FORCE: A Fast & 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 reduce 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: \((x_1)(\overline{x_1} + x_3)(\overline{x_1} + x_2)\).

Complete SAT search algorithms [14, 19] are often based on the Davis-Lemmon-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 canonical 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.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
given variable ordering, we define the original formula is reordered (see example in Figure 1). For a min-cut linear placement then produces an ordering vertices. One-dimensional min-cut linear placement leads to smaller number of BDD vertices. This averages centers of gravity of all hyperedge connected to vertex e. 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 |P| variables and |C| clauses. The average variable cut is equal to the product of the average clause span and |C|/(|P| – 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 $O(N\log N)$, where $N$ is the size of the input.

Procedure: FORCE {
1 randomly generate an initial order of vertices;
2 repeat limit times or until total span stops decreasing
3 for each hyperedge $e \in E$
4 compute center of gravity of $e$;
5 for each vertex $v \in V$
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.

force-directed placement. 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 pseudocode 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 $e$:

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

where $l_v$ and $|E_v| e$ 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 $v$:

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

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