MINCE: A Static Global Variable-Ordering for SAT and BDD

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

Abstract

Many popular algorithms that work with Boolean functions are dramatically dependent on the order of variables in input representations of Boolean functions. Such algorithms include satisfiability (SAT) solvers that are critical in formal verification and Binary Decision Diagrams (BDDs) manipulation algorithms that are increasingly popular in synthesis and verification. Finding better variable-orderings is a well-recognized problem in each of those contexts. Currently, all leading-edge variable-ordering algorithms are dynamic in the sense that they are invoked many times in the course of the "host" algorithm that solves SAT or manipulates BDDs. Examples include the DLIS ordering for SAT solvers and variable-sifting during BDD manipulations. In this work we propose a universal variable ordering MINCE (MIN Cut Etc.) that preprocesses a given Boolean formula in CNF. MINCE is completely independent from target algorithms and outperforms both DLIS for SAT and variable sifting for BDDs. We argue that MINCE tends to capture structural properties of Boolean functions arising from real-world applications.

1 Introduction

Algorithms that efficiently manipulate Boolean functions arising in real-world applications are becoming increasingly popular in several areas of computer-aided design and verification. In this work we focus on two classes of these algorithms: complete Boolean satisfiability (SAT) solvers [7, 15, 19, 24, 27, 31] and algorithms for manipulating Binary Decision Diagrams (BDDs) [4, 8, 17]. A generic complete SAT solver must correctly determine whether a given Boolean function represented in the *conjunctive normal form* (CNF) evaluates to *false* for all input combinations. Aside from its pivotal role in complexity theory [11], the SAT problem has been widely applied in electronic design automation. Such applications include ATPG [16, 28], formal verification [2], timing verification [25], routing of field-programmable gate arrays [20], among others. While no exact polynomial time algorithms are known for the general case, many exact algorithms [15, 19, 24, 27, 31] manage to complete very quickly for problems of practical interest. Such algorithms are available in the public domain and are typically based on "elementary steps" that consider one variable at a time (e.g., branch-and-bound algorithms need to select the next variable for branching.) Previously published results [19, 24, 27, 31], as well as our empirical data, clearly imply that the order of these steps critically affects the runtime of leading edge SAT algorithms. This order of steps depends on the order of variables used to represent the input function, but can also be controlled dynamically based on the results of previous steps.

BDDs [4, 8] are commonly used to implicitly represent large solution spaces in combinatorial problems that arise in synthesis and verification. A BDD is a directed acyclic graph constructed in such a way that its directed paths represent combinatorial objects of interest (such as subsets, clauses, minterms, etc.). An exponential compression rate is achieved by BDDs whose number of paths is exponential in the number of vertices and edges (graph size). BDDs can be transformed by algorithms that visit all vertices and edges of the directed graph in some order and therefore take linear time in the "current" size of the graph. However, when new BDDs are created, some of these algorithms tend to significantly increase the number of vertices, potentially leading to exponential memory and runtime requirements. Several BDD ordering techniques have been proposed to overcome this problem. These techniques include static [10, 18] and dynamic approaches [21, 23]. Just as for SAT solvers, the order of "elementary steps" is critically important. This order can either be chosen *statically*, i.e. by pre-processing the input formula, or *dynamically*, based on the outcome of previous steps during the search process.

A reliable and fast variable-ordering heuristic for a given application can dramatically affect its competitiveness and is often considered an important part of implementation. For example, the leading-edge SAT solver GRASP [24] is typically used with the dynamic variable-ordering heuristic DLCS (select the *variable* that appears in the maximum number of unresolved clauses) or DLIS (select the *literal* that appears in the maximum number of unresolved clauses), and the reknowned CUDD package [26] for BDD manipulation incorporates the dynamic variable-sifting heuristic which is applied many times in the course of BDD transformations. Variable sifting is affected by the initial order, but can also be completely turned off.

We noticed that, for some benchmark CNF formulae in Table II and Table IV (such as hole-9 and par16-2-c), turning off sifting for BDD manipulations and turning off DLIS in SAT resulted in significantly smaller runtimes. For BDDs, this also led to memory savings, especially for circuit benchmarks from the ISCAS89 set. In other words, the order of variables produced when encoding problems into CNF formulae was superior to the best known dynamic variable orderings. Note that such "static" variable orderings are easier to work with because they do not require modifying the source code of the host algorithm. In particular, the same variable ordering implementation can be used for SAT solvers and BDD manipulations if it, indeed, improves both classes of algorithms. However, for any given application, even if superior "static" variable orderings exist, they may be overlooked by specific encoding procedures. Therefore, we propose a domain-independent algorithm to automatically find good "static" variable-ordering that capture global properties of a given CNF formula.

The remainder of the paper is structured as follows. Section 2 motivates our reliance on hypergraph partitioning. In Section 2.1, we describe how hypergraph partitioning is performed. Section 3 describes applications to SAT and BDDs and provides experimental evidence of the effectiveness of partitioning-based variable ordering. The paper ends with conclusions and future work in Section 4.

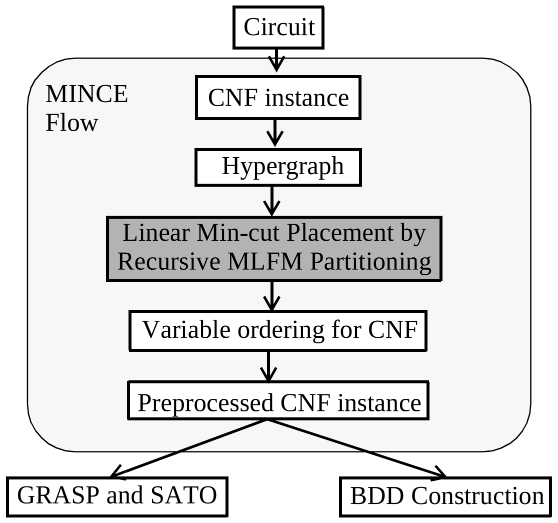
Circuit
MINCE Flow
CNF instance
Hypergraph
Linear Min-cut Placement by Recursive MLFM Partitioning
Variable ordering for CNF
Preprocessed CNF instance
GRASP and SATO
BDD Construction

Figure 1: The MINCE heuristic based on Multi-Level Fiduccia-Mattheyses (MLFM) partitioning [5, 6, 14]

2 Problem Partitioning

We first observe that Boolean functions arising in many applications represent spacial, logical or causal dependencies/connections among variables. Therefore, processing "connected" variables together seems intuitively justified. For example, if a large SAT instance is not satisfiable because of a small group of inconsistent variables, the variables in this group must be "connected" by some clauses. If we can partition all variables into, say, two largely independent groups, then such a function is likely to be represented by a BDD with a small cut, i.e. there will be relatively few edges between these two groups. BDDs with many small cuts tend to have fewer edges, and therefore fewer vertices (since every vertex is a source of exactly two edges). This intuition suggests that we interpret CNF formulae as hypergraphs by representing variables by vertices and clauses by edges. Two vertices share an edge if the two corresponding variables share a clause in the formula. Applying balanced min-cut partitioning to such hypergraphs separates the original CNF formula into relatively independent subformulae. Ordering the variables in each part together would be a step towards ordering "connected" variables next to each other, as advocated earlier. Once the first partitioning is performed, the parts can be partitioned recursively. This process can provide a complete variable ordering. We note that cuts of CNF formulae have been studied in [22], and instances having small cuts were theoretically shown to be "easy" for SAT. Our work seeks constructive and efficient ways to amplify the "easiness" of CNF instances with small cuts by finding good variable orderings. Additionally, Berman [1] related the size of BDDs to circuit width.

2.1. Recursive Bisection and Hypergraph Placement

Recursive min-cut bisection of hypergraphs has been intensively studied in the context of VLSI placement for at least 30 years. In particular, the recursive bisection procedure described earlier for CNF formulae corresponds to the *linear placement problem* [13], where hypergraph vertices are placed in one, rather than in two, dimensions. It is well-known that placement by recursive bisection leads to small "half-perimeter wire-length" that translates back to small average clause span in CNF formulae. Here we define the *span* of a clause with respect to a variable ordering as *the difference between the greatest and the smallest number of variables in this clause* (so that the span exactly correspond to the half-perimeter wirelength of a hyperedge). We can also define the *i-th cut* with respect to a given ordering as the number of clauses including variables with numbers both less than and greater than *i+0.5*.

Observation: Given a variable ordering, the total clause span equals the sum of all cuts. Average clause span is proportional to the average cut, and the coefficient is approximately equal the clause-to-variable ratio of the CNF formula.

Since recursive bisection appears to optimize both cuts and average clause spans, we will use the leading-edge hypergraph placer Capo [5] based on recursive mincut bisection [6, 14]. The Capo placer incorporates a number of improvements to classical recursive bisection which reduce the total clause span (and thus the average cut). Such techniques include bisection with high balance tolerance and adaptive cut-line selection, which allows better freedom in partition sizes in order to improve the cut. The underlying multi-level hypergraph partitioner MLPart [6] outperforms the well-known hMetis [14] and executes two independent starts for each partitioning, followed by one V-cycle of the better solution.

We propose the following heuristic that orders variables in CNF formulae (see Figure 1). An initial CNF formulae (that may or may originate from circuit or other applications) is converted into a hypergraph. Min-cut linear placement is applied to the hypergraph using the Capo placer [5] to produce an ordering of hypergraph vertices. This ordering is translated back into an ordering of variables in the original CNF instance. The original CNF formulae is preprocessed by applying this new order. After that, the new CNF formulae can be used as input to an arbitrary SAT solver or to construct a BDD representation of the boolean function it represents. The results produced by SAT solvers or BDD manipulations can be translated back into the original variable at any time.

Note that this approach does not require modifications in SAT solvers, BDD manipulation software or the Capo placer. We call this heuristic MINCE (MIN-Cut, Etc.) and implemented it by chaining publicly available software with PERL scripts.

To enable black-box usage of publicly available software (Capo), we ignore polarities of literals in CNF formulae. We note that the oriented version of min-cut bisection has been extensively studied in the context of timing-driven placement. In particular, finding a good unoriented cut implies an oriented cut which is at least as good. Vice versa, in most real-world examples, near-optimal oriented cuts can be found by unoriented partitioning.

Wood and Rutenbar have already used linear hypergraph placement as a variable ordering technique for BDD minimization in 1998 [29]. However, they relied on spectral methods which require converting hyperedges to edges and then minimizing quadratic edge length rather than the half-perimeter (linear) edge length. Spectral placement methods used in [29] do not appear to have direct connections to cut minimization. As of 2001, spectral methods for partitioning and placement are practically abandoned due to their unacceptable runtime for large-scale instances and poor solution quality as measured by half-perimeter edge length. This can be contrasted with min-cut placement that is among the fastest known approaches, provides good solutions and is obviously related to cut minimization.

On the empirical side, our results with BDD minimization presented below show that our static variable ordering heuristic MINCE outperforms variable-sifting in both runtime and memory. According to [12], variable sifting is the best known heuristic for BDD minimization. From this, we conclude that our proposed tech-

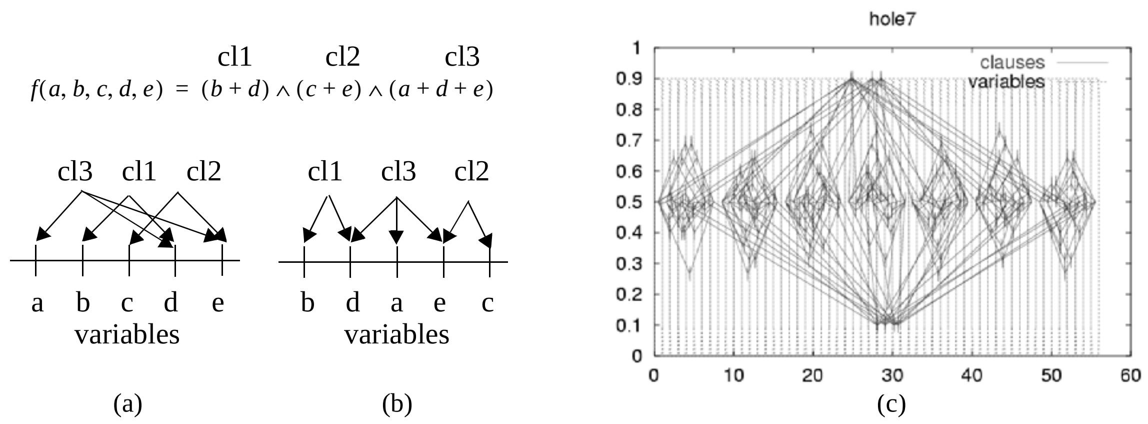

Figure 2: (a) Example of default vertex-ordering (b) Example of improved vertex-ordering
(c) Sample hypergraph representing the structure of the *Hole-7* instance

niques outperform all other published approaches to BDD minimization. Furthermore, applications that require several BDD transformations or solving similar SAT problems are expected to run faster, since the ordering is generated once and used for all runs.

2.2. Illustration

Figure 2 illustrates the difference between good and a bad variable ordering for a CNF formula. We use the Capo placer to find an ordering of vertices, i.e. variables, that produces a small total (equivalent average) clause span. Figure 2(b) shows a sample ordering returned by Capo for the example described. The total span of all clauses in this CNF formula is reduced from 8 to 4 by this better variable ordering. In addition, the number of edges crossing each variable, which we will refer to as the *variable cut*, is reduced. In the given example, the original problem had a maximum *variable cut,* implied by variable c, of 3 which was eventually reduced to 1 using the new Capo ordering.

In general, *structured* problems such as the *hole-n* benchmark can be easily divided by Capo into several partitions. Figure 2(c) shows an example of the hypergraph generated by Capo for the *hole-7* instance. Clearly, the problem's structure consists of several partitions. The initial variable ordering implied an average *clause span* and *variable cut* equal to 74 and 20, respectively. In comparison, the new variable-ordering, reflected an average *clause span* and *variable cut* equal to 17 and 4.7, respectively. We conjecture that such variable ordering, which groups each partition, should yield better SAT or BDD runtime and memory results.

Similar techniques and intuitions apply in related contexts. For example, one can apply MINCE to DNF formulae rather than CNF formulae. In this and related cases, one starts with a description of a Boolean function that is sparse, i.e., the description "connects" very few groups of variables (by clauses, minterms, in terms of circuit connectivity, etc). Recursive partitioning results in a variable ordering where "connected" variables are close to each other. Since "connections" between variables often imply logical dependencies, min-cut orderings allow SAT solvers and BDD engines to track fewer variables beyond their neighborhoods.

3 Application of MinCut to SAT & BDDs

In the following section, we present experimental evidence for the improvements obtained by the new static variable ordering. We decided to use GRASP as our SAT solver [24] and CUDD as our BDD solver [26]. All experiments were conducted on a Pentium-II 333 MHz, running Linux with 512 Mb. of physical memory. In terms of CNF problems, we used the DIMACS benchmarks [9] in addition to the *n-queens* problem. We also used a flat version of the ISCAS89 circuit benchmarks [3] expressed in CNF. For all experiments, the CPU time and memory limits were set to 10,000 seconds and 500 Mb, respectively.

SAT Experiment: Table I and Table II summarize the runtime results of running the *MINCE* variable ordering versus using the dynamic *MSTS*, *MSOS*, *DLCS*, *DLIS*, or the static *fixed* variable ordering [24]. Also, the tables list the time needed to order the problem using Capo. All runtimes are presented in units of seconds. "#I" denotes the number of instances solved by each decision heuristic and "S/U" describes the type of the problem (S for satisfiable and U for unsatisfiable). The average *variable cut* is also included using the original and the new *MINCE* variable orderings.

As the data clearly illustrate, deciding on closely-connected variables leads to a reduction in search time. Specifically, since "connected" variables are ordered next to each other, this approach allows the solver to quickly identify and avoid unpromising partial solutions. In other words, instead of deciding on variables from separate partitions, we can focus on a single partition. The chances of identifying a solution is more likely since the variables are strongly connected. This approach appears to be effective on *structured* problems, such as the *hole-n* or the *n-queens* problem. These problems typically consist of multiple closely connected partitions, in which deciding on variables in each partition can improve the search performance. In some sense, such partitioning establishes a *natural* ordering of the instance's decision variables. As an example, a speedup of 14, 14, 14, 12, and 8, was obtained for the *hole-10* benchmark over the MSTS, MSOS, DLCS, DLIS, and *fixed* decision heuristics, respectively. Some large instances from the *n-queens* set also achieved significant speedups over other decision heuristics. For example, none of the dynamic or *fixed* decision heuristic were able to solve the *nqueens-35* instance in 10,000 seconds. On the other hand, Capo's variable ordering solved the instance in less than 320 seconds. In general, GRASP run time is almost always reduced when the recursive bisection ordering is used. However, for particularly easy SAT instances recursive bisection itself required more time than GRASP with either *fixed*, MSTS, MSOS, DLCS, or DLIS ordering. Clearly, more research is needed to study the effects of partitioning and reordering on SAT instances and formulate practical recommendations that guarantee improvement, e.g., in the

Bench-mark	#I	MSTS		MSOS		DLCS		DLIS		Fixed		CAPO			Avg Var Cut	
		#I	Time	#I	Time	#I	Time	#I	Time	#I	Time	#I	Order	Time	Fix	New
aim	72	72	2.61	72	3.16	72	3.81	72	6.71	72	2.72	72	174.1	4.73	11676	6542
bf	4	4	2.63	4	4.97	4	2.56	4	2.3	3	10019	4	68.72	2.08	2853	440
dub	13	13	29.06	13	18.12	13	2.15	13	2.73	13	0.71	13	7.84	0.66	1717	106
hanoi	2	1	10005	1	12267	0	20000	0	20000	2	83.13	2	60.03	89.64	408	321
hole	5	3	26956	2	30193	4	11705	5	9466	5	6287	5	3.55	776.5	581	108
ii16	10	10	5407	10	6189	8	20259	9	10321	10	17685	10	726.8	84.22	76466	7935
ii32	17	16	11063	16	11187	17	9492.6	17	4.94	15	20598	16	488	10047	49616	11531
ii8	14	14	2.98	14	2.75	14	8.79	14	7.99	14	1.04	14	260.75	0.71	25396	2749
jnh	50	50	5.08	20	6.58	50	6.48	50	8.51	50	27.62	50	422.74	30.1	25952	22701
par16	10	10	21652	10	20470	8	27708	9	21855	10	2536	10	91.78	2713.8	4789	879
par8	10	10	0.19	10	0.21	10	0.22	10	0.22	10	0.21	10	16.63	0.16	1613	436
pret	8	8	0.72	8	0.68	8	0.7	8	0.66	8	0.59	8	4.75	0.64	865	138
ssa	8	8	97.33	8	12.63	8	3.73	8	2.44	6	20001	8	239.99	359.16	6104	768
Total	223	219	75224	218	80355	216	89193	219	61679	218	77242	222	2566	14109	208036	54654

TABLE I: Summary of GRASP runtimes for the DIMACS set

Selected Instances	S/U	MSTS Time	MSOS Time	DLCS Time	DLIS Time	Fixed Time	Capo			Capo Speedup				
							Order	Time	Total	MSTS	MSOS	DLCS	DLIS	Fixed
aim100-20no2	U	0.04	0.02	0.01	0.01	0.01	0.96	0.01	0.97	0.04	0.02	0.01	0.01	0.01
bf0432-007	U	1.72	3.85	1.74	1.48	10000	10.12	1.59	11.71	0.15	0.33	0.15	0.13	854
hanoi4	S	4.54	2267	10000	10000	1.75	14.82	1.64	16.46	0.28	138	608	608	0.11
hole8	U	6879	10000	140	70.31	61	0.44	9.01	9.45	728	1058	14.81	7.44	6.46
hole9	U	10000	10000	1556	752	623.1	0.62	60.08	60.7	165	165	25.63	12.39	10.27
hole10	U	10000	10000	10000	8637	5597	1.46	706	708	14.13	14.13	14.13	12.20	7.91
ii16b1	S	174	217	10000	10000	4840	119	1.49	120	1.44	1.80	83.09	83.09	40.22
ii16b2	S	133	153	71.26	238	5507	60.68	0.8	61.48	2.17	2.50	1.16	3.87	89.57
ii32c4	S	650	696	24.85	1.24	10000	102	0.9	102.9	6.32	6.77	0.24	0.01	97.18
par16-2-c	S	1321	1325	2469	3570	184	3.86	74.4	78.26	16.88	16.93	31.55	45.62	2.35
par16-5	S	7329	7348	315	10000	111	14.53	40.33	54.86	134	134	5.73	182	2.03
pret150_25	U	0.15	0.13	0.14	0.12	0.12	0.7	0.12	0.82	0.18	0.16	0.17	0.15	0.15
ssa0432-003	U	0.04	0.04	0.06	0.05	0.06	2.83	0.03	2.86	0.01	0.01	0.02	0.02	0.02
ssa2670-130	U	1.12	1.87	0.61	0.39	10000	10.86	357	368	0.00	0.01	0.00	0.00	27.18
Nqueens-20	S	482	1485	23	24.87	3160	40	0.31	40.31	11.96	36.84	0.57	0.62	78.39
Nqueens-25	S	10000	10000	178	183	94.89	92.64	0.79	93.43	107	107	1.90	1.96	1.02
Nqueens-30	S	10000	10000	5233	5402	10000	217	2.27	219	45.61	45.61	23.87	24.64	45.61
Nqueens-35	S	10000	10000	10000	10000	10000	317	1.06	318	31.42	31.42	31.42	31.42	31.42

TABLE II: GRASP runtimes for selected benchmarks from the DIMACS set and the *n-queens* problem

worst-case or average sense. Capo was also able to effectively reduce the average *variable cut* for all benchmarks. For some instances, such as the *bf** which represent bridging fault problems, the new average *variable cut* was reduced by an order of 6.5.

Although not presented in the tables of results, we tested the given benchmarks using the SATO [31] SAT solver. SATO implements an intelligent dynamic decision heuristics. SATO was able to solve the given DIMACS benchmarks in approximately 45,000 seconds (4 instances timed-out after 10,000 seconds) as opposed to 16,700 seconds using GRASP with recursive bisection ordering. However, for some instances, SATO was faster. Capo failed to generate effective variable ordering for these instances, since most of them were not structured.

BDD Experiment: Table III and Table IV show the runtimes needed to construct the BDDs of selected Boolean functions from the ISCAS89 circuit benchmarks and the DIMACS set. In both tables, the columns represent the *fixed, random, fixed* with *sifting, random* with *sifting*, and *Capo* orderings, respectively. The tables include benchmarks that were successfully built using any of the given orderings. *n.c.* indicates that the problem did not complete within the given resources. Clearly, not only was Capo's ordering faster in building the BDDs, but in most cases it required a smaller

Instance	Fixed		Random		Fixed-Sift		Random-Sift		Capo				Avg Cut	
	Build	**Max**	**Build**	**Max**	**Build**	**Max**	**Build**	**Max**	**Order**	**Build**	**Total**	**Max**		
	Time	**Node**	**Time**	**Node**	**Time**	**Node**	**Time**	**Node**	**Time**	**Time**	**Time**	**Node**	**Fix**	**New**
s208.1	n.c.		n.c.		14.8	6420	21.57	13593	0.88	0.66	1.54	3384	104	16
s27	0.06	181	0.07	204	0.08	181	0.08	204	0.18	0.07	0.25	73	11	5
s298	n.c.		n.c.		47.83	28005	56.55	26832	0.97	3.03	4	14495	157	28
s344	n.c.		n.c.		525.1	130538	703.45	192172	1.28	7.48	8.76	14214	137	17
s349	n.c.		n.c.		267.78	83319	707.24	248381	1.36	10.77	12.13	19290	149	18
s382	n.c.		n.c.		159.08	88182	113.25	32583	1.23	5.48	6.71	13597	176	26
s386	n.c.		n.c.		258.12	96688	168.84	48016	1.8	91.74	93.54	310441	172	55
s400	n.c.		n.c.		564.91	193893	292.7	114904	1.12	5.8	6.92	20060	182	26
s420	n.c.		n.c.		361.84	93977	590.79	122458	1.47	4.69	6.16	17673	183	19
s444	n.c.		n.c.		252.83	85039	605.12	241874	1.71	5.02	6.73	7731	192	25
s526	n.c.		n.c.		n.c.		n.c.		2.92	17.74	20.66	37656	271	42
s526n	n.c.		n.c.		n.c.		n.c.		1.99	10.35	12.34	18385	262	40
s641	n.c.		n.c.		n.c.		n.c.		2.35	42.12	44.47	158960	190	23
s713	n.c.		n.c.		n.c.		n.c.		2.86	62.86	65.72	174356	216	25
s838.1	n.c.		n.c.		n.c.		n.c.		3.74	105.46	109.2	147753	419	29
s838	n.c.		n.c.		n.c.		n.c.		3.82	322.13	325.95	885548	366	29

TABLE III: Statistics for constructing the BDDs of the ISCAS89 Benchmarks

Instance	Fixed		Random		Fixed-Sift		Random-Sift		Capo				Avg Cut	
	Build	**Max**	**Build**	**Max**	**Build**	**Max**	**Build**	**Max**	**Order**	**Build**	**Total**	**Max**		
	Time	**Node**	**Time**	**Node**	**Time**	**Node**	**Time**	**Node**	**Time**	**Time**	**Time**	**Node**	**Fix**	**New**
aim-100-1_6-no-1	0.55	33065	0.45	18918	0.33	3113	0.45	3465	0.72	0.08	0.8	222	84	32
dubois50	n.c.		n.c.		12.36	3215	14.66	4276	0.69	0.25	0.94	432	201	11
hole10	26	131071	116.43	3390725	12.75	19004	12.37	18985	1.46	0.38	1.84	19517	201	30
hole8	2.08	20223	4.43	145393	3.01	5095	3.03	4058	0.44	0.14	0.58	3767	108	21
hole9	7.74	52223	23.3	810524	5.37	8472	5.5	10216	0.62	0.2	0.82	8748	149	25
ii8a1	25.71	372073	n.c.		4.43	7220	7.22	6846	0.89	1.18	2.07	17558	75	20
par16-1-c	535.65	893751	n.c.		1826	500497	2140	434239	3.84	115.71	119.55	171529	271	101
par8-1	133.14	90742	113.37	160072	88.38	34503	97.35	28624	2.43	32.07	34.5	36260	253	39
pret150_25	n.c.		n.c.		649	271636	302.82	156136	0.7	1.01	1.71	3393	152	18
ssa0432-003	n.c.		n.c.		n.c.		n.c.		2.83	29.87	32.7	168678	287	46

TABLE IV: Statistics for constructing the BDDs for Selected DIMACS Benchmarks

number of nodes than other used orderings, including sifting. In terms of circuits, Capo performs an indirect ordering of the gates such that the length of wires in the circuit is minimized. This has a great impact on the runtime and BDD sizes. For the ISCAS89 circuits, Capo was able to construct the BDDs for all 16 circuits as opposed to 10 circuits with sifting and 1 circuit with a *fixed* variable-ordering. On average the new ordering has been able to obtain significant performance improvement in comparison with BDD variable sifting. Capo's ordering time is also minimal for most cases. The average *variable cut* for the ISCAS89 circuits was reduced from 200 to 26 using the new variable ordering. Accordingly, the average *variable cut* for the selected DIMACS benchmarks was also reduced from 178 to 34.

This approach has several advantages. The technique is simple and easy to use. Its "static" style allows for a variety of applications that dynamic approaches fail at. Furthermore, despite the small overhead required to pre-process the problem using Capo, the total runtimes are in general faster than other state-of-the-art available approaches such as dynamic sifting for BDDs or dynamic DLCS and DLIS orderings for SAT. The approach also minimizes the memory needed in most cases.

4 Conclusions & Future Work

Our work proposes a static variable-ordering heuristic MINCE for CNF formulae with applications to SAT and BDDs. The main advantage of this heuristic is its very good performance on standard benchmarks in terms of implied runtime of SAT solvers as well as memory/runtime of BDD primitives. We believe that this is due to the fact that the proposed variable-ordering is *global* and relies on high-performance hypergraph partitioning and placement (CAPO). Unlike problem-specific dynamic variable-ordering heuristics, such as MSTS, MSOS, DLCS, DLIS, and variable-sifting, MINCE can be implemented once and used for different applications without modifying the application code. Given that MINCE shows strong improvements for both SAT and BDDs on a wide variety of standard benchmarks, we believe that it is able to capture some structural properties of CNF instances. For example, when a CNF/BDD is created from a circuit, it is not difficult to see that MINCE essentially performs recursive partitioning and linear placement of this circuit, and then orders variables in CNF/BDD so that respective circuit elements are located near each other on average.

Our on-going work addresses additional types of benchmarks, better justifications of the MINCE heuristic and also analyses of the rare cases when it fails to produce near-best variable-ordering. An important direction for future research is to account for polarities of literals. It is, in fact, surprising that MINCE is so successful without even using polarities of literals. We are also preparing a public-domain implementation of MINCE.

5 Acknowledgments

We would like to thank the anonymous reviewers for pointing out helpful comments and suggesting additional references for the paper.

6 References

[1] C. Berman, "Circuit Width, Register Allocation, and Ordered Binary Decision Diagrams," *in IEEE Transactions on Computer Aided Design*, 10(8), 1991.

[2] A. Biere, A. Cimatti, E. M. Clarke, M. Fujita, and Y. Zhu, "Symbolic Model Checking using SAT procedures instead of BDDs," *in Proc. of the Design Automation Conference (DAC)*, 1999.

[3] F. Brglez, D. Bryan, and K. Kozminski, "Combinational problems of sequential benchmark circuits," *in Proc. of the International Symposium on Circuits and Systems*, 1989.

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

[5] A. Caldwell, A. Kahng and I. Markov, "Can Recursive Bisection Produce Routable Placements?" *in Proc. of the Design Automation Conference (DAC)*, 2000.

[6] A. E. Caldwell, A. B. Kahng, and I. L. Markov, "Improved Algorithms for Hypergraph Bipartitioning," *in Proc. of the IEEE ACM Asia and South Pacific Design Automation Conference*, 2000.

[7] M. Davis and H. Putnam, "A Computing Procedure for Quantification Theory," *in Journal of the Association for Computing Machinery*, vol. 7, pp. 201-215, 1960.

[8] R. Drechsler and B. Becker, "Binary Decision Diagrams, Theory and Implementation," *Kluwer Academic Publishers*, 1998.

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

[10] M. Fujita, H. Fujisawa, and N. Kawato, "Evaluation and Improvements of Boolean Comparison Method Based on Binary Decision Diagrams," *in Proc. of the International Conference on Computer Aided Design (ICCAD)*, 1988.

[11] M. Garey and D. Johnson, "Computers and Intractability, A guide to the Theory of NP-Completeness," *Freeman*, 1979.

[12] G. Hachtel and F. Somenzi, "Logic Synthesis and Verification Algorithms," *Kluwer Academic Publishers*, 1996.

[13] S. Hur and J. Lillis, "Relaxation and clustering in a local search framework: application to linear placement," *in Proc. of the Design Automation Conference (DAC)*, 1999.

[14] G. Karypis, R. Aggarwal, V. Kumar, and S. Shekhar, "Multilevel Hypergraph Partitioning: Applications in VLSI Design," *in Proc. of the Design Automation Conference (DAC)*, 1997.

[15] W. Kunz, D. Pradhan, "Recursive Learning: A new Implication Technique for Efficient Solutions to CAD-problems: Test, Verification and Optimization," *IEEE Transaction on Computer-Aided Design*, 13(9), pp. 1143-1158, 1994.

[16] T. Larrabee, "Test Pattern Generation Using Boolean Satisfiability," *IEEE Transactions on Computer-Aided Design*, vol. 11, no. 1, pp. 4-15, 1992.

[17] Y. Lu, J. Jain, and K. Takayama, "BDD Variable Ordering Using Window-based Sampling," *in International Workshop on Logic Synthesis (IWLS)*, 2000.

[18] S. Malik, A. Wang, R. Brayton, and A. Sangiovanni-Vincentelli, "Logic Verification using Binary Decision Diagrams in a Logic Synthesis Environment," *in Proc. of the International Conference on Computer Aided Design (ICCAD)*, 1988.

[19] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: Engineering an Efficient SAT Solver," *in Proc. of the Design Automation Conference (DAC)*, 2001.

[20] G. Nam, K. Sakallah, and R. Rutenbar, "Satisfiability-Based Layout Revisited: Detailed Routing of Complex FPGAs Via Search-Based Boolean SAT," *in Proc. of the International Symposium on Physical Design (ISPD)*, 1999.

[21] S. Panda and F. Somenzi, "Who are the variables in your neighborhood," *in Proc. of the International Conference on Computer Aided Design* (ICCAD), 1995.

[22] M. Prasad, P. Chong, and K. Keutzer, "Why is ATPG easy?" *in Proc. of the Design Automation Conference* (DAC), 1999.

[23] R. Rudell, "Dynamic variable ordering for ordered binary decision diagrams," *in Proc. of the International Conference on Computer Aided Design (ICCAD)*, 1993.

[24] J. Silva and K. Sakallah, "GRASP-A New Search Algorithm for Satisfiability," *in Proc. of the International Conference on Computer Aided Design (ICCAD)*, 1996.

[25] L. Silva, J. Silva, L. Silveira, and K. Sakallah, "Timing Analysis Using Propositional Satisfiability," *in IEEE International Conference on Electronics, Circuits and Systems*, 1998.

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

[27] G. Stalmarck, "System for Determining Propositional Logic Theorems by Applying Values and Rules to Triplets that are Generated from Boolean Formula," *United States Patent no.* 5,276,897, 1994.

[28] P. R. Stephan, R. K. Brayton, and A. L. Sangiovanni-Vincentelli, "Combinational Test Generation Using Satisfiability," *in IEEE Transactions on Computer-Aided Design*, 1996.

[29] R. G. Wood and R. A. Rutenbar, "FPGA Routing and Routability Estimation Via Boolean Satisfiability," *in IEEE Trans. on VLSI*, vol. 6, no. 2, June 1998.

[30] R. Zabih and D. A. McAllester, "A Rearrangement Search Strategy for Determining Propositional Satisfiability," *in Proc. of the Nat'l Conference on Artificial Intelligence*, 1998.

[31] H. Zhang, "SATO: An Efficient Propositional Prover," *in International Conference on Automated Deduction*, 1997.