pattern generation and logic synthesis [5].

The major drawback of these approaches is that in general they obtain good results with respect to quality of the solution, but the running times are often much larger than that of classical heuristics.

In this paper we present a new approach to apply GAs to CAD of ICs: GAs are not directly applied to the problem to be solved. Instead the GA determines a good heuristic with respect to given constraints. The designer himself can for example give upper bounds for the runtime. We develop a model for the description of the learning process.

Finally, we perform experiments with respect to a fixed problem that has intensively been studied in the past few years, i.e. we consider minimization of *Fixed Polarity Reed-Muller expressions* (FPRMs) [8]. Several methods for heuristic minimization have been presented [9, 10, 6] and also optimization by GAs has been considered [4]. (This application is only to be seen as a (well-suited) example for our application, since the problem is well understood. The results directly transfer to problems in other areas.) We show that we can determine better heuristics than all previously published. The heuristics are still much faster than the GA-approach from [4].

II. A Model for Learning Heuristics by GAs

Due to the high complexity of the design process of CAD of ICs often heuristics are used. These heuristics are developed by the designer himself. But they also often fail for specific classes of circuits. Thus it would help a lot, if the heuristics could learn from previous examples, e.g. from benchmark examples.

We assume that the problem to be solved has the following property: There is defined a non empty set of optimization procedures that can be applied to a given (non-optimal) solution in order to further improve its quality. (These procedures are called *Basic Optimization Modules*

*This work was supported in part by DFG grant Be 1176/4-2.

(BOMs) for the rest of the paper.) These BOMs are the basic modules that will be used in the following. Each heuristic is a sequence of BOMs. The goal of our approach is to determine a good (or even optimal) sequence of BOMs such that the overall results obtained by the heuristic are improved. This assumption concerning the problem is valid for all problems that can efficiently be optimized by greedy or hill-climbing heuristics.

Since the notations used for GAs are not uniquely determined we briefly introduce the notation that is used in the following. We assume a *finite set of finite strings* of fixed length over a given universe. This set is also called the *population*. For the considered problem that is to be solved by the GA an *objective function* (or *fitness function*) is given that measures the *fitness* of each element.

The main operations of GAs on strings are:

Reproduction: Copying strings according to their fitness.

Crossover: Construction of a new element x from two parents y_1 and y_2, where the first part up to a cut position i is taken from y_1 and the second part is taken from y_2.

Mutation: Construction of a new element from a parent by copying the whole element and randomly changing its value at mutation position i.

In the following we assume that the reader is familiar with the basic concepts of GAs. (For more details see [7, 1].)

For simplicity of the description we use multi-valued strings, i.e. strings that may assume various values from a fixed finite range. It does not influence the model, if the strings are considered over a two-valued alphabet. In this case an encoding must be used and method must be applied how to handle invalid solutions. (Here standard methods can be applied [7, 1].)

The set of BOMs defines the set H of all possible heuristics that are applicable to the problem to be solved in the given environment. H may include problem specific heuristics but can also include some random operators.

To each BOM $h \in H$ we associate a cost function $cost : H \rightarrow \mathbf{R}$, where $\mathbf{R}$ denotes the real valued numbers. *cost* estimates the resources that are needed for a heuristic. (If we aim at fast heuristics a heuristic h with a large value $cost(h)$ should be avoided (if possible).) We measure the fitness *fit* of a string $s = (h_1, h_2, \ldots, h_l)$ of length l (representing a heuristic composed from l BOMs) by

$$fit(s) = c_c / fit_c(s) + c_q \cdot fit_q(s),$$

where

$$fit_c(s) = \sum_{i=0}^{l-1} cost(h_i)$$

is the cost fitness of string s and

$$fit_q(s) = \sum_{i=0}^{\#\ of\ examples} quality(example_i)$$

is the quality fitness of string s. c_c and c_q are problem specific constants.

The cost fitness measures the cost for the application of the string. If this cost is relatively high the resulting heuristic will take long time. If the heuristic has a low cost fitness it will terminate quickly.

The quality fitness measures the quality of the heuristic that is represented by the string s by summing up the results for a given *set of examples*. Obviously the choice of the examples largely influences the quality of the resulting heuristic. Here the designer has to select a representative set of benchmarks. If this set cannot be determined, the GA can be run on a large set of arbitrary functions. The function *quality* measures the quality of the result with respect to the given problem. This function is typically the fitness function that is used in "normal" GAs.

The constants c_c and c_q are used to influence the primary goal of the heuristic: If c_c is set to 0 the GA will only optimize the heuristic with respect to the quality of the result, i.e. it will not care about the expenditure of the BOMs. If c_q is set to a small value the GA will determine a very fast heuristic, but the quality of the result will not be very good. Using these parameters the designer can influence the trade-off between runtime and quality and he can determine the primary goal of the GA: Should the heuristic focus on fast runtime or on good results?

III. Application of the Model

As an example we consider the minimization problem for a subclass of 2-level AND/EXOR-circuits, the so-called *Fixed Polarity Reed-Muller expressions* (FPRMs).

A. Problem Domain

An *FPRM* is an exclusive-OR of AND product terms, where each variable only appears in complemented or uncomplemented form, but not both. FPRMs are a canonical representation of Boolean functions $f : \mathbf{B}^n \rightarrow \mathbf{B}$, if the polarity for each variable is fixed. The choice of the polarity largely influences the size of the resulting FPRM, as is shown by the following example:

Example 1 Let $f : \mathbf{B}^n \rightarrow \mathbf{B}$, given by $f = \overline{x}_1 \overline{x}_2 .. \overline{x}_n$. Thus, only one term is needed, if all variables are complemented. If all variables are uncomplemented the resulting expression consists of 2^n terms, i.e. $f = 1 \oplus x_1 \oplus x_2 \oplus \ldots \oplus x_1 x_2 .. x_n$.

For the representation of Boolean functions we use a multi-level data structure, called *OFDDs*, as defined in [6]. Using the OFDD it is possible to determine very fast the number of terms of the corresponding FPRM and to change the polarity of the FPRM by manipulation of the OFDD.

We now consider the following problem, that will be solved using GAs:

> *How can we determine a heuristic that finds a polarity for a given Boolean function f such that the number of terms in the corresponding FPRM is minimized?*

Notice once more that we do *not* optimize FPRMs by GAs. Instead we optimize the heuristic that is applied to FPRM-minimization.

B. Genetic Algorithm

In this section we briefly describe the *Genetic Algorithm* (GA) that is applied to the problem given above.

Representation: In applications of GAs, often the encoding problem is one of the most difficult. In our application we use a multi-valued encoding, for which the problem can easily be formulated. Each position in a string represents an application of a heuristic. Thus a string represents a sequence of heuristics. If a string has n components at most n applications of basic elements are possible. (This upper bound is set by the designer and limits the runtime of the heuristic.) Thus, each element of the population corresponds to an n-dimensional multi-valued vector. Using this multi-valued encoding each string represents a valid solution.

In the following we consider a three-valued vector: 0 means that no operation is performed. 1 (2) represents the heuristic H1 (H2) from [6]. (We restrict to these simple alternatives, so that the concept of the algorithm becomes clear and not too many details are needed.)

Objective Function and Selection: As an *objective function* that measures the *fitness* of each element we apply the heuristics to several benchmark examples. Obviously the choice of the benchmarks largely influences the (quality of the) results. On the other hand the designer can create several different heuristics for different types of circuits, e.g. a fast but simple heuristic for very large problems or a very "time consuming" heuristic for small examples. The function *quality* was calculated on OFDDs and determines the number of terms in the FPRM. (For simplicity we only describe the case that c_c is set to 0, since cost functions for the heuristics from [6] can easily be determined.)

The selection is performed by *roulette wheel selection*, i.e. each string is chosen with a probability proportional to its fitness. Additionally, we also make use of *elitarism* [1]: A part of the best elements of the old population is included in the new one anyway. This strategy guarantees that the best element never gets lost and that a faster convergency is obtained. GA practice has shown that this method is usually advantageous. Additionally we use *steady-state reproduction*, i.e. the size of the population is constant after each iteration.

Initialization: At the beginning of each GA run an initial population is randomly generated. The fitness is assigned to each element. This fitness is calculated using the operations described in [6].

Genetic Operators: As genetic operators we used *reproduction*, *crossover* and *mutation* as described in Section II. and some slightly modified operators. (The details are left out, since they are of no relevance for the understanding of the paper.) All operators are directly applied to multi-valued strings of finite length that represent elements in the population. The parent(s) for each operation is (are) determined by the mechanisms described above. All genetic operators only generate valid solutions, if they are applied to the multi-valued strings.

```
genetic_algorithm (benchmark) {
 generate_random_population () ;
 calculate_fitness () ;
 {
   apply_operators_with_corresponding_probabilities () ;
   calculate_fitness () ;
   update_population () ;
 } while (improvement obtained) ;
 return ;
}
```

Fig. 1. Sketch of basic algorithm

Algorithm: Using the genetic operators our algorithm works as follows:

1. Initially a random population of multi-valued finite strings is generated.
2. The better half of the population is copied in each iteration without modification. Then the genetic operators *reproduction* and *crossover* are applied to another $\frac{pop}{2}$ elements. The elements are chosen according to their fitness as described above. The newly created elements are then mutated by one of the three mutation operators with a given probability. After each iteration the size of the population is constant (*steady-state reproduction*).
3. The algorithm stops if no improvement is obtained for 5 iterations.

A sketch of the algorithm is given in Fig. 1.

C. Experimental Results

In this section we present results of experiments that were carried out on the OKFDD-package presented in [3, 2] on a *SUN Sparc 1+* workstation.

We first applied our GA to a small example: We tried to determine the best heuristic for the benchmarks *co14*, *rd53*, *root* and *sao2*. The population size was chosen small, i.e. we only considered three elements. The length of the strings was fixed to five.

The results are given in Table I. *in* (*out*) denotes the number of inputs (outputs) of benchmark *name*. The results of the most powerful heuristic from [6] are given in column $H2_m^{iter}$. In [6] OFDD-based heuristics have been compared to several other approaches. There it has been shown that with this method the best results were obtained. This heuristic obtains in general good results. But for some examples it badly fails, since the greedy algorithm gets stuck very early (see benchmark *co14*). The

TABLE I
EXAMPLE WITH SMALL BENCHMARKS

name	in	out	exact	$H2_m^{iter}$	GA-heuristic
co14	14	1	14	8192	14
rd53	5	3	20	20	20
root	8	5	118	118	118
sao2	10	4	100	100	100

TABLE II
EXAMPLE WITH LARGE BENCHMARKS

name	in	out	exact	$H2_m^{iter}$	GA-heuristic
apex7	48	37	-	604	515
bc0	26	11	-	1118	1117
cps	24	109	-	293	293
chkn	29	7	-	900	900
ex5	8	63	113	120	119
gary	15	11	349	350	350
ibm	48	17	-	1220	1181
mish	94	43	-	57	54
tial	14	8	3683	4039	3683

heuristic obtained by the GA-approach is given in the last column. By a comparison with the exact solutions (column *exact*) it follows that it determined the optimal results for all examples. For the whole GA-run only 150 CPU seconds were needed.

In a second series of experiments we applied our GA to larger benchmarks from LGSynth93. For some of these benchmarks the optimal result cannot be determined. The results in comparison with the heuristics from [6] are given in Table II. (A dash symbolizes that the exact algorithm could not determine the solution due to its exponential behaviour.) For this experiment the population size was set to five elements. The length of the strings was fixed to seven. For all benchmarks the newly created heuristic obtained the same or even better results than the heuristic from [6]. The heuristic generated by the GA takes the best results on average. On the other hand it may happen that the optimal result is not determined (see benchmarks *ex5* and *gary*). The GA-run took less than 10 CPU hours. In contrast the runtime of the resulting heuristic is fast, i.e. it needs less than 1 CPU minute for the large benchmarks on average.

Finally, we applied the heuristic generated by the GA to other functions that were unknown during the optimization process. The results are given in Table III. The newly generated heuristic outperforms the heuristic from [6] on average. In only one out of six examples the GA-heuristic obtained a (slightly) worse result.

TABLE III
APPLICATION TO NEW BENCHMARKS

name	in	out	exact	$H2_m^{iter}$	GA-heuristic
cmb	16	4	-	192	136
log8mod	8	5	53	70	66
ts10	22	16	-	432	440
cordic	23	2	-	6438	6438
m181	15	9	67	71	67

IV. CONCLUSIONS

We presented a method to learn heuristics for CAD of ICs using Genetic Algorithms. The method is very general, since it applies to all problems for which greedy and hill-climbing heuristics can be applied.

We demonstrated the efficiency of our approach by an application to a problem from the area of logic synthesis. The GA-approach developed more efficient heuristics than previously known.

It is focus of current work to apply this approach to other problems in CAD of ICs.

REFERENCES

[1] L. Davis. *Handbook of Genetic Algorithms.* van Nostrand Reinhold, New York, 1991.

[2] R. Drechsler and B.Becker. Dynamic minimization of OKFDDs. In *Int'l Conf. on Comp. Design*, 1995.

[3] R. Drechsler and B. Becker. PUMA: An OKFDD-package and its implementation. In *European Design & Test Conf.*, 1995. University Booth.

[4] R. Drechsler, B. Becker, and N. Göckel. A genetic algorithm for minimization of fixed polarity reed-muller expressions. In *Int'l Conf. on Artificial Neural Networks and Genetic Algorithms*, pages 392–395, 1995.

[5] R. Drechsler, H. Esbensen, and B. Becker. Genetic algorithms in computer aided design of integrated circuits. Technical report, J.W.Goethe-University, Frankfurt, 17/94, 1994.

[6] R. Drechsler, M. Theobald, and B. Becker. Fast OFDD based minimization of fixed polarity reed-muller expressions. In *European Design Automation Conf.*, pages 2–7, 1994.

[7] D.E. Goldberg. *Genetic Algortithms in Search, Optimization & Machine Learning.* Addision-Wesley Publisher Company, Inc., 1989.

[8] I.S. Reed. A class of multiple-error-correcting codes and their decoding scheme. *IRE Trans. on Inf. Theory*, EC-3:6–12, 1954.

[9] A. Sarabi and M.A. Perkowski. Fast exact and quasi-minimal minimization of highly testable fixed-polarity and/xor canonical networks. In *Design Automation Conf.*, pages 30–35, 1992.

[10] C.C. Tsai and M. Marek-Sadowska. Efficient minimization algorithms for fixed polarity and/xor canonical networks. In *Great Lakes Symp. VLSI*, pages 76–79, 1993.