Disjoint Sum of Product Minimization by Evolutionary Algorithms

Nicole Drechsler   Mario Hilgemeier
Görschwin Fey   Rolf Drechsler

Institute of Computer Science
University of Bremen
28359 Bremen, Germany
{nd,mh,fey,rd}@informatik.uni-bremen.de

Abstract

Recently, an approach has been presented to minimize Disjoint Sum-of-Products (DSOPs) based on Binary Decision Diagrams (BDDs). Due to the symbolic representation of cubes for large problem instances, the method is orders of magnitude faster than previous enumerative techniques. But the quality of the approach largely depends on the variable ordering of the underlying BDD.

This paper presents an Evolutionary Algorithm (EA) to optimize the DSOP representation of a given Boolean function. The EA is used to find an optimized variable ordering for the BDD representation. Then the DSOP is derived from the optimized BDD using structural and symbolic techniques. Experiments are performed to adjust the parameters of the EA. Experimental results are given to demonstrate the efficiency of the approach.

1 Introduction

A DSOP is a representation of a Boolean function as a sum of disjoint cubes. DSOPs are used in several applications in the area of CAD, e.g. the calculation of spectra of Boolean functions [7, 8, 22] or as a starting point for the minimization of *Exclusive-Or-Sum-Of-Products* (ESOPs) [17, 20]. In [9, 21] some techniques for minimization of DSOPs have been introduced. They are working on explicit representations of the cubes and therefore are only applicable to small instances of the problem.

BDDs in general are an efficient data structure for representation and manipulation of Boolean functions. They are well-known and widely used in logic synthesis [16, 10] and formal verification [2, 15] of integrated circuits. BDDs are well-suited for applications in the area of logic synthesis, because the cubes in the ON-set of a Boolean function are implicitly represented in this data structure.

A hybrid approach for the minimization of DSOPs relying on BDDs in combination with structural methods has recently been introduced in [11]. It has been shown that BDDs are applicable to the problem of DSOP minimization. Given a BDD of a Boolean function, the DSOP can easily be constructed: each one-path, i.e. a path from the root to the terminal 1 vertex, corresponds to a cube in the DSOP, and moreover, different one-paths lead to disjoint cubes. For the construction of the BDD the variables of the Boolean function are considered in a fixed order. The permutation of the variables largely influences the number of one-paths in the BDD and thus the number of cubes in the corresponding DSOP. Additionally, the importance of choosing a "good" variable order to get a small DSOP has theoretically been supported.

In this context EAs have been shown to be a promising approach, i.e. they work very well for BDD minimization and other variants of permutation problems (see e.g. [6, 14]).

In this paper we present an EA for determining a good ordering for a BDD. The BDD is optimized in such a way that the corresponding DSOP is minimized. The parameters of the EA are studied in detail. Experimental results are given that show improvement over the best known heuristics.

The paper is organized as follows: Section 2 briefly introduces the necessary notion of BDDs and one-paths and shows how the DSOP representation is related to BDDs. In Section 3 the EA is discussed in detail. Experimental results are given in Section 4. Finally, conclusions are drawn.

2 Preliminaries

2.1 Binary Decision Diagrams

A BDD is a directed acyclic graph $G_f = (V, E)$ that represents a Boolean function $f : \mathbf{B}^n \longrightarrow \mathbf{B}^m$. The Shannon decomposition $g = x_i g_{x_i} + \overline{x}_i g_{\overline{x}_i}$ is carried out in each internal node v labeled with *label*$(v) = x_i$ of the graph, therefore v has the two successors *then*(v) and *else*(v). The leaves are labeled with 0 or 1 and correspond to the constant Boolean functions. The root node *root*(G_f) corresponds to the function f. In the following, *BDD* refers to a reduced ordered BDD (as defined in [1]) and the size of a BDD is given by the number of nodes.

Definition 1 *A* one-path *in a BDD $G_f = (V, E)$ is a path*

$$p = (v_0, ..., v_{l-1}, v_l),$$

$$v_i \in V, (v_i, v_{i+1}) \in E$$

with $v_0 = root(G_f)$ and label$(v_l) = 1$. p has length $l + 1$.

$\mathcal{P}_1(G_f)$ denotes the number of all different one-paths in the BDD G_f.

2.2 BDD and DSOP

Consider a BDD G_f representing the Boolean function $f(x_1, ..., x_n)$. A one-path $p = (v_0, ..., v_l)$ of length $l + 1$ in G_f corresponds to an $(n - l)$-dimensional cube that is a subset of $ON(f)$[1]. The cube is described by:

$$m_p = \bigcap_{i=0}^{l-1} l_i, \text{where}$$

$$l_i = \begin{cases} \overline{label(v_i)}, & \text{if } v_{i+1} = else(v_i) \\ label(v_i), & \text{if } v_{i+1} = then(v_i) \end{cases}$$

Two paths p_1 and p_2 in a BDD are different iff they differ in at least one edge. Since all paths originate from $root(G_f)$, there is a node v where the paths separate. Let $label(v) = x_i$. Therefore one of the cubes includes x_i, the other $\overline{x}_i$. Hence, the cubes m_{p_1} and m_{p_2} are disjoint.

Now the DSOP can easily be built by summing up all cubes corresponding to the one-paths.

Remark 1 Let G_f be a BDD of $f(x_1, ..., x_n)$ and $\mathcal{M}_1$ be the set of one-paths in G_f. Then G_f represents the DSOP

$$\sum_{p \in \mathcal{M}_1} m_p,$$

where m_p is the cube given above.

From this it is clear that the number of cubes in the DSOP represented by G_f is equal to $\mathcal{P}_1(G_f)$. Thus, as opposed to the usual goal of minimizing the number of nodes in a BDD, here the number of one-paths is minimized.

Known techniques to minimize the number of nodes can be used to minimize the number of paths by changing the objective function. One such technique is sifting [19]: A variable is chosen and moved to any position of the variable order based on exchange of adjacent variables. Then it is fixed at the best position (i.e. where the smallest BDD results), afterwards another variable is chosen. No variable is chosen twice during this process.

For the evaluation of our fitness function, in the following the improved path-minimization algorithm based upon sifting from [11] is used. This algorithm employs structural techniques to reduce the number of cubes in the DSOP.

3 Evolutionary Algorithm

In this section we describe the *Evolutionary Algorithm* (EA) that is applied to the problem given above. In Sections 3.1 to 3.4 the submodules of our EA (like the evolutionary operators) and the overall structure of the algorithm are described. Finally, the detailed choices for the parameter settings are discussed.

The DSOP of a Boolean function that is represented by a BDD directly depends on the chosen variable ordering. Finding a good or even optimal variable ordering is a permutation problem that is optimized by an evolutionary approach.

[1]$ON(f)$ is the ON-set of the Boolean function f, i.e. the variable assignments that evaluate f to value 1.

3.1 Representation

A population is a set of individuals that represent solutions of the given optimization problem. In this application an individual represents the ordering position of each variable:

We use an integer encoding to represent the ordering of the variables. (A binary encoding would require special repair operators to avoid the creation of invalid solutions similar to the TSP, see e.g. [23].) Each integer vector of length n represents a permutation of the variables and thus it is a feasible ordering.

The length of the strings is given by the number of variables n, because for each variable the ordering position has to be stored.

3.2 Objective Function and Selection

The objective function assigns to each individual a fitness that measures the quality of the variable ordering. First the BDD is constructed using the variable ordering given by the individual, then the number of *reduced* one-paths are counted [11].

The selection is performed by *roulette wheel selection* and we also make use of *steady-state reproduction* [4]: The best individuals of the old population are included in the new one of equal size. This strategy guarantees that the best element never gets lost and a fast convergence is obtained. (EA practice has shown that this method is usually advantageous.)

3.3 Evolutionary Operators

Different types of crossover operators and mutations for permutation problems are used. For the crossover operators two parents are selected by the method described above. For more details about the evolutionary operators we refer to [13, 3, 18, 5, 14].

Notice that a simple exchange of the parts between the cut positions (as often applied to binary encoded EA problems) is not possible, since this would often produce invalid solutions.

PMX [13]: Choose two cut positions randomly in the parent elements. Construct the children by choosing the part between the cut positions from one parent and preserve the *absolute* position and order of as many variables as possible from the second parent.

OX [3]: Choose two cut positions randomly in the parent elements. Construct the children by choosing the part between the cut position from one parent and preserve the *relative* position and order of as many variables as possible from the second parent.

CX [18]: Choose a single position i in the parent elements at random. Copy the values of this position into the children at exactly the same position: $child1[i] = parent1[i]$ and $child2[i] = parent2[i]$. Then, position j in *parent1* is determined, such that $parent1[j] = parent2[i]$. Set $i := j$ and continue the procedure as described above, until the new position j has already been copied in the children elements. Then a “cycle” has been found and the remaining positions in *child1* (*child2*) are taken from *parent1* (*parent2*).

MERGE [14]: It produces the first child in the following way. Alternating between the parents, MERGE takes one variable index from each parent (in the order they appear in the parents) until the double permutation length is reached. After doing this, MERGE checks from left to right if an index has been used already. If this is the case, that index number is removed. The second child is produced by exchanging even and odd positions in the index list.

INV: The inversion operator INV inverts the order of the index list for a randomly chosen part of the chromosome. It produces one child from each parent.

Additionally, three different *mutation operators* are used:

Mutation (MUT): Select one individual and choose two different positions at random. Swap the values of these two positions.

2-time Mutation (MUT2): Perform MUT two times on the same parent.

Mutation with neighbor (MUT-N): Perform MUT at two adjacent positions.

3.4 Flow of the Algorithm

Using the operators introduced above our EA works as follows:

- Initially a random population of permutations (variable ordering) is generated.
- The crossover operators described in Section 3.3 are applied to selected elements and create $|pop|/2$ offspring. The parent elements are chosen according to their fitness (see Section 3.2). The offspring are then mutated by the mutation operators with a given probability.
- The fitness of the offspring is calculated as described in Section 3.2. The worse half of the population is then replaced by new individuals.
- The last two steps are iterated until the termination criterion holds.

Experiments have shown that it is sufficient to consider 24 individuals in a population. For larger population sizes the execution times are getting “too high”.

To get a good parameter setting of the EA, several experiments and operator studies have been performed. Due to the page limitation we can not discuss these experiments in detail. The following weighting of the evolutionary operators has been selected with respect to the given test suite: PMX, OX, CX, MERGE, and INV are performed with probabilities 1%, 43%, 1%, 11%, and 43%, respectively. The mutation probabilities are all set to 1%.

The termination criterion depends on the best fitness value of the considered population: The algorithm stops if no improvement is obtained for $50 * \log_{10} best\_fitness$ iterations. The idea is: if the considered fitness value is higher, more optimization potential can be expected.

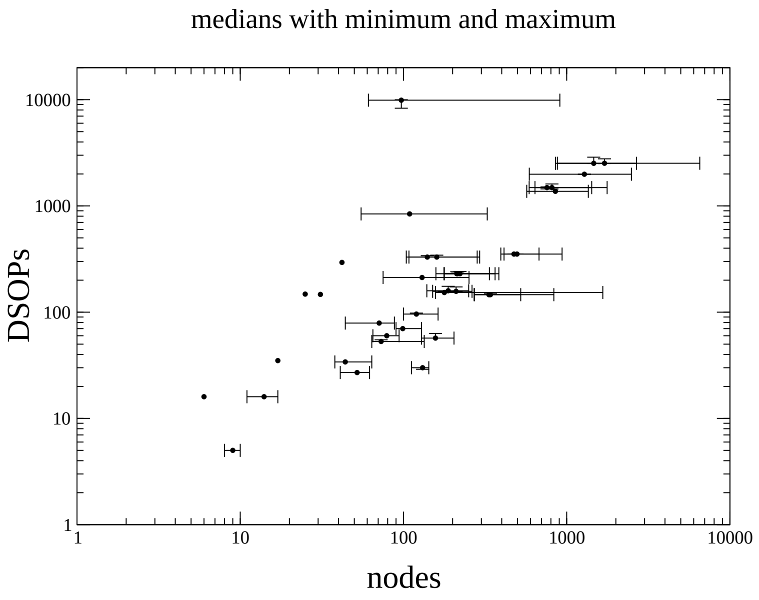

Figure 1: Representation of benchmark suite

4 Experimental Results

Experiments were carried out on a Pentium II system at 450 MHz with 256 MB of physical memory. The machine was running under Linux. Control programs for the experiments were written in Python. The EA was based on the C++ library for evolutionary algorithms GAME [12] (version 2.43).

The experimental results show the quality of the proposed evolutionary method. The EA is applied to several benchmark functions, most of them taken from LGSynth93. The experiments are summarized in the following tables.

In Table 1 the EA for all considered benchmarks is analyzed. To get an impression of the (local) minimum number of cubes per function, the EA is applied 25 times to the 37 benchmarks in the test suite. Each time a randomly chosen seed for the random number generator was used. In column **hybrid** the number of cubes resulting from the method proposed in [11] for each function is given. Columns **EA** summarize the results from the EA proposed in this paper. **min.** and **max.** show the minimal and maximal DSOP of all test runs, respectively. In column **median** the median values and in column Δ the differences between *hybrid* and **min.** are shown.

As can be seen, no DSOP number calculated by the EA in column **min.** is worse than the number of DSOPs found previously [11]. In more than 60% of the test cases the minima in column **hybrid** could be improved. E.g. for benchmark *cordic* the size of the DSOP is improved by more than 50%.

Fortunately, it can be observed that the differences between the minimum and maximum are relatively small. I.e. the performance of the proposed evolutionary method is very stable and nearly independent of the random number generator.

Table 1: Descriptive statistics and comparison of EA minima to hybrid algorithm

	DSOP				
	hybrid		*EA*		Δ
function		**min.**	**median**	**max.**	
s1196	2861	2504	2516	2877	357
s1238	2861	2504	2521	2776	357
s1488	369	352	352	353	17
s1494	369	352	352	353	17
s208	53	53	53	55	0
s27	16	16	16	16	0
s298	70	70	70	70	0
s344	330	330	330	344	0
s349	330	330	330	340	0
s382	238	230	230	239	8
s386	61	57	57	63	4
s400	238	230	230	241	8
s444	243	230	230	240	13
s510	170	153	153	155	17
s526	162	156	157	173	6
s526n	162	156	159	175	6
s641	1700	1444	1486	1609	256
s713	1700	1445	1486	1511	256
s820	155	146	146	146	9
s832	155	146	146	149	9
alu4	1545	1372	1372	1372	173
b12	60	60	60	60	0
clip	262	212	212	212	50
inc	66	27	27	27	39
majority	5	5	5	5	0
misex1	34	34	34	34	0
misex2	30	29	30	30	1
rd53	35	35	35	35	0
rd73	147	147	147	147	0
rd84	294	294	294	294	0
sao2	96	96	96	98	0
t481	1009	841	841	841	168
xor5	16	16	16	16	0
5xp1	82	79	79	79	3
9sym	148	148	148	148	0
cordic	19763	8311	9885	9978	11452
misex3	2255	1973	1986	1993	282

This small spread in the results can also be observed in Figure 1, where it is demonstrated by small or invisible range bars of the median DSOP values (ordinate). The figure also shows the size range of the benchmarks in the test suite.

Table 2: Comparison of different operator weightings

	median			*minimum*		
function	$\mathbf{EA}_P$	**EA**	Δ	$\mathbf{EA}_P$	**EA**	Δ
s1196	2534	2516	18	2504	2504	0
s1238	2544	2521	23	2504	2504	0
s1488	352	352	0	352	352	0
s1494	352	352	0	352	352	0
s208	53	53	0	53	53	0
s27	16	16	0	16	16	0
s298	70	70	0	70	70	0
s344	330	330	0	330	330	0
s349	330	330	0	330	330	0
s382	230	230	0	230	230	0
s386	57	57	0	57	57	0
s400	230	230	0	230	230	0
s444	236	230	6	230	230	0
s510	153	153	0	153	153	0
s526	160	157	3	156	156	0
s526n	162	159	3	156	156	0
s641	1496	1486	10	1444	1444	0
s713	1497	1486	11	1458	1445	13
s820	146	146	0	146	146	0
s832	146	146	0	146	146	0
alu4	1372	1372	0	1372	1372	0
b12	60	60	0	60	60	0
clip	212	212	0	212	212	0
inc	27	27	0	27	27	0
majority	5	5	0	5	5	0
misex1	34	34	0	34	34	0
misex2	30	30	0	29	29	0
rd53	35	35	0	35	35	0
rd73	147	147	0	147	147	0
rd84	294	294	0	294	294	0
sao2	97	96	1	96	96	0
t481	841	841	0	841	841	0
xor5	16	16	0	16	16	0
5xp1	79	79	0	79	79	0
9sym	148	148	0	148	148	0
cordic	9932	9885	47	9882	8311	1571
misex3	1986	1986	0	1973	1973	0

In Table 2 a comparison of different operator weightings is given. The operator weighting used for the EA proposed in here (column **EA**) is shown in parallel with the results using a different kind of operator weighting (column $\mathbf{EA}_P$). The problem specific weightings from [14] were used as reference (PMX: 98%, Inversion: 1%, MUT1: 7%, MUT2: 7%). The comparison of the results shows the advantage of the proposed setting for DSOP minimization.

Finally, the runtimes of the EA are shortly discussed. In principle, compared to specialized heuristics, the runtimes of EAs are relatively large. In this application they are in general smaller than 200 CPU seconds. And for small benchmark functions, the EA converges in a few CPU seconds. The quality of the results and the relatively short runtimes show the efficiency of the chosen termination criterion.

5 Conclusions

An approach based on Evolutionary Algorithms to minimize the DSOP representation of a Boolean function was presented. The experimental results show the quality of the proposed evolutionary method. For more than 60% of the test cases the results from the specialized heuristic could be improved. Even the runtimes of the EA are in an acceptable range. The experiments underlined the robustness of the approach.

References

[1] R.E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Trans. on Comp.*, 35(8):677–691, 1986.

[2] R.E. Bryant. Binary decision diagrams and beyond: Enabling techniques for formal verification. In *Int'l Conf. on CAD*, pages 236–243, 1995.

[3] L. Davis. Applying adaptive algorithms to epistatic domains. In *Proceedings of IJCAI*, pages 162–164, 1985.

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

[5] R. Drechsler. *Evolutionary Algorithms for VLSI CAD.* Kluwer Academic Publisher, 1998.

[6] R. Drechsler, B. Becker, and N. Göckel. A genetic algorithm for variable ordering of OBDDs. *IEE Proceedings*, 143(6):364–368, 1996.

[7] B.J. Falkowski. Calculation of rademacher-walsh spectral coefficients for systems of completely and incompletely specified boolean functions. In *IEEE Proceedings on Circuits*, pages 1698–1701, 1993.

[8] B.J. Falkowski and C.-H. Chang. Paired haar spectra computation through operations on disjoint cubes. In *IEEE Proceedings on Circuits, Devices and Systems*, pages 117–123, 1999.

[9] B.J. Falkowski, I. Schäfer, and C.-H. Chang. An effective computer algorithm for the calculation of disjoint cube representation of boolean functions. In *Midwest Symposium on Circuits and Systems*, pages 1308–1311, 1993.

[10] F. Ferrandi, A. Macii, E. Macii, M. Poncino, R. Scarsi, and F. Somenzi. Symbolic algorithms for layout-oriented synthesis of pass transistor logic circuits. In *Int'l Conf. on CAD*, pages 235–241, 1998.

[11] G. Fey and R. Drechsler. A hybrid approach combining symbolic and structural techniques for disjoint SOP minimization. In *Workshop on Synthesis And System Integration of Mixed Information technologies (SASIMI)*, pages 54–60, 2003.

[12] N. Göckel, R. Drechsler, and B. Becker. GAME: A software environment for using genetic algorithms in circuit design. In *Applications of Computer Systems*, pages 240–247, 1997.

[13] D.E. Goldberg and R. Lingle. Alleles, loci, and the traveling salesman problem. In *Int'l Conference on Genetic Algorithms*, pages 154–159, 1985.

[14] M. Hilgemeier, N. Drechsler, and R. Drechsler. Minimizing the number of one-paths in BDDs by an evolutionary algorithm. In *Congress on Evolutionary Computation*, pages 1724–1731, 2003.

[15] Th. Kropf. *Introduction to Formal Hardware Verification.* Springer, 1999.

[16] Y.-T. Lai, S. Sastry, and M. Pedram. Boolean matching using binary decision diagrams with applications to logic synthesis and verification. In *Int'l Conf. on CAD*, pages 452–458, 1992.

[17] A. Mishchenko and M. Perkowski. Fast heuristic minimization of exclusive-sums-of-products. In *Int'l Workshop on Applications of the Reed-Muller Expansion in Circuit Design*, pages 242–250, 2001.

[18] I.M. Oliver, D.J. Smith, and J.R.C. Holland. A study of permutation crossover operators on the traveling salesman problem. In *Int'l Conference on Genetic Algorithms*, pages 224–230, 1987.

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

[20] T. Sasao. EXMIN2: A simplification algorithm for Exclusive-OR-Sum-of products expressions for multiple-valued-input two-valued-output functions. *IEEE Trans. on CAD*, 12:621–632, 1993.

[21] L. Shivakumaraiah and M.Thornton. Computation of disjoint cube representations using a maximal binate variable heuristic. In *Southeastern Symposium on System Theory*, pages 417–421, 2002.

[22] M. Thornton, R. Drechsler, and D.M. Miller. *Spectral Techniques in VLSI CAD.* Kluwer Academic Publisher, 2001.

[23] D. Whitley, T. Starkweather, and D. Fuquay. Scheduling problems and traveling salesman: The genetic edge recombination operator. In *Int'l Conference on Genetic Algorithms*, pages 133–140, 1989.