Towards a Better Understanding of the Functionality of a Conflict-Driven SAT Solver ⋆

Nachum Dershowitz[1,3], Ziyad Hanna[2], and Alexander Nadel[1,2]

[1] School of Computer Science, Tel Aviv University, Ramat Aviv, Israel
{nachumd, ale1}@tau.ac.il
[2] Design Technology Solutions Group, Intel Corporation, Haifa, Israel
{ziyad.hanna, alexander.nadel}@intel.com
[3] Microsoft Research, Redmond, WA

Abstract. We show that modern conflict-driven SAT solvers implicitly build and prune a decision tree whose nodes are associated with flipped variables. Practical usefulness of conflict-driven learning schemes, like 1UIP or *All*UIP, depends on their ability to guide the solver towards refutations associated with compact decision trees. We propose an enhancement of 1UIP that is empirically helpful for real-world industrial benchmarks.

1 Introduction

Modern conflict-driven backtrack-search SAT solvers are widely used in applications in academia and industry. Each invocation can be associated with a decision tree, and tree pruning is a commonly used, intuitive concept for developing and analyzing enhancements. But, since the introduction of Conflict-Directed Backjumping (CDB) [4], it has become unclear how to characterize the decision tree built in the process. The main difficulty arises from the fact that a CDB-based solver may flip values of implied variables, rather than decision variables. Also, it may skip decision levels when backtracking. As a result of this vagueness, modern solvers are more commonly understood as resolution engines, using decision-tree construction as a heuristic, rather than as algorithms constructing decision trees (e.g., [3]). Unfortunately, this provides little insight for reasoning about the behavior of learning schemes and for developing new ones. Witness the statement [5]: "The effectiveness of certain ... schemes can only be determined by empirical data for the entire solution process".

We propose a framework that allows one to reason about a CDB-based solver as a decision-tree construction based engine. We rely on the following hypothesis: nodes in the decision tree, implicitly constructed by a CDB-based solver, are associated with flipped variables, rather than with initially picked decision variables. This approach allows us to explain why 1UIP [1] is empirically advantageous over other schemes (cf. [5, 3]). It also suggests a practically useful enhancement, called "local conflict clause recording".

⋆ This research was supported in part by the Israel Science Foundation (grant no. 250/05). The work of Alexander Nadel was carried out in partial fulfillment of the requirements for a Ph.D.

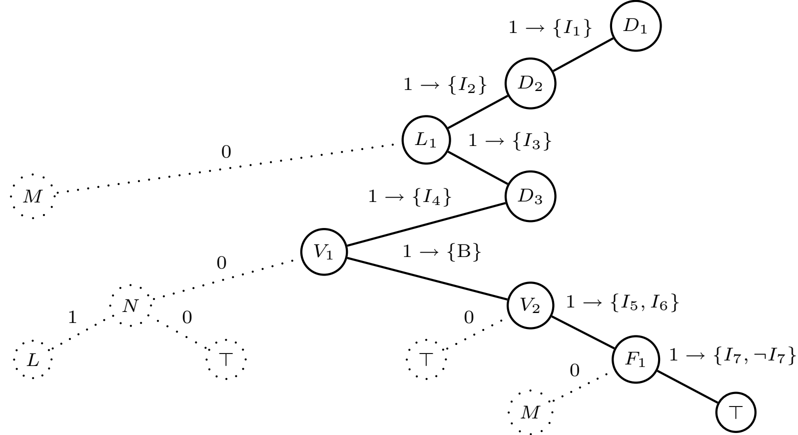

Fig. 1. Snapshot of a CDB-based solver run. The solid rightmost path is the current assignment stack. There are three decision levels. Each flipped variable is associated with a left decision subtree, denoted by dotted parts. Nodes correspond to decision or flipped variables and edges are marked with the Boolean values assigned to these variables and, optionally, with implied literals.

2 Implicit Decision-Tree Construction and Pruning

An *asserting conflict clause* is a conflict clause containing the negation of one and only one literal, called a *pivot literal*, assigned at the last decision level. The 1UIP [1], 2UIP [5] and *All*UIP [5] clauses are all asserting. After the pivot variable is flipped, it is called a *flipped variable*. The *parent clause* of an implied literal A, denoted $Par(A)$, is the clause where the value of A is implied.

Decision-tree construction for plain backtracking can be understood as adding a new node to the tree, labeled with a decision variable B, assigned value $\sigma = Val(B)$, and a new left edge, labeled σ, upon each decision. The left subtree of B, denoted $LTree(B)$, is constructed recursively. When the solver backtracks to B and flips $Val(B)$, the tree is updated with a new right edge, labeled $\neg\sigma$, and a right subtree is constructed.

In our view, a CDB-based solver maintains a forest of left subtrees. Every flipped variable is associated with a left subtree. The forest is merged into one tree, comprising a refutation trace of the whole formula, only after the last conflict. Upon conflict, when a pivot variable B is flipped, its left decision subtree is constructed by merging left subtrees of a subset of flipped variables, assigned after B. Suppose the solver is in a conflict situation, the conflicting clause is γ and the decision level is k. We call a flipped variable that belongs to level k an *lf-variable*, and a flipped variable that belongs to levels lower than k an *lu-variable*. An lf-variable is *active* if it is connected to γ and is dominated by B in the implication graph. In our example (Fig. 1 and Fig 2(a)), the only active lf-variable is F_1. Lf-variable V_1 is not dominated by B. Lf-variable V_2 is not connected to the conflicting variable. Thus, both V_1 and V_2 are inactive.

Algorithm 1 constructs the left decision subtree of a pivot variable B. A recursive function *TNewTree* is invoked. It receives four parameters: (1) root

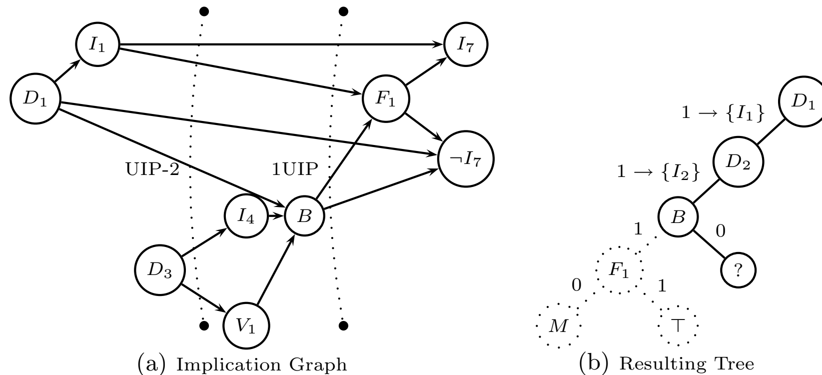

Fig. 2. Implication graph and decision tree for Fig. 1 with 1UIP and UIP-2 cuts and the resulting tree after applying Algorithm 1 and conflict-driven backjumping for 1UIP scheme.

Algorithm 1 *On conflict, returns* $LTree(B)$ *of the pivot variable* B

1: Let $F_1 \dots F_n$ be active lf-variables. Suppose $LTree(F_{n+1})$ and $Tree(F_1)$ are leaves.
2: **for** $i := n$ **downto** 1 **do**
3: $\quad Tree(F_i) := TNewTree(F_i; \neg Val(F_i); LTree(F_i); Tree(F_{i+1}))$
4: **return** $Tree(F_1)$

variable; (2) first value of the root variable; (3) left subtree; and (4) right subtree. See Fig. 2(b) for the result of applying Algorithm 1 and conflict-driven backjumping for 1UIP scheme in our example.

Applying Algorithm 1 allows a CDB-based solver to *skip* some flipped variables. Skipping a flipped variable means excluding its left subtree from the final decision tree characterizing the run of a solver. Skipped variables fall into three categories: (1) lu-variables, skipped during backtracking (L_1 in our example); (2) inactive lf-variables, connected to the conflicting clause vertices, but not dominated by the pivot variable (V_1 in our example); (3) inactive lf-variables, not connected to the conflicting clause vertices (V_2).

We distinguish between two types of decision-tree pruning: *backward tree pruning* is carried out upon conflict detection by skipping existing subtrees; *forward tree pruning* is performed by recording conflict clauses useful in terms of frequent participation in Boolean constraint propagation (BCP) during the subsequent search. Algorithm 1 carries out backward tree pruning implicitly by not including the left decision subtrees of inactive lf-variables in the left decision subtree of the pivot variable. To the best of our knowledge, this kind of decision-tree pruning has not been highlighted in the literature. A more prominent kind of backward tree pruning is carried out by the solver while backtracking non-chronologically [4]. We underscore the fact that the effectiveness of this kind of pruning depends on the size of the left decision subtrees of skipped flipped variables, rather than on the number of skipped decision levels, as usually presumed.

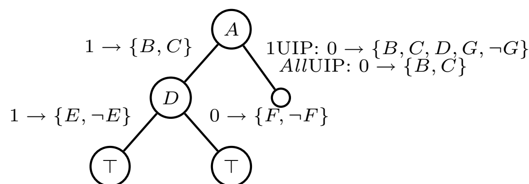

Fig. 3. Example of superiority of 1UIP over *All*UIP. Suppose we invoke a CDB-based SAT solver on an input formula $(A \vee D \vee G) \wedge (A \vee D \vee \neg G) \wedge (A \vee C) \wedge (A \vee B) \wedge (\neg A \vee B) \wedge (\neg A \vee C) \wedge (\neg B \vee \neg C \vee \neg D \vee E) \wedge (\neg B \vee \neg C \vee \neg D \vee \neg E) \wedge (\neg A \vee D \vee F) \wedge (\neg A \vee D \vee \neg F)$. The solver first picks the literal A, propagates its value, then picks D, propagates and encounters a conflict. The 1UIP clause is $\neg B \vee \neg C \vee \neg D$; the *All*UIP clause is $\neg A \vee \neg D$. After flipping D, both the *All*UIP and the 1UIP conflict clauses are $\neg A$. After propagating, 1UIP would yield a conflict, meaning that the formula is unsatisfiable. In contrast, *All*UIP would not result in a conflict, since all previously recorded conflict clauses are satisfied

3 Usefulness of Conflict-Clause Recording Schemes

The *UIP-2* scheme for conflict learning takes UIP number 2 of the last decision level as the pivot variable. We compared the best known scheme, 1UIP [1], with *All*UIP [5] and UIP-2, which we feel are representative enough to explain the advantages of 1UIP over other schemes, too. (We do not discuss conflict clause minimization due to space restrictions.)

Choosing the first UIP, rather then UIP number 2 of the last decision level, is optimal for backward pruning. Indeed, the first UIP is the closest to the conflict; thus it tends to dominate fewer lf-variables. Also, the first UIP allows backtracking to the highest possible decision level, maximizing the number of uf-variables skipped during backtracking.

Why is 1UIP better than *All*UIP? Replacing literals of other decision levels by their dominator does not impact backward tree pruning. Indeed, the number of inactive lf-variables and the backtrack level remain the same. We claim that 1UIP clauses tend to contribute more to BCP than *All*UIP clauses, so are more useful for forward pruning. Let B be the pivot variable and k the decision level at the moment of a conflict. Denote by $Fr^{+}(B)$ the fraction of the conflict clauses that contain the variable B out of all conflict clauses recorded since B was last assigned. The key observation, confirmed empirically in Sect. 5, is that $Fr^{+}(B)$ tends to be much higher for *All*UIP than for 1UIP. Indeed, 1UIP conflict clauses tend to contain literals implied from B at k, rather than B itself. *All*UIP clauses tend to contain B, since B dominates all the literals at k. Hence, after flipping B, more of the *All*UIP conflict clauses, recorded before the flip, will be satisfied and will not contribute to BCP (compared with 1UIP conflict clauses). See Fig. 3 for an example.

4 Local Conflict-Clause Recording

A *Local Conflict-Clause (LCC)* is a non-asserting conflict clause, recorded in addition to the 1UIP conflict clause if the last decision level contains some active lf-variables. To record it, the last active lf-variable is considered to be a decision

variable, defining a new decision level. An LCC is the 1UIP clause with respect to this new decision level.

A clause α is *inconsistent* with a decision-tree path P if α contains the negation of one of the literals of P. Consider a conflict situation, with pivot variable B and active lf-variables $F_1, F_2, \ldots, F_n$. Suppose the leftmost path of $LTree(B)$ is $P_1 = (G_1, \ldots, G_l)$. The rightmost path of $LTree(B)$ must be $P_f = (F_1, \ldots, F_n)$. The key observation is that there is an asymmetry between P_1 and P_f in that P_1 tends to be inconsistent with more clauses than P_f. Indeed, each of the clauses $Par(G_i)$ is inconsistent with P_1, since it must contain $\neg G_i$. This is not the case with P_f. It is not guaranteed that there exist clauses containing $\neg F_j$, since parent clauses of F_j's contain F_j rather than $\neg F_j$. Denote the number of left edges in a path by $\ell(P)$. An arbitrary path P in $LTree(B)$ is guaranteed to be inconsistent with at least $\ell(P)$ clauses. In general, the greater $\ell(P)$, the greater the chance is that there will be aggressive propagation, once the literals of P are assigned.

The main goal of adding LCCs is to improve forward tree pruning when literals, corresponding to a path with small $\ell(P)$, are assigned. In addition, LCCs tend to contribute more to BCP than 1UIP clauses immediately after flipping the pivot variable. Indeed, after flipping the pivot variable, the 1UIP clause is always satisfied, whereas the local conflict clause may contribute to BCP, since it may not contain the pivot variable.

5 Experimental Results

We implemented 1UIP, UIP-2 and *All*UIP within the industrial CDB-based solver, Eureka [2] (but without decision-stack shrinking). All experiments were carried out on a machine with 4GB memory and two Intel Xeon CPU 3.06 processors. We used instances from 11 well-known industrial benchmark families. These three schemes are compared in Table 1 on 8 instances.

The main conclusions of our experiments are: (1) 1UIP is indeed more powerful and robust than other schemes. It is always faster than UIP-2, and outperforms *All*UIP by orders of magnitude on 4 instances, appearing in the left column of Table 1. (2) Fr^+ is double for *All*UIP than for 1UIP. This explains 1UIP's superiority over *All*UIP by confirming the hypothesis of Sect. 3. (3) Of all schemes, UIP-2 skips the fewest nodes/flipped variables. Additional empirical findings, omitted here, show that this happens mainly due to the fact that there are fewer inactive lf-variables not dominated by the pivot variable in the implication graph. This agrees with the theoretical analysis in Sect. 3. (4) Surprisingly, *All*UIP allows one to skip more nodes and flipped variables than 1UIP on some examples. We found that it happens mainly due to the fact that many lf-variables are not connected to the conflicting clause for *All*UIP. According to the analysis in Sect. 3, the number of skipped nodes and variables should be about the same for both schemes. This expected behavior is indeed observed on the 4 instances of the left column of Table 1, where *All*UIP is outperformed by several orders of magnitude. Studying the reasons for the unexpected behavior

Table 1. Comparing 1UIP, UIP-2 and *All*UIP on selected instances. The rows display: (Tm) execution time in seconds; (Con) number of conflicts; (Fr^+) average Fr^+; (NSk) average number of decision-tree nodes skipped per conflict

Instance	Res	1UIP	UIP-2	*All*UIP	Instance	Res	1UIP	UIP-2	*All*UIP
4pipe	Tm	51	148	11930	*longmult10*	Tm	485	513	590
	Con	101277	308946	29985706		Con	237814	261669	379737
	Fr^+	0.41	0.38	0.83		Fr^+	0.37	0.34	0.84
	NSk	0.19	0.14	0.24		NSk	0.13	0.11	0.24
5pipe	Tm	50	347	> 14400	*longmult11*	Tm	559	756	690
	Con	85119	562304	28185547		Con	273200	346414	471626
	Fr^+	0.40	0.33	0.84		Fr^+	0.37	0.35	0.83
	NSk	0.18	0.14	0.21		NSk	0.14	0.11	0.25
8pipe_k	Tm	2426	> 14400	> 14400	*rotmul*	Tm	578	1186	992
	Con	1478419	10129202	13192438		Con	615314	1371339	1576324
	Fr^+	0.37	0.26	0.81		Fr^+	0.52	0.48	0.84
	NSk	0.21	0.13	0.19		NSk	0.16	0.13	0.27
9pipe_k	Tm	1493	> 14400	> 14400	*term1mul*	Tm	2173	5213	2975
	Con	640559	6040439	6548156		Con	1585135	3750774	3059096
	Fr^+	0.37	0.27	0.85		Fr^+	0.55	0.54	0.86
	NSk	0.20	0.16	0.20		NSk	0.15	0.11	0.26

Table 2. Effect of LCC recording (time is in sec.; t/o is the number of instances that timed out)

Family	Threshold	Default Time	Default t/o	Def. + LCC Time	Def. + LCC t/o
sat04_ind_maris03_gripper_sat	3 hours	2238	0	986	0
sat04_ind_goldberg03_hard_eq_check	3 hours	30336	2	15353	0
sat04_ind_maris03_gripper_unsat	4 hours	30135	4	17842	2
velev_fvp_unsat.3.0	3 hours	18199	2	10928	2
velev_fvp_sat.3.0	3 hours	9041	0	7155	0
velev_vliw_sat_2.0	3 hours	5970	0	4715	0
barrel	3 hours	260	0	226	0
velev_pipe_unsat_1.0	3 hours	15880	0	13094	0
velev_vliw_unsat_4.0	3 hours	17260	0	14810	0
longmult	3 hours	5413	0	5076	0
velev_vliw_sat_4.0	3 hours	5116	0	6882	0

on the other 4 instances, where the gap between 1UIP and *All*UIP is not large, is left for future research.

Table 2 shows the effect on 11 families of local conflict-clause recording within the default version of Eureka. The technique is helpful overall on 10 of them. Accordingly, LCC recording can be recommended as a default strategy for modern CDB-based solvers.

References

1. M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: engineering an efficient SAT solver. In *DAC'01*, pages 530–535, 2001.
2. A. Nadel, M. Gordon, A. Palti, and Z. Hanna. Eureka-2006 SAT solver. `http://fmv.jku.at/sat-race-2006/descriptions/4-Eureka.pdf`.
3. L. O. Ryan. Efficient algorithms for clause learning SAT solvers. Master's thesis, Simon Fraser University, Burnaby, Canada, 2004.
4. J. P. M. Silva and K. A. Sakallah. GRASP-a new search algorithm for satisfiability. In *ICCAD'96*, pages 220–227. IEEE Computer Society, 1996.
5. L. Zhang, C. F. Madigan, M. H. Moskewicz, and S. Malik. Efficient conflict driven learning in a boolean satisfiability solver. In *ICCAD'01*, pages 279–285. IEEE Press, 2001.