Learning Heuristics for OKFDD Minimization by Evolutionary Algorithms

Nicole Göckel Rolf Drechsler Bernd Becker

Institute of Computer Science, Albert-Ludwigs-University, 79110 Freiburg i.Br.,Germany
email: {goeckel,drechsle,becker}@informatik.uni-freiburg.de

Abstract— Ordered Kronecker Functional Decision Diagrams (OKFDDs) are a data structure for efficient representation and manipulation of Boolean functions. OKFDDs are very sensitive to the chosen variable ordering and the decomposition type list, i.e. the size may vary from linear to exponential.

In this paper we present an Evolutionary Algorithm (EA) that learns good heuristics for OKFDD minimization starting from a given set of basic operations. The difference to other previous approaches to OKFDD minimization is that the EA does not solve the problem directly. Rather, it develops strategies for solving the problem. To demonstrate the efficiency of our approach experimental results are given. The newly developed heuristics combine high quality results with reasonable time overhead.

I. Introduction

Decision Diagrams (DDs) are often used in CAD systems for efficient representation and manipulation of Boolean functions. The most popular data structure is the *Ordered Binary Decision Diagram* (OBDD) [1]. OKFDDs are a generalization of OBDDs and OFDDs as well and try to combine the advantages of both representations by allowing the use of Shannon decompositions and (positive and negative) Davio decompositions in one and the same DD [3].

OKFDDs are very sensitive to the variable ordering. In addition to the position of a variable in the ordering a so-called decomposition type has to be chosen for OKFDDs. Thus, there is a need for heuristics to choose a suitable variable ordering and decomposition type list for OKFDDs.

In the last few years many authors presented heuristics for finding good variable orderings for OBDDs. The most promising methods are based on *dynamic variable ordering* [7, 8]: OBDDs for some Boolean functions could be constructed for which all other topology oriented methods failed. In [3] it has been shown that dynamic variable ordering methods for OBDDs can also be applied to OKFDDs.

Recently, a new method based on evolutionary algorithms has been proposed for OKFDD minimization [5]. The major drawback of this approach is that in general it obtains good results with respect to quality of the solution, but the runtimes are often much larger than that of classical heuristics. Due to the high complexity of the design process of CAD of ICs often "fast" heuristics are important. Recently, a theoretical model for learning heuristics by *Genetic Algorithms* (GAs) has been presented [4]. The new aspect of this model is that the GA is not directly applied to the problem. Instead, the GA develops a good heuristic for the problem to be solved. A first application of this model to OBDD minimization has been reported in [6].

In this paper we present an *Evolutionary Algorithm* (EA)[1] based approach to learn heuristics for OKFDD minimization. Our EA learns heuristics starting from some simple basic operations that are based on dynamic reordering. The learning environment is a set of benchmark examples, that is called the *training set*. We show by experiments that our EA designs heuristics that improve the results obtained by iterated DTL-sifting [3] by about 10%. Furthermore, the runtimes of the developed heuristics are low, since the costs of the heuristic are minimized during the learning process.

II. The Learning Model

In [4] a learning model has formally been introduced. In this section we briefly review the main notations and definitions to make the paper self-contained.

It is assumed 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* (BOMs).) These BOMs are the basic modules that will be used. Each heuristic is a sequence of BOMs. The goal of the approach is to determine a good (or even optimal) sequence of BOMs such that the overall results obtained by the heuristic are improved.

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* that estimates the resources that are needed for a heuristic with respect to given examples. 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) for the underlying maximization problem by

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

The cost fitness

$$fit_c(s) = \sum_{i=0}^{\#\ of\ examples} \sum_{j=0}^{l-1} cost(h_j, example_i)$$

of string s has to be minimized and the quality fitness

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

[1] In our application we make use of a modified GA, i.e. the GA works on multi-valued strings. Thus following the standard terminology we call our implemented algorithm an EA.

ASP-DAC '97
0-89791-851-7$5.00

of string s has to be maximized. c_c and c_q are problem specific constants.

The cost fitness measures the time for the application of the string. The quality fitness measures the quality of the heuristic that is represented by the string s by summing up the results for a given *training set*.

For more details about the learning model see [4, 6].

III. Problem Domain

A. Kronecker Functional Decision Diagrams

For the definition of *Ordered Kronecker Functional Decision Diagrams* (OKFDDs) we refer to [3]. The size of OKFDDs is very sensitive to the variable ordering and the choice of the *Decomposition Type List* (DTL). We now consider the following problem that will be solved using our EA:

> *How can we determine a good heuristic to perform variable ordering and DTL choice for an OKFDD representing a given Boolean function f such that the number of nodes in the OKFDD is minimized?*

B. Dynamic Variable Ordering

It is well-known for decision diagrams that the sizes can be minimized by exchanging adjacent variables [7]. In the following we briefly describe the algorithms that are used as BOMs in our approach in the next section:

Sifting (S) [8]: By the sifting algorithm, the variables are sorted into decreasing order based on the number of nodes at each level and then each variable is traversed through the OKFDD in order to locate its local optimal position while all other variables and the DTL remain fixed.

Siftlight (L): Siftlight is a restricted form of sifting that does not allow the algorithm to do any hill climbing, i.e. the variables are directly located in the next minimum.

DTL-sifting (D) [3]: By DTL-sifting the variables are sorted into decreasing order based on the number of nodes at each level. Then each variable traverses for all three different decomposition types the OKFDD analogously to *sifting*.

Exact (E): Perform the exact minimization algorithm for OKFDDs for only three adjacent levels (=window). A window is chosen for optimization, if the sum over all nodes in these levels is maximal.

Inversion (I): The variable ordering is inverted.

IV. Evolutionary Algorithm

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

A. Representation

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 BOM. Thus a string represents a sequence of heuristics. If a string has n components at most n applications of BOMs 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.

In the following we consider six-valued vectors over the alphabet $\{S, L, D, E, I, N\}$: $S(L, D, E, I)$ represents sifting (siftlight, DTL-sifting, exact, inversion). N (no operation) means that no operation is performed. The operation N takes no ressources and thus the costs of the resulting heuristic can be minimized. (We restrict to these alternatives, since they have shown to work very well in our application.)

B. Objective Function and Selection

As an *objective function* that measures the *fitness* of each element we apply the heuristics to benchmark *training sets*. Obviously the choice of the benchmarks largely influences 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 problem instances or a relative "time consuming" heuristic for small examples. For the calculation of function *quality* the OKFDD has been constructed using the initial variable ordering and Shannon decompostion in each node. Then the heuristic represented by the considered element is performed and the number of nodes has been counted. The function is then given by the equation $quality = 1/nodes$, since the number of nodes has to minimized. Function *cost* measures the computation time that is used to evaluate an element, i.e. the time that is needed to reorder the OKFDD, in 10^{-2} CPU seconds.

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 *steady-state-reproduction* [2].

C. Operators

As genetic operators we used *reproduction*, *crossover* and *mutation* and some slightly modified operators. All operators are directly applied to six-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.

D. Algorithm

Using the genetic operators our algorithm works as follows:

1. The initial population of size 10 is generated randomly and the length of the strings is set to 20.
2. Then $\frac{pop}{2}$ elements are generated by the genetic operators that are applied with a corresponding probability. The newly created elements are then mutated with a probability of 15%. After each iteration the size of the population is constant.
3. If no improvement is obtained for 50 generations the algorithm stops.

V. Experimental Results

In this section we present results of experiments that were carried out on a *SUN Sparc 20* workstation. All runtimes *time* are given in CPU seconds. The benchmark functions are taken from LGSynth91. The best results are given in bold in the following.

In a first series of experiments we developed a heuristic for OKFDD minimization on a small *training set* that is composed of only five functions. The learning time for all heuristics takes about 12-14 CPU hours.

The results are given in Table I, where *in* (*out*) denotes the number of inputs (outputs) of the corresponding benchmark and *size* gives the number of nodes. DTL denotes the results after DTL-sifting iteratively applied until no further improvement could be obtained. (Thus, DTL-sifting implicitly makes use of a more powerful "do-until"-operator. Nevertheless our results demonstrate that our

TABLE I
TRAINING SET

name	*in*	*out*	DTL		EA1: $c_c = 1$		EA2:$c_c = \frac{1}{10}$		EA3:$c_c = \frac{1}{100}$		EA4: $c_c = 0$	
			size	*time*	*size*	*time*	*size*	*time*	*size*	*time*	*size*	*time*
add6	12	7	24	0.3	68	< 0.1	25	0.2	**23**	0.3	**23**	0.6
alu2	10	6	138	1.0	157	0.1	127	0.9	120	1.2	**119**	2.4
frg1	28	3	**70**	1.2	79	< 0.1	75	0.9	74	1.3	72	1.6
x6dn	39	5	217	1.7	244	0.1	203	2.2	**196**	2.7	**196**	4.4
Z5xp1	7	10	**28**	0.2	41	0.2	**28**	0.4	**28**	0.4	**28**	1.1

TABLE II
APPLICATION TO NEW BENCHMARKS

name	*in*	*out*	*DTL*		*EA1*		*EA2*		*EA3*		*EA4*	
			size	*time*	*size*	*time*	*size*	*time*	*size*	*time*	*size*	*time*
addm4	9	8	126	0.6	163	0.1	130	0.8	126	1.0	**125**	2.1
apex2	39	2	310	147.8	553	22.6	274	153.0	**243**	198.1	273	178.3
apex7	49	37	264	2.3	300	0.3	225	2.7	230	3.1	**220**	4.8
bc0	26	11	431	2.3	522	0.3	426	2.5	**423**	3.1	**423**	4.5
bcd	26	38	565	6.1	574	0.8	568	6.2	569	8.5	**561**	11.2
cm85a	11	3	35	< 0.1	35	< 0.1	26	0.1	**20**	0.1	**20**	0.2
chkn	29	7	260	16.8	324	1.8	246	11.7	**228**	15.3	246	17.2
cps	24	109	573	10.7	726	1.1	**545**	7.0	563	9.3	548	11.2
ex5	8	63	**234**	0.6	241	0.1	304	0.9	**234**	1.1	281	2.9
ex7	16	5	75	0.8	77	0.1	62	1.0	**60**	1.3	**60**	1.8
gary	15	11	286	0.9	297	0.1	282	1.1	**280**	1.5	292	2.3
in7	26	10	**64**	0.9	83	0.2	73	1.3	76	1.7	75	1.9
m181	15	9	**53**	0.6	54	0.1	55	0.8	**53**	1.0	56	1.9
mlp4	8	8	**107**	0.5	134	0.2	108	1.0	108	1.3	108	2.7
pdc	16	40	**572**	1.6	605	0.8	649	5.8	639	7.9	663	10.2
rd73	7	3	30	0.1	30	< 0.1	**21**	0.1	**21**	0.2	**21**	0.6
risc	9	8	**56**	0.1	65	< 0.1	**56**	0.1	**56**	0.2	**56**	0.6
sqn	7	3	**47**	0.1	55	< 0.1	51	0.1	51	0.2	51	0.4
t1	21	23	**103**	0.1	114	0.1	126	0.4	119	0.6	120	0.7
tial	8	31	550	6.4	613	0.8	489	4.8	473	9.7	**454**	10.0
ts10	22	16	**131**	2.0	145	2.4	160	7.2	160	10.3	**131**	10.9
vg2	25	8	186	0.5	193	0.1	95	0.9	94	1.2	**76**	1.8

TABLE III
EXTENDED TRAINING SET

name	*in*	*out*	*DTL*		*EA1*		*EA2*		*EA3*		*EA4*	
			size	*time*	*size*	*time*	*size*	*time*	*size*	*time*	*size*	*time*
add6	12	7	24	0.3	41	< 0.1	24	0.2	**23**	0.2	**23**	0.6
alu2	10	6	138	1.0	149	0.1	137	0.5	**122**	1.1	124	2.2
frg1	28	3	**70**	1.2	94	< 0.1	77	1.1	75	1.1	72	2.4
x6dn	39	5	217	1.7	232	0.1	215	1.8	213	3.2	**203**	5.7
Z5xp1	7	10	**28**	0.2	29	0.1	**28**	0.2	**28**	0.3	**28**	1.2
pdc	16	40	572	1.6	574	0.2	**564**	4.5	**564**	6.2	565	12.9
sqn	7	3	**47**	0.2	56	< 0.1	**47**	0.1	**47**	0.1	**47**	0.5
t1	21	23	103	0.1	111	< 0.1	110	0.2	103	0.5	**102**	1.1

methods obtains better results with "weaker" operators.) The four rightmost columns show the results after applying the newly developed heuristics that are learned by our EA for varying parameter settings of c_c. For each EA c_q is fixed by a problem specific constant factor and c_c is chosen in the range from 0 to 1; the exact settings of c_c are given in Table I in the top row. The larger c_c is chosen the larger is the influence of the timing aspect in our EA.

The results obtained by EA1, where runtime is the main optimization goal, in all cases are worse than DTL-sifting, but the corresponding runtimes are much better, i.e. never larger than 0.2 CPU seconds. If the influence of the timing aspect is decreased, the quality of the results gets higher. EA2 produces better results than DTL-sifting on average and the runtimes are also often faster. The heuristic that is developed by EA4 does not consider any timing optimization. As can be seen EA4 determines the best OKFDD sizes on average on the training set. The runtimes are still in an acceptable range, i.e. it takes only a few seconds.

In a next series of experiments we applied the developed heuristics to new benchmarks that were not included in the training set, i.e. functions that were unknown during the optimization process of the EA. The results are given in Table II. As can easily be seen the timing behaviour of the heuristics is similar to the first experiment. The application of DTL-sifting obtains best solutions for about 40% of the considered Boolean functions. For EA3 and EA4 the best solutions are obtained for nearly 50% even though the resulting heuristics were not trained on these functions. It is important to notice that the learned heuristic EA4 is never more than 20% worse than DTL-sifting. On the other hand DTL-sifting gets stuck very early in some cases and gets results that are more than 50% worse (see e.g. *vg2*).

In a next experiment we extended the training set by three further examples that obtained unsatisfying results, i.e. functions *pdc,t1* and *sqn* from Table II. For these functions DTL-sifting created smaller OKFDDs. The goal is to improve the heuristics by adapting them to the *extended* training set. The results of this experiment are given in Table III. As can easily be seen the results (especially on the new examples) get much better than for DTL-sifting.

Finally, the new heuristics that were learned on the extended training set are applied to the remaining benchmarks (that were again unknown during the optimization). The results are given for EA3 and EA4 in Table IV. (EA3 and EA4 are chosen because of their performance concerning the quality of the results.) The experiments show that EA4 is now only in one case, i.e. *chkn*, worse than DTL-sifting (and the difference in this case is only one node). For **all** other benchmarks EA4 is (much) better or equal to DTL-sifting. The runtimes are still reasonable, i.e. less than 20 CPU seconds for most examples.

All in all, the experiments have shown that our EA is able to design a heuristic that is more powerful than a "hand-designed" heuristic. The larger the training set is chosen the higher is the average gain in the quality of the results. This also enables the option to add new examples to the EA run and in this way to dynamically adapt the heuristic to new problem instances.

TABLE IV
APPLICATION TO NEW BENCHMARKS II

name	*DTL*		*EA3*		*EA4*	
	size	*time*	*size*	*time*	*size*	*time*
addm4	126	0.6	126	0.9	**125**	2.4
apex2	310	148.4	323	170.9	**273**	270.1
apex7	264	2.3	259	2.9	**187**	7.4
bc0	**431**	2.3	**431**	2.3	**431**	7.3
bcd	565	6.1	**561**	7.1	**561**	13.4
cm85a	35	< 0.1	35	0.1	**33**	0.3
chkn	**260**	16.8	315	13.6	261	24.7
cps	573	10.7	**556**	7.9	560	18.9
ex5	**234**	0.6	**234**	0.9	**234**	2.6
ex7	**75**	0.8	**75**	1.1	**75**	2.2
gary	286	0.9	292	1.3	**279**	3.3
in7	**64**	0.9	78	1.5	**64**	3.0
m181	**53**	0.6	**53**	1.0	**53**	2.5
mlp4	**107**	0.5	108	1.1	**107**	3.1
rd73	30	0.1	30	0.2	**21**	0.7
risc	**56**	0.1	**56**	0.2	**56**	0.6
tial	550	6.4	533	5.7	**471**	13.4
ts10	**131**	2.0	**131**	8.1	**131**	13.2
vg2	186	0.5	187	1.1	**184**	3.6

VI. Conclusions

We presented an *Evolutionary Algorithm* (EA) that learns strategies for OKFDD minimization. The EA depends on a parameter set that influences the quality and the runtimes of the resulting heuristic.

The EA learns the heuristic on a *training set* composed of examples of Boolean functions. The resulting heuristic works very well with respect to quality and costs on these examples. The application to new benchmarks that were unknown during the learning process demonstrates the efficiency of the new heuristic. Furthermore, the integration of new examples into the learning set improves the quality of the newly developed heuristic. This technique enables the designer to create problem specific heuristics, e.g. for a specific class of circuits.

Finally, it should be mentioned that our methods can be direcly applied to more restricted classes of DDs, like OBDDs and OFDDs.

References

[1] R.E. Bryant. Graph - based algorithms for Boolean function manipulation. *IEEE Trans. on Comp.*, 8:677–691, 1986.

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

[3] R. Drechsler and B. Becker. Dynamic minimization of OKFDDs. In *Int'l Conf. on Comp. Design*, pages 602–607, 1995.

[4] R. Drechsler and B. Becker. Learning heuristics by genetic algorithms. In *ASP Design Automation Conf.*, pages 349–352, 1995.

[5] R. Drechsler, B. Becker, and N. Göckel. Minimization of OKFDDs by genetic algorithms. In *Int'l Symposium on Soft Computing*, pages B:271–B:277, 1996.

[6] R. Drechsler, N. Göckel, and B. Becker. *Learning Heuristics for OBDD Minimization by Evolutionary Algorithms.* LNCS 1141. Parallel Problem Solving from Nature, 1996.

[7] M. Fujita, Y. Matsunga, and T. Kakuda. On variable ordering of binary decision diagrams for the application of multi-level synthesis. In *European Conf. on Design Automation*, pages 50–54, 1991.

[8] R. Rudell. Dynamic variable ordering for ordered binary decision diagrams. In *Int'l Conf. on CAD*, pages 42–47, 1993.