FORCE: A Fast and Easy-To-Implement Variable-Ordering Heuristic

Fadi A. Aloul, Igor L. Markov, Karem A. Sakallah
Department of Electrical Engineering and Computer Science
University of Michigan
{faloul, imarkov, karem}@eecs.umich.edu

ABSTRACT

The MINCE heuristic for variable-ordering [1] can successfully reduce the size of BDDs and accelerate SAT-solving. Applications to reachability analysis have also been successful [12]. The main drawback of MINCE is its implementation complexity - the authors used a pre-existing min-cut placer [6] that is several times larger than any existing SAT solver. Tweaking MINCE is difficult.

In this work we propose a replacement heuristic, FORCE which is easy to implement from scratch and tweak. It is dramatically faster than MINCE in practice. While FORCE may produce seemingly inferior variable orderings, the difference with MINCE orderings does not affect subsequent SAT-solving.

Categories and Subject Descriptors

I.1 [**Symbolic and Algebraic Manipulation**]: Algorithms.

General Terms

Algorithms, Performance, Experimentation, Verification.

Keywords

BDDs, SAT, CNF, backtrack search, variable order, pre-processing, hypergraph, partitioning, placement.

1 INTRODUCTION

Algorithms for electronic design automation (EDA) [15, 22], including those for synthesis and verification, require efficient manipulation of Boolean functions. Boolean satisfiability (SAT) [14, 19] solvers and binary decision diagrams (BDDs) [5] have traditionally been used with such applications, but their worst-case complexity remains exponential and can hardly be improved.

A key observation is that Boolean functions arising in EDA applications possess useful structural properties, e.g., related variables in satisfiability typically participate in the same clauses. Uses of problem structure are known to improve the efficiency of the SAT and BDD algorithms. For example, Prasad et al. [17] theoretically showed that combinational circuits with small cuts yield easy instances of automatic test pattern generation (ATPG), which are essentially SAT instances. BDDs with smaller cuts tend to have fewer edges and vertices, speeding up BDD manipulations [1, 3].

Based on these observations, the MINCE heuristic [1] reorders Boolean variables to place ‘‘connected’’ variables close to each other. The ordering relies on high-performance hypergraph partitioning and placement to reduces the cut in the problem. MINCE is executed as a preprocessing step and does not require the modification of the application code. MINCE can accelerate SAT solving and BDD manipulation, and reduce BDD memory consumption. The use of the external black-box tool Capo [6], however, complicates the process of integrating MINCE with other applications and slows down the variable ordering process.

In this paper, we propose a new domain-independent algorithm, FORCE, for ordering variables of CNF formulas and BDDs. FORCE does not rely on external black-box tools and can be implemented of less than 500 lines of code in C. It can be easily integrated into any application or used as a simple pre-processing tool. FORCE is orders-of-magnitude faster than MINCE and shows competitive performance in SAT and BDD applications. FORCE can be used as an alternative to MINCE for applications that require flexibility and multiple variable-ordering calls.

The remainder of the paper is organized as follows. Section 2 covers the necessary background. It reviews SAT and BDDs, motivates the use of hypergraph partitioning, and describes MINCE. It also mentions the main ideas behind force-directed placement. The FORCE heuristic is described in Section 3. In Section 4, we present our experimental results. The conclusions and future work are described in Section 5.

2 BACKGROUND AND PREVIOUS WORK

Boolean satisfiability solvers and BDD operations are popular in in formal verification, logic synthesis and other EDA fields.

Boolean Satisfiability. This problem involves finding an assignment to a set of binary variables that satisfies a set of constraints in conjunctive normal form (CNF), i.e., a conjunction of clauses, each of which is a disjunction of literals. A literal is either a variable or its negation. An example of a CNF formula is: $(x_1)(\bar{x}_2 + x_4)(\bar{x}_3 + x_2)$.

Complete SAT search algorithms [14, 19] are often based on the Davis-Logemann-Loveland (DLL) approach [9], i.e., a depth-first search in the decision tree over the problem variables. Reordering variable decisions can accelerate this algorithm, and decision heuristics can be classified as static [1] (pre-processing) or dynamic [14, 19]. For example, the GRASP SAT solver [19] is typically used with the dynamic heuristic DLIS, which selects the literal that appears in the maximum number of unresolved clauses.

Binary Decision Diagrams. BDDs [5] provide a canonnical and compact representation of Boolean functions. They are directed acyclic graphs produced by compacting decision trees of Boolean functions. The size of BDDs can still be exponential in the number of variables in the problem and a good variable ordering is essential to keep the size of BDDs manageable. Proposed variable ordering heuristics can also classified as being static [11, 13] and dynamic [16, 18]. Sifting [16, 18] is one of the most popular dynamic techniques, but has a high runtime overhead.

Since variable ordering is not a problem in itself, it is important that the search for good orderings do not consume more resources than the implied speed-up to particular applications.

GLSVLSI'03, April 28-29, 2003, Washington, DC, USA.

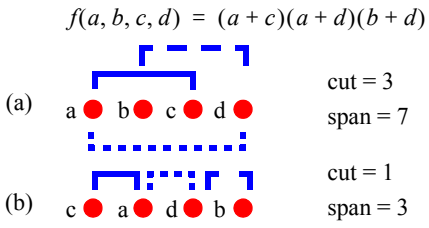


Figure 1: Example of a CNF to hypergraph conversion using (a) original variable ordering (b) improved variable ordering

Problem Partitioning. A key idea behind existing algorithms for variable ordering is to order "connected" variables next to each other. E.g., in Boolean satisfiability, decisions on variables that share common constraints are often made within a short time interval. If decisions follow the order of variables, ordering connected variables next to each other may speed satisfiability solvers. In terms of BDDs, partitioning the problem, yields a BDD with a smaller cut which implies fewer edges. Since a vertex is a source of two edges, this leads to a smaller number of BDD vertices.

MINCE [1] analyzes the structure of Boolean formulae to find a good variable ordering for SAT and BDDs. It views a CNF formula as a hypergraph whose vertices correspond to the formula's variables and whose hyperedges correspond to its clauses. One-dimensional min-cut linear placement then produces an ordering vertices. This ordering is translated back into an ordering of variables, and the original formula is reordered (see example in Figure 1). For a given variable ordering, we define the *span* of a clause as the difference between the smallest and greatest variables in the clause. The *cut* for variable x is the number of clauses that include variables with indices both *less than* and *greater than* $(x + 0.5)$. The cut of a formula is the max cut over all variables.

The use of min-cut linear placement leads to smaller "half-perimeter wire-length", which is equivalent to a smaller average clause span in CNF formulas. The average clause span is related to the average cut as follows.

Lemma 2.1 [1] Consider a CNF formula with $|V|$ variables and $|C|$ clauses. The average variable cut is equal to the product of the average clause span and $|C|/(|V|-1)$.

Since the total number of variables and clauses are fixed, the average clause span is proportional to the average cut. Therefore, the two objectives can be minimized using the same algorithm.

To solve the linear placement problem, MINCE uses the hypergraph placer Capo [6] based on recursive min-cut bisection to optimize cuts and average clause span. MINCE has best- and worst-case complexity of $\Theta(N(\log N)^2)$, where N is the size of the input.

Force-directed placement. The MINCE heuristic capitalizes on 30 years of progress in min-cut placement, but other placement algorithms may also be useful for SAT and BDD. Wood and Rutenbar used spectral placement techniques for BDD ordering [23], but those algorithms are slow, difficult to implement and cannot accomodate important constraints. Since our work seeks *easy-to-implement* algorithms, we focus on force-directed placement [8]. A key idea behind such algorithms is to analogize interconnect between placeable objects with springs that exert forces according to the Hooke's law. Starting from an arbitrary, e.g., random, initial solution, we compute the forces acting on each object and displace objects in the direction of the forces. Such an iteration is very fast, and is typically repeated until a given objective function stops decreasing. The main difficulty in force-directed methods is to prevent overlaps and enforce slot assignments. In terms of variable ordering this means that variable locals are integer and no two variables share the same location. Sometimes overlaps are removed by introducing "repelling forces" between objects, in addition to "attracting forces" that correspond to interconnect. We observe that force-directed placement is greatly simplified if performed in one dimension because slot constraints can be enforced by sorting. In practice force-directed VLSI placement is not as good as min-cut placement in minimizing average net length (clause span) or cut-based objectives. However, it is often used in commercial placement products because it is fast, easy to implement and amenable to additional objectives and constraints.

```
Procedure: FORCE {
1  randomly generate an initial order of vertices;
2  repeat limit times or until total span stops decreasing
3    for each hyperedge e ∈ E
4      compute center of gravity of e;
5    for each vertex v ∈ V
6      compute tentative new location of v based on
centers of gravity of hyperedges;
7    sort tentative vertex locations;
8    assign integer indices to the vertices;
9  }
```

Figure 2: The FORCE heuristic.

3 THE FORCE ALGORITHM

The proposed heuristic FORCE performs one-dimensional placement of a given hypergraph, and outputs a new vertex order that tends to put connected vertices close to each other. The pseudo-code of the algorithm is shown in Figure 2. If no initial ordering is given, FORCE randomly generates an initial ordering. The remaining part of the algorithm is a loop that performs iterations. In each iteration, the algorithm begins by traversing all hyperedges and computing the *center of gravity* (COG) of each hyperedge e:

$$COG(e) = \left(\sum_{v \in e} l_v\right)/|e| \qquad (3.1)$$

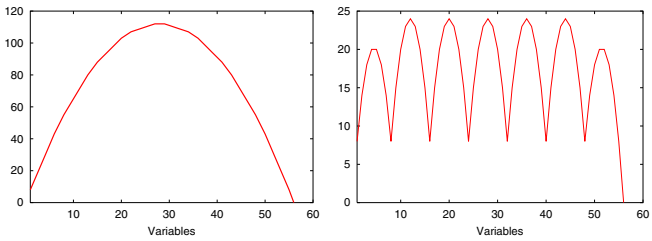
where l_v and $|e|$ denote the location of vertex v under the given linear ordering and the number of vertices connected to hyperedge e, respectively. Next, the algorithm traverses all vertices and computes their tentative new locations (not necessarily integers!) using the following heuristic. Denoting with l_v' the new tentative location of vertex v and with E_v the hyperedges connected to vertex v:

$$l_v' = \left(\sum_{e \in E_v} COG(e)\right)/|E_v| \qquad (3.2)$$

This averages centers of gravity of all hyperedge conected to vertex x. The iteration is finalized by sorting tentative locations and assigning integer indices to them. Iterations continue until a given mtric of ordering, e.g. total span, stops improving. We additionally bound the number of iterations by $c\log|V|$, where c is a constant and $|V|$ is the total number of vertices. Each traversal takes worst-

Table 1: Chaff runtimes (in seconds) for CNF-SAT instances using various decision heuristics.

Instance	#V	#C	VSIDS	MINCE			FORCE			Average Variable Cut		
			Solve	Order	Solve	Total	Order	Solve	Total	Orig	MINCE	FORCE
hole10	110	561	874	0.84	81.6	82.5	0.01	85.5	85.5	201	30	30
hole11	132	738	1000	0.86	919	920	0.01	738	738	263	35	35
fpga10_8	120	448	281	0.71	46.8	47.5	0.03	68.9	68.9	117	31	35
fpga10_9	135	549	476	0.87	344	345	0.03	5.09	5.12	141	37	40
chnl9_11	198	1012	205	1.37	18.2	19.5	0.01	16	16	181	35	30
chnl9_12	216	1212	519	1.96	35	37	0.01	30.4	30.4	215	35	35
xor1_32	94	250	1000	0.48	174	175	0.01	116	116	90	33	42
xor1_36	106	282	803	0.57	1000	1000	0.02	229	229	106	33	47
Urq3_1	43	334	1000	0.5	663.5	664	0.01	485	485	220	96	119
Urq3_9	37	236	25.3	0.37	6.93	7.3	0.01	3.97	3.98	162	65	81
2pipe_1_ooo	834	7026	7.16	31.7	1.18	32.8	1.28	5.19	6.47	2517	791	1048
2pipe_2_ooo	925	8212	10.55	32.4	2.04	34.4	1.37	2.17	3.54	3054	895	1241
2pipe	861	6695	7.57	22.4	0.94	23.4	0.6	1.14	1.74	2187	734	930
grout3.3-8	912	8356	2.43	22.6	10.9	33.5	0.33	143.4	143.7	2082	259	380
grout3.3-10	1056	10.8k	19.4	52.1	7.06	59.2	0.41	525.8	526.2	2707	304	448
Total			**6230**	**170**	**3311**	**3481**	**4**	**2456**	**2460**	**14243**	**3413**	**4541**



(a) (b)

Figure 3: The "cutwidth profiles" for the Hole-7 SAT instance (56 variables, 204 clauses). The vertical axis here represents the number of clauses (hyperedges) that cross a given variable. The profile in (a) is for the original ordering and has cutwidth over 100. The profile in (b) is for FORCE ordering and much lower max-cut (less than 25).

case linear time in $|V| + |C|$, and sorting takes $\Theta(|V|\log|V|)$ time. Hence, FORCE has a worst case performance of $O((|C| + |V|\log|V|)\log|V|)$ (we assume the average degree of hyperedges and the average degree of vertices are limited by a constant. This compares favorably with many SAT and BDD algorithms that have exponential worst-case performance. Since best-case time for some SAT-solving algorithms (e.g., DLL [9]) is linear in the size of the input, FORCE is more competitive on hard instances. In practice, the computational overhead of FORCE is often so low, that it can compete with linear-time algorithms.

An additional post-processing step can "slide a small window", e.g., of size 4, along the final ordering returned by FORCE and exhaustively enumerate all permutations of variables in the window. The variables in the window are re-ordered in the best possible way, and a new window is considered. Such algorithms have been explored in VLSI placement [7].

Figure 3 shows cut-profiles of a pigeon-hole CNF instance using the original and FORCE variable orderings (the latter is similar to MINCE). The better ordering reduces cut and reveals structure.

4 EXPERIMENTAL RESULTS

Improvements obtained using FORCE are shown in two experiments - (i) faster SAT solving, and (ii) faster and more memory-efficient BDD operations.

We used the SAT solver Chaff, and the BDD package CUDD [20]. The SAT experimental results are given for instances from the pigeon-hole [10], FPGA routing (*fpga* and *chnl*) [15], xor-chains, randomized Urquhart [21], global routing (*grout*) [2], and microprocessor verification benchmarks (*pipe*) [22]. The BDD experimental results are given for the circsuit consistency functions of the ISCAS89 [4] circuit benchmarks, expressed in CNF format. We used a Linux workstation with a 333 Mhz Pentium-II processor. Runtime and memory limits were 1,000 seconds and 500 MB, resp.

We compared the performance of Chaff using three decision heuristics: static MINCE [1], static FORCE, and the dynamic variable state independent decaying sum (VSIDS) [14]. VSIDS selects the variable that appears in the highest number of clauses and gives some priority to variables that appear in recent conflict-induced clauses. Random restarts was disabled in Chaff.

Table 1 shows instance sizes, Chaff runtimes, ordering runtimes, and the average variable cut for three orderings. We observe:

1. For the pigeon-hole and FPGA routing instances, both MINCE and FORCE yield the best search runtimes.
2. For the xor-chain and Urquhart instances, FORCE wins.
3. For the global routing instances, VSIDS leads to the best search runtimes. These instances have large average variable cuts.
4. The results are mixed for the microprocessor verification instances. The use of MINCE leads to the best search runtimes, but FORCE's search runtimes are better than VSIDS.
5. MINCE and FORCE always significantly reduce variable cuts.
6. Ordering runtimes are correlated with the size of the instance. FORCE is orders-of-magnitude faster than MINCE.

Our approach is more effective on *highly*-structured problems, such as the pigeon-hole or FPGA routing instances, which consist of *multiple partitions*. On these problems, MINCE and FORCE capture problem structure, speeding up SAT solvers.

The dynamic decision heuristic VSIDS outperforms static heuristics on *general* structured EDA instances, e.g., the global routing instances. This is partly because the VSIDS decision heuristic accounts for the added conflict-induced clauses, which may increase cutwidth and eliminate the advantage of static ordering.

While the worst-case complexity of Chaff is exponential, FORCE runs in near-linear time, and the implementation constant (in the leading term of asymptotic complexity) is much smaller than that of MINCE. FORCE can be integrated into dynamic ordering heuristics, so as to account for conflict-induced clauses and can also be called periodically within backtrack SAT solvers.

Table 2: Statistics for constructing the BDDs of the ISCAS89 CNF Benchmarks [4].
(nodes = maximum number of BDD nodes seen during the construction of the BDD, in thousands)

Circuit	Without sifting						With sifting						Order Time		Average Var Cut		
	Original		MINCE		FORCE		Original		MINCE		FORCE		MINCE	FORCE	Orig	MINCE	FORCE
	time	nodes	time	nodes	time	nodes	time	nodes	time	nodes	time	nodes					
s27	0.08	0.2	0.09	0.08	0.08	0.07	0.07	0.2	0.08	0.08	0.07	0.07	0.23	0	11	4	6
s208.1	time-out		0.18	1.2	1.34	22	24	8	0.19	1.2	0.42	3	0.65	0.02	104	17	23
s298	mem-out		1.34	13.5	2.97	31	563	294	4.54	4	13.6	9.2	0.69	0.06	157	28	36
s344	mem-out		1.15	12.2	16	145	mem-out		6.17	10	28	24	1.01	0.04	137	17	28
s349	mem-out		0.8	10.7	4.81	31	5509	789	8.01	6.4	23	17.7	1.06	0.08	149	18	28
s382	mem-out		1.12	13.7	3.09	22	273	118	10	11	15.3	9	0.85	0.06	176	25	33
s400	mem-out		1.14	14.4	3.03	22	330	68	5.41	7.5	10.2	7.2	1.21	0.04	182	26	32
s386	mem-out		12.8	83	3.06	26	403	131	38	20	17.9	9.6	1.09	0.06	172	55	60
s444	mem-out		0.65	6.8	17	85	mem-out		4.05	4.2	25	15	1.39	0.05	192	26	39
s420	mem-out		0.99	7.2	13	86	1175	142	7.94	6.7	46	20	1.06	0.13	182	19	30
s510	mem-out		13	91	151	1.1K	mem-out		60	31	139	26	3.28	0.14	224	69	79
s526	mem-out		8.63	38	9.13	60	mem-out		17	12	33	15	1.34	0.11	271	42	49
s526n	mem-out		4.15	25	9.62	63	mem-out		15.5	12	38.6	19	1.55	0.06	262	41	48
s641	mem-out		30	125	time-out		time-out		150	67	time-out		1.7	0.12	190	23	39
s713	mem-out		26.7	108	909	2.8K	mem-out		113	59	time-out		1.79	0.22	216	27	41
s832	mem-out		time-out		time-out		time-out		mem-out		time-out		5.64	0.13	432	156	162
s838	mem-out		5.73	31	881	7.8K	time-out		104	27	743	105	2.03	0.4	366	29	42
s953	mem-out		369	1.1K	time-out		mem-out		time-out		mem-out		2.87	0.28	369	85	104
s838.1	mem-out		4.56	19	140	608	mem-out		55	15.7	342	32	2.62	0.62	419	29	47
Total	**0.08**	**0.2**	**482**	**1700**	**2164**	**12901**	**8277**	**1550**	**599**	**295**	**1475**	**312**	**32**	**2.6**	**4211**	**734**	**926**
#Built	**1**		**18**		**16**		**8**		**17**		**15**						

FORCE is also applicable as a variable ordering heuristic for BDDs and leads to significantly smaller BDDs. Table 2 shows runtimes in seconds, average cut and the maximum number of nodes (in thousands) seen during the construction of the BDDs, where clauses are processed in order of decreasing indices of their smallest literals. We report all results with and without sifting. FORCE and MINCE clearly outperform the original ordering and dynamic sifting (which is also very slow). While MINCE outperforms FORCE in many instances, the overall runtimes are often better when FORCE is used because the two produce very similar solutions and BDD/SAT applications are not very sensitive to the difference..

5 CONCLUSIONS

We proposed the FORCE heuristic for ordering variables for SAT-solvers and BDD algorithms. It is much faster and easier to implement than the existing MINCE heuristic, but yields comparable improvements in runtime of SAT solvers and BDD algorithms.

6 REFERENCES

[1] F. Aloul, I. Markov, and K. Sakallah, "Faster SAT and Smaller BDDs via Common Function Structure," *ICCAD*, 443-448, 2001.

[2] F. Aloul, A. Ramani, I. Markov, and K. Sakallah, "Generic ILP versus Specialized 0-1 ILP: an Update," *ICCAD*, 450-457, 2002.

[3] C. Berman, "Circuit Width, Register Allocation, and Ordered Binary Decision Diagrams," *IEEE Trans. on CAD*, 10(8), 1059-1066, 1991.

[4] F. Brglez, D. Bryan, and K. Kozminski, "Combinational problems of sequential benchmark circuits," *in Proc. of ISCAS,* 1929-1934*,* 1989.

[5] R. Bryant, "Graph-based algorithms for Boolean function manipulation," *in IEEE Trans. on Computers,* 35(8), 677-691, 1986.

[6] A. Caldwell, A. Kahng, and I. Markov, "Can Recursive Bisection Produce Routable Placements?" *in Proc. of DAC*, 477-482, 2000.

[7] A. Caldwell, A. Kahng, and I. Markov, "Optimal Partitioners and End-case Placers for Standard-Cell Layout", *in IEEE Trans. on CAD*, 19(11), 1304-1314, 2000.

[8] C. Cheng and E. Kuh, "Module Placement Based on Resistive Network Optimization," *in IEEE Trans. on CAD*, 3(7), 218-225, 1984.

[9] M. Davis, G. Logemann, and D. Loveland, "A Machine Program for Theorem Proving," *in Comm. of the ACM*, 5(7), 394-397, 1962.

[10] DIMACS Challenge benchmarks in *ftp://Dimacs.rutgers.EDU/pub/challenge/sat/benchmarks/cnf*.

[11] M. Fujita, H. Fujisawa, and N. Kawato, "Evaluation and Improvements of Boolean Comparison Method Based on Binary Decision Diagrams," *in Proc. of ICCAD*, 2-5, 1988.

[12] H. Jin, A. Kuehlman, and F. Somenzi, "Fine-grain Conjunction Scheduling for Symbolic Reachability Analysis", *in Tools and Algorithms for Construction and Analysis of Systems*, 312-326, 2002.

[13] S. Malik, A. Wang, R. Brayton, and A. Sangiovanni-Vincentelli, "Logic Verification using Binary Decision Diagrams in a Logic Synthesis Environment," *in Proc. of ICCAD*, 6-9, 1988.

[14] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: Engineering an Efficient SAT Solver," *DAC*, 530-535, 2001.

[15] G. Nam, F. Aloul, K. Sakallah, and R. Rutenbar, "A Comparative Study of Two Boolean Formulations of FPGA Detailed Routing Constraints," *Int'l Symposium on Physical Design*, 222-227, 2001.

[16] S. Panda and F. Somenzi, "Who are the variables in your neighborhood," *in Proc. of ICCAD*, 74-77, 1995.

[17] M. Prasad, P. Chong, and K. Keutzer, "Why is ATPG easy?" *in Proc. of DAC*, 22-28, 1999.

[18] R. Rudell, "Dynamic variable ordering for ordered binary decision diagrams," *in Proc. of ICCAD*, 42-47, 1993.

[19] J. Silva and K. Sakallah, "GRASP: A Search Algorithm for Propositional Satisfiability," *IEEE Trans. on Comp.,* 48(5), 506-521, 1999.

[20] F. Somenzi, "Colorado University Decision Diagram package (CUDD)," 1997 `http://vlsi.colorado.edu/~fabio/CUDD`

[21] A. Urquhart, "Hard Examples for Resolution," in *Journal of the ACM,* 34(1), 209-219, 1987.

[22] M. Velev and R. Bryant, "Superscalar Processor Verification Using Reductions of the Logic of Equality with Uninterpreted Functions to Propositional Logic," *In Proc. Conf. on Correct Hardware Design and Verification Methods*, LNCS 1703, 37-53, 1999.

[23] R. G. Wood and R. A. Rutenbar, "FPGA Routing and Routability Estimation Via Boolean Satisfiability," *in IEEE Trans. on VLSI*, 6(2), 222-231, 1998.