HaifaSat: A New Robust SAT Solver

Roman Gershman[1] and Ofer Strichman[2]

[1] Computer Science, Technion, Haifa, Israel
[2] Information Systems Engineering, Technion, Haifa, Israel
gershman@cs.technion.ac.il, ofers@ie.technion.ac.il

Abstract. The popular abstraction/refinement model frequently used in verification, can also explain the success of a SAT decision heuristic like Berkmin. According to this model, conflict clauses are abstractions of the clauses from which they were derived. We suggest a clause-based decision heuristic called Clause-Move-To-Front (CMTF), which attempts to follow an abstraction/refinement strategy (based on the resolve-graph) rather than satisfying the clauses in the chronological order in which they were created, as done in Berkmin. We also show a resolution-based score function for choosing the variable from the selected clause and a similar function for choosing the sign. We implemented the suggested heuristics in our SAT solver HaifaSat. Experiments on hundreds of industrial benchmarks demonstrate the superiority of this method comparing to the Berkmin heuristic. There is still room for research on how to explore better the resolve-graph information, based on the abstraction/refinement model that we propose.

1 Introduction

A SAT solver can be thought of as a search engine based on *enumeration* of solutions, but also as a *proof engine* based on inference through the resolution rule. Traditionally the first view was dominant, hence the emphasis in designing SAT solvers and explaining their success was on pruning search spaces. Decision heuristics and learning schemes can all be interpreted as aiming at this goal. Yet the harder and larger the CNF instances are, pruning alone cannot account for the success of modern SAT solvers. It is their ability as proof engines that makes them succeed. This distinction has practical implications, too. For example, for many years decision heuristics gave higher priority to variables in shorter clauses, and to learning shorter conflict clauses. The reasoning was that such clauses can potentially prune larger search-spaces. Although this claim is true, all modern decision heuristics (VSIDS [4], VMTF [3], Berkmin [2]) ignore the length of the clauses, after reaching empirically the conclusion that there are more important considerations. Ryan experimented in his thesis [3] with first-UIP and all-UIP learning schemes, and although the latter generate on average shorter clauses, the former is empirically better. He hypothesized that the learning scheme should be geared towards resolution rather than for pruning. In this article we extend this approach by looking on clause-learning and the decision

S. Ur, E. Bin, and Y. Wolfsthal (Eds.): Haifa Verification Conf. 2005, LNCS 3875, pp. 76–89, 2006.

heuristic as one complete mechanism and refer to a SAT solver as a prover rather than as a search engine. It turns out, empirically, that when conflict clauses are effective, which is the case in all real-world instances, this is the right way to go.

Conflict clauses are derived through a process of resolution (see, for example, [6] and [1] for a more formal treatment of this subject). If a clause c is derived by resolution from a set of clauses $c_1 \dots c_n$ then

$$c_1 \wedge \dots \wedge c_n \rightarrow c$$

while the other direction does not hold. This means that we can see c as an over-approximating abstraction of the resolving clauses $c_1 \dots c_n$. Attempting to satisfy c first, therefore, can be seen as an attempt to satisfy the abstract model first. Like any abstraction/refinement technique, a successful assignment to c is one that satisfies the concrete model (the $c_1 \dots c_n$ clauses) as well. And an unsuccessful assignment leads to a refinement step, or, in our case, to derivation of new conflict clauses which further constrain the abstract model. According to this model, Berkmin is only one of many possible strategies to refine the abstract model. In Section 3 we suggest one such alternative clause-based decision heuristic called Clause-Move-To-Front (CMTF), which attempts to follow the order of the clauses in the *resolve-graph* [7] rather than their chronological order in which they were created. In Section 4 we also show a resolution-based score function for choosing the variable from the selected clause and a similar function for choosing the sign. In Section 5 we report experimental results on hundreds of industrial benchmarks that prove the advantage of our approach.

2 Background

The explanation of our methods and the analysis of various heuristics later on will require some basic definitions.

Abstraction and refinement of formulas. While the typical use of the terms *abstraction* and *refinement* refer to models and programs, here we define them in the context of formulas.

We say that a formula $\hat{f}$ is a conservative abstraction (over-approximation) of another formula f if

$$f \rightarrow \hat{f}$$

A refinement process of $\hat{f}$ with respect to f finds an intermediate formula $\hat{f}_1$ such that

$$f \rightarrow \hat{f}_1 \quad \text{and} \quad \hat{f}_1 \rightarrow \hat{f}$$

Abstraction-Refinement is an iterative process in which one begins with some abstract formula $\hat{f}$ of a concrete formula f and gradually refines it through a series of formulas $\hat{f}_1, \dots, \hat{f}_n$ until proving or disproving the desired property of f (in the worst case $\hat{f}_n = f$).

Restricting our attention to CNF formulas, suppose we have two sets of clauses, S and $\hat{S}$, s.t. $\hat{S} \subseteq S$. It is straight forward that $S \rightarrow \hat{S}$ and, therefore, $\hat{S}$ is an abstraction of S.

2.1 Conflict Clauses and Resolution

The well-known binary resolution rule is:

$$\frac{a_1 \vee \ldots \vee a_n \vee \beta \qquad b_1 \vee \ldots \vee b_m \vee (\neg\beta)}{a_1 \vee \ldots \vee a_n \vee b_1 \vee \ldots \vee b_m}$$

where $a_1, \ldots a_n, b_1, \ldots b_m, \beta$ are literals. β is known as the *resolution variable* (also known as the *pivot variable*) of this derivation. Clauses $(a_1, \ldots, a_n, \beta)$ and $(b_1, \ldots, b_n, \overline{\beta})$ are called *resolving clauses* and clause $(a_1, \ldots, a_n, b_1, \ldots b_n)$ is a *resolvent*. It follows by the soundness of the rule, that the resolvent is always implied by its resolving clauses and can therefore be thought of as an abstraction of the clauses that participated in the derivation.

Algorithm 1. The First-UIP resolution algorithm

```
    procedure AnalyzeConflict(Clause: conflict)
 2:   currentClause ← conflict;
      ResolveNum ← 0;
 4:   NewClause ← ∅;
      repeat
 6:     for each literal lit ∈ currentClause do
          v ← var(lit);
 8:       if v is not marked then
            Mark v;
10:         if dlevel(v) = CurrentLevel then
              ++ ResolveNum;
12:         else
              NewClause ← NewClause ∪ {lit};
14:         end if
          end if
16:    end for
       u ← last marked literal on the assignment stack;
18:    Unmark var(u);
       −− ResolveNum;
20:    ResolveCl ← Antecedent(u);
       currentClause ← ResolveCl \ {u};
22:  until ResolveNum = 0;
     Unmark literals in NewClause;
24:  NewClause ← NewClause ∪ {ū};
     Add NewClause to the clause database;
26: end procedure
```

We now show why the process of generating conflict clauses indeed can be seen as a sequence of resolution steps. Algorithm 1 shows a simple and efficient implementation of the First-UIP resolution scheme, which is implemented in most competitive SAT solvers, including our solver HaifaSat. We will refer to this algorithm simply as the *resolution* algorithm. First, a conflicting clause is

set to be the current resolved clause. The main loop processes literals in the current clause. All literals from the previous decision levels are gathered into $NewClause$ at line 13 and marked. Literals from the current level are marked in order to resolve on them (i.e., use them as resolution variables) further. In every iteration a new marked (yet unprocessed) literal u is chosen in line 17. This literal must be from the current decision level. The algorithm resolves on u by setting $currentClause$ to be the antecedent clause without u.

$ResolveNum$ counts the number of the marked literals from the current decision level that still have to be processed. When $ResolveNum = 0$ at line 22, then u is the $FirstUIP$ or the *asserted* literal. The negation of this literal is added to the $NewClause$ causing u's value to be flipped after backtracking. For more details on the resolution algorithm see [4,3].

We will use the following definition in order to denote the initial state of $NewClause$:

Definition 1 (Asserting clause). *Suppose a new conflict clause C was created in Alg. 1 with asserted literal u. Suppose also that the solver backtracks after the conflict to level dl. Then C becomes an* asserting *clause when it implies $\overline{u}$ for the first time at level dl, and stops being asserting when the solver backtracks from dl.*

It follows from definition that every conflict clause becomes asserting exactly once.

$$
\begin{aligned}
c_1 &= (\neg x_4 \vee x_2 \vee x_5)\\
c_2 &= (\neg x_4 \vee x_{10} \vee x_6)\\
c_3 &= (\neg x_5 \vee \neg x_6 \vee \neg x_7)\\
c_4 &= (\neg x_6 \vee x_7)\\
&\vdots \quad \vdots
\end{aligned}
$$

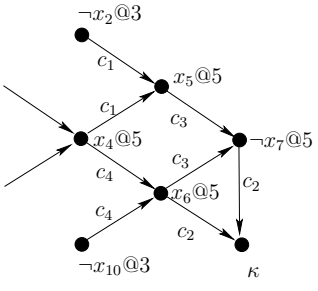

Fig. 1. A partial implication graph and set of clauses demonstrate AnalyzeConflict. x_4 is the $FirstUIP$, and $complx_4$ is the asserted literal.

Example 1. Consider the following partial implication graph [5] and set of clauses Denote by $Resolve(s, t, x)$ the binary resolution of clauses s and t with the resolution variable x. Then the conflict clause $c_5 : (x_{10}, x_2, \neg x_4)$ is computed through a series of binary resolutions, starting from the conflicting clause c_4, and going backwards on the implication graph until all literals in the conflict clause are either from previous decision levels or the $firstUIP$.

$$Resolve(Resolve(Resolve(c_4, c_3, x_7)), c_2, x_6), c_1, x_5) = (x_{10}, x_2, \neg x_4)$$

Algorithm 1 implicitly performs these resolution steps while computing the conflict clause c_5. □

NewClause is derived through a series of binary resolutions that can be seen as a tree: every time the solver reaches line 21, an intermediate clause (consisting of all marked literals) is resolved with the antecedent clause of the chosen resolution variable. We can treat this process as one atomic action of *Hyper-resolution* (resolution between more than two clauses). Since each conflict clause is derived from a set of other clauses, we can keep track of this process with a *Resolve-Graph* [7]. Here we define a variation of the well-known resolve-graph that distinguished between two types of resolutions:

Definition 2 (Colored Resolve Graph). *A* Resolve Graph *is a directed acyclic graph where each node corresponds to a clause, and there is an edge* (u, v) *if and only if* v *participated in the Hyper-resolution of* u *as a CurrentClause at line 6 of Alg 1.*

The color *of the edge* (u, v) *is defined to be blue if* v *was an asserting (conflict) clause during the resolution and red otherwise.*

In this graph, edges come from the resolvent to its resolving clauses. The leafs of the graph correspond to the original clauses in the formula. Notice that since a conflict at level *dl* necessarily implies that the solver backtracks from *dl* and unassigns all the variables that were resolved on, any asserting clause which participated in the resolution will stop being asserting. Therefore for any conflict clause there can be at most one incoming blue edge. The original clauses do not have outgoing edges, and only red incoming edges.

Example 2. Consider once again the implication graph in Figure 1. Since $c_1 \ldots c_4$ participate in the resolution of c_5, the corresponding resolve-graph is as appears in Figure 2. Assuming that $c_1 \ldots c_4$ are original clauses, then all the edges in this graph are red, because original clauses cannot be asserting.

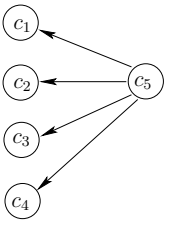

Fig. 2. A resolve-graph corresponding to the implication graph in Figure 1

Now consider a similar case in which c_2 is not an original clause, and at the time when $x_4@5$ is assigned it does not yet exist (the notation $l@i$, adopted from [5], means that literal l is satisfied at decision level i). The implication graph at this stage appears in Figure 3. Now assume that due to further decisions and implications in deeper decision levels a conflict is encountered, the solver creates

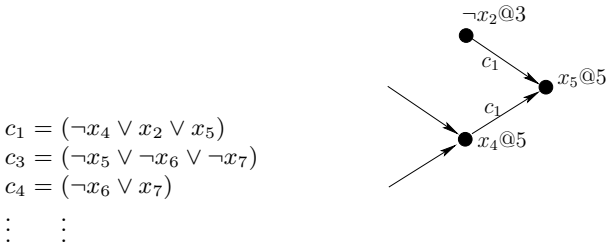

Fig. 3. A partial implication graph corresponding to c_1, c_3, c_4 and the decision $x_4@5$

the new conflict clause c_2, backtracks to decision level 5 and asserts $x_6@5$. This, in turn, completes the implication graph to the way it looks in Figure 1. But now, since c_2 asserts x_6, we consider its edge on the resolve-graph from c_5 as blue. □

The distinction between the two type of edges is important because a blue edge (u, v) indicates that the solver had to create u in order to later create v[1].

2.2 The Berkmin Decision Heuristic

We describe shortly the Berkmin Decision heuristic, since not only that it is considered to be one of the best known, but also because it is a clause-based decision heuristic, like HaifaSat's heuristic, and therefore convenient for comparison.

Berkmin [2] pushes every new conflict clause to a stack, and makes a decision by choosing an unassigned variable from the last unsatisfied conflict clause in this stack (if there is more than one such variable, it uses the VSIDS score system[4]). If all the conflict clauses are satisfied, it continues with a different heuristic.

In Fig. 4(a) we show a sketch of the progress of Berkmin, which is helpful in understanding why this process can be seen as abstraction-refinement. Clauses $c_1, \ldots, c_{100}$ are conflict clauses ordered by their creation time (c_1 is first). Berkmin tries to satisfy these clauses from last to first, i.e. from right to left. Suppose that all clauses $c_{51} \ldots c_{100}$ are already satisfied, and now Berkmin focuses on c_{50}. We refer to $S = \{c_{51}, \ldots, c_{100}\}$ as our current abstract formula of the original formula φ (it is abstract because each of the clauses in S is derived by a resolution chain from the clauses of φ). Clauses in S must be satisfied, since the decision heuristic reached c_{50}. Therefore, S is an abstract formula of φ. Berkmin now makes a decision on a variable from c_{50} which leads to a conflict and learning of a new clause. The decision heuristic backtracks to the clauses on the end of the list, until finally, through possibly additional iterations of conflicts and added clauses, it reaches c_{50} again while all the clauses to its right are satisfied. Denote by S' the clauses to the right of c_{50} at this point, e.g. $S' = \{c_{51} \ldots, c_{110}\}$. Clearly $S \subseteq S'$ and S' is an abstraction of φ. We can therefore say that S' is a refinement of S with respect to φ.

[1] By this we do not mean that this is the only way to create v.

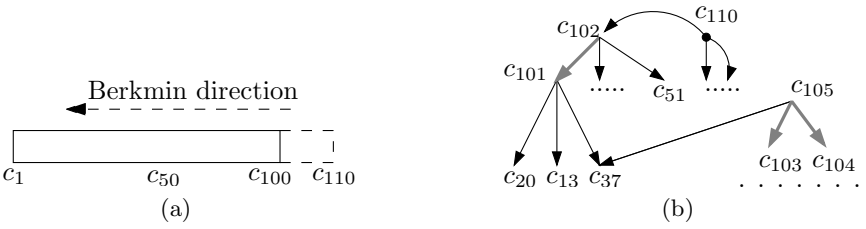

Fig. 4. Berkmin's decision heuristic can be thought of as an abstraction-refinement, where a range of the conflict clauses from the right end until c_i represents an abstract model of the clauses on the left of c_i. (a) Berkmin clauses stack: after encountering a conflict, the new resolved clauses are added on the right end. By the time the solver returns to c_{50}, it will have a partial assignment that satisfies a refined model, i.e. the clauses $c_{51} \ldots c_{110}$ (b) The resolve sub-graph of some newly created clauses. Grey thick edges denote the blue edges in the graph.

This view of the process possibly explains why a strategy of giving absolute priority to variables in a specific clause is empirically better than previous approaches like VSIDS that used only a score function.

Fig. 4(a) shows a 'linear' view of the conflict clauses in the order that they are added, which is also the order in which they are considered by Berkmin. The Berkmin heuristic never tries to satisfy a clause before satisfying its resolvents and thus mimics a gradual process of refinement.

A different view of conflict clauses considers their partial order in the Resolve Graph. Fig.4(b) presents a possible Resolve sub-Graph corresponding to the same set of clauses. After the conflicts, Berkmin starts from satisfying c_{110}. c_{102} is a resolving clause that can potentially refine the initial model, however Berkmin first passes through $c_{105}, c_{104}, c_{103}$ to which c_{110} is not connected at all. Therefore Berkmin is dispersed trying to refine several abstractions. Such unfocused behavior can lead to longer proofs. This problem is exactly what our decision heuristic CMTF attempts to solve, as we soon show.

Our SAT solver HaifaSat makes a decision in three steps: it chooses an unsatisfied clause according to the CMTF heuristic, it then chooses an unassigned variables from this clause, and finally gives it a value. The next sections describe in detail these decision steps.

3 The Clause-Move-To-Front (CMTF) Decision Heuristic

The description above of Berkmin's decision heuristic, and the alternative view of the conflict clauses as being part of a resolve-graph, hints towards the process which is described in Figure 5. In the first line *roots*(*ResolveGraph*) refer to resolvent clauses that did not resolve other clauses. Note that in this general scheme a clause is processed only if at least one of its abstractions (its resolvent clauses) has already been processed. It is easy to see that Berkmin is an

1: $S = roots(ResolveGraph)$;
2: Choose an unsatisfied clause (vertex) $v \in S$; If there is no such clause, exit;
3: Process v; ▷ Processing a clause, among other things, satisfies it.
4: $S = S \cup children(v)$;
5: Goto 2

Fig. 5. A Resolve-Graph Based decision heuristic

instantiation of the scheme. In fact, Berkmin is more strict and processes a clause only if *all* its abstractions are satisfied.

CMTF is a method that instantiates this scheme in a different way. It causes the decision heuristic to be more focused on the current refinement path, i.e. to satisfy children of the currently satisfied clause s. It works as follows:

- All the conflict clauses are stored in a list.
- During the resolution in Alg 1, a bounded number of resolving conflict clauses which are processed at line 6 are moved to the front (front corresponds to the right end of Fig 4(a)). The newly created clause $NewClause$ is also added to the list (can be done at line 25).
- Clauses are processed from right to left in the list, while ignoring satisfied clauses. If all the conflict clauses are satisfied then the original VMTF strategy (from Siege [3]) is applied.

The idea of this strategy is to keep clauses that participate in resolution adjacent to their resolvents (at least until the next time they participate in a resolution, a case in which they can be moved to a new location).

CMTF shows a big improvement on many industrial problems comparing to the Berkmin heuristic. Both are specific instantiation of the scheme showed above. The advantages of CMTF is its simplicity and the fact that the explicit storage of the resolve-graph is not required. However, it seems that there is still room for future research on how to use the general scheme. For example, classic AI search methods like best-first-search can be used to decide on the exploration order of nodes in S at line 2. It may happen that partial or full storage of the resolve-graph will improve the performance.

4 Resolution-Based-Scoring

In the previous section we showed how HaifaSat decides which clause to satisfy first. Given a clause c there can still be several ways to satisfy it. HaifaSat computes dynamically an *activity score* for each variable and then chooses the variable with the maximal score. Then another *sign score* is used to determine its Boolean value.

We define a scoring heuristic based solely on the resolution algorithm (Algorithm 1). The idea, intuitively, is to give higher weights to variables that were frequently resolved on recently, while distinguishing between resolutions that were necessary for the progress of the solver, and those that were made due to

the imperfection of the decision heuristic. We will need several definitions and lemmas to explain this heuristic more precisely.

Suppose that every time the solver makes a decision or processes a conflict it writes into a log the event $a_i = (dl, e)$ where dl is the decision level where the event occurred and $e \in Conflicts \cup Decisions$ is either a conflict event or a decision event. The global index i is incremented every time the event happens. We call the sequence $\{a_i\}_1^N$ the *flat log* of the solver's run. We will denote by $DL(a_i)$ the decision level of the event. We consider only the case in which $dl > 0$. All conflict events other, potentially, than the last one in an unsatisfiable instance are included by this definition[2]. It must hold that for any conflict c there exists a decision d at the same level as c. In such a case, we say that d is refuted by c. More formally:

Definition 3 (Refuted decision by a conflict). *Let $a_j = (dl, c)$ be a conflict event. Let $a_k = (dl, d)$, $k < j$, be the last decision event with decision level dl preceding a_j (note that for $i \in [k+1, j-1] : DL(a_i) > dl$). We say that d is the* refuted decision *of the conflict c, and write $\mathbf{D(a_j)} = a_k$.*

Note that because of non-chronological backtracking the opposite direction does not hold: there are decisions that do not have conflicts on their levels that refute them.

For any conflict event a_j, the range $(D(a_j), a_j)$ defines a set of events that happened after $D(a_j)$ and led to the conflicts that were resolved into the conflict a_j which, in turn, refuted $D(a_j)$. These events necessarily occurred on levels deeper than $DL(D(a_j))$.

Definition 4 (Refutation Sequence and sub-tree events). *Let a_j be a conflict event with $a_i = D(a_j)$. Then the (possibly empty) sequence of events $a_{i+1}, \ldots, a_{j-1}$ is called the* Refutation Sequence *of a_j and denoted by $\mathbf{RS}(a_j)$. Any event $a_k \in RS(a_j)$ is called a* sub-tree event *of both a_j and a_i.*

Example 3. Consider the conflict event $a_j := (27, c_{110})$ in Fig. 6. For every event a_i that follows decision $D(a_j) = (27, d_{202})$ until (but not including) the conflict c_{110} it holds that $a_i \in RS(a_j)$. Note that the solver can backtrack from deeper levels to level 27 as a result of conflict events. However no event between a_i and a_j occurred on levels smaller or equal to 27. □

The number of resolutions for each variable is bounded from above by the number of sub-tree conflicts that were resolved into the current conflict. However, not all sub-tree conflict clauses resolve into the current refuting conflict. Some of them could be caused by the imperfection of the decision heuristic and are therefore not used at this point of the search. Our goal is to build a scoring system that is based solely on those conflicts that contribute to the resolution of the current conflict clause. In other words, we compute for each variable an *activity score* which reflects the *number of times it was resolved-on in the process of generating the relevant portion of the refutation sequences of recent conflicts*. We *hypothesize that this criterion for activity leads to faster solution times.*

[2] A conflict that occurs at level 0 proves that the instance is unsatisfiable.

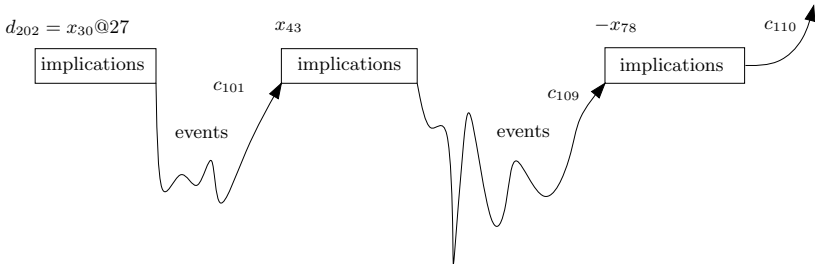

Fig. 6. A possible scenario for the flow of the solver's run. After deciding x_{30} at decision level 27 the solver iteratively goes down to deeper decision levels and returns twice to level 27 with new asserted literals x_{43} and $\overline{x}_{78}$. The latter causes a conflict at level 27 and the solver backtracks to a higher decision level. Implications in the boxes denote assignments that are done during BCP after implying decision or asserting literal.

The information in the colored resolve-graph can enable us to compute such a score.

Definition 5 (Asserting set). *Let $G = (V, E)$ be a colored resolve-graph, and let $v \in V$ be a conflict clause. The* Asserting set *$B(v) \subset V$ of v is the subset of (conflict) clauses that v has a blue path to them in G.*

The following theorem relates between a resolve-graph and sub-tree conflicts.

Theorem 1. *Let e_v be the conflict event that created the conflict clause v. Then the asserting set of v is contained in the refutation sequence of e_v, i.e. $B(v) \subseteq RS(e_v)$. In particular, since conflict events in $B(v)$ participate in the resolution of v by definition, they necessarily correspond to those sub-tree conflicts of e_v that participate in the resolution of v.*

Note that $B(v)$ does not necessarily include *all* the sub-tree conflicts that resolve into v, since the theorem guarantees containment in only one direction. Nevertheless, our heuristic is based on this theorem: it computes the size of the asserting set for each conflict.

In order to prove this theorem we will use the following lemmas.

Lemma 1. *Denote by $stack(a_j)$ the stack of implied literals at the decision level $DL(a_j)$, where a_j is a decision event. Suppose that a literal t is asserted and entered into $stack(a_j)$, where a_j is a decision event. Further, suppose that t is asserted by the conflict clause cl (cl is thus asserting at this point) which was created at event a_i. Then it holds that $j < i$, i.e cl was created after the decision event a_j occurred.*

Proof. Right after the creation of cl, the DPLL algorithm backtracks to some level dl' with a decision event $a_k = (dl', d)$ and implies its asserted literal. It holds that $k < i$, because the solver backtracks to a decision level which already exists when cl is created. By the definition of an asserting clause, cl can be asserting exactly once, and since cl is asserting on level dl', it will never be asserting after

the DPLL algorithm will backtrack from dl'. Therefore it must hold that $a_k = a_j$ $(k = j)$ and $dl' = dl$. □

Lemma 2 (Transitivity of RS). *Suppose that a_i, a_j are conflict events s.t. $a_i \in RS(a_j)$. Then, for any event $a_k \in RS(a_i)$ it follows that $a_k \in RS(a_j)$.*

Proof. First, we will prove that $D(a_i) \in RS(a_j)$, or, in other words, that $D(a_i)$ occurred between $D(a_j)$ and a_j. Clearly, $D(a_i)$ occurred before a_i and, therefore, before a_j. Now, falsely assume that $D(a_i)$ occurred before $D(a_j)$. Then the order of events is $D(a_i), D(a_j), a_i$. However, this can not happen since $D(a_j)$ occurred on shallower (smaller) level than a_i and this contradicts the fact that all events between $D(a_i)$ and a_i occur on the deeper levels. Therefore, both $D(a_i)$ and a_i occurred between $D(a_j)$ and a_j. Now, since a_k happened between $D(a_i)$ and a_i it also happened between $D(a_j)$ and a_j and from this it holds that $a_k \in RS(a_j)$. □

Using this lemma we can now prove Theorem 1.

Proof. We need to show that any blue descendant of v is in $RS(e_v)$. By Lemma 2 it is enough to show it for the immediate blue descendants, since by transitivity of RS it then follows that for any blue descendant. Now, suppose that there exists a blue edge (v, u) in the resolve-graph. By the definition a blue edge, clause u was asserting during the resolution of v. On the one hand, u was resolved during the creation of v and, therefore, was created before v. On the other hand, by Lemma 1 it was created after $D(e_v)$. Therefore, $e_u \in RS(e_v)$. □

Definition 6 (Sub-tree weight of the conflict). *Given a resolve-graph $G(V, E)$ we define for each clause v a state variable $W(v)$:*

$$W(v) = \begin{cases} \sum_{(v,u)\in E} W(u) + 1 & v \text{ is asserting} \\ 0 & \text{otherwise} \end{cases}$$

The function $W(v)$ is well-defined, since the resolve-graph is acyclic. Moreover, since the blue sub-graph rooted at v forms a tree (remember that any node has at most one incoming blue edge), $W(v)$ equals to $|B(v)|+1$. Our recursive definition of $W(v)$ gives us a simple and convenient way to compute it as part of the resolution algorithm. Algorithm 2 is the same as Algorithm 1, with the addition of several lines: in line 5 we add $W \leftarrow 1$, at line 24 we add $W+=W(ResolveCl)$ and, finally, we set $W(NewClause) \leftarrow W$ at line 29. We need to guarantee that $W(C)$ is non-zero only when C is an asserting clause. Therefore, for any antecedent clause C, when its implied variable is unassigned we set $W(C) \leftarrow 0$.

Computing the scores of a variable. Given the earlier definitions, it is now left to show how activity score and sign score are actually computed, given that we do not have the resolve-graph in memory. For each variable v we keep two fields: $activity(v)$ and $sign\_score(v)$. At the beginning of the run $activity$ is initialized to $\max\{lit\_num(v), lit\_num(\overline{v})\}$ and $sign\_score$ to $lit\_num(v) - lit\_num(\overline{v})$. Alg. 2 shows the extended version of the resolution algorithm which

computes the weights of the clauses and updates the scores. Recall that any clause weight is reset to zero when its implied variable is unassigned, so that any clause weight is contributed at most once. In order to give a priority to recent resolutions we occasionally divide both activities and sign scores by 2.

Our decision heuristic chooses a variable from the given clause with a biggest activity and then chooses its value according to the sign score: TRUE for the positive values and FALSE for the negative values of the sign score.

Algorithm 2. First-UIP Learning Scheme, including scoring

```
    procedure AnalyzeConflict(Clause: conflict)
 2:   currentClause ← conflict;
      ResolveNum ← 0;
 4:   NewClause ← ∅;
      wght ← 1;
 6:   repeat
        for each literal lit ∈ currentClause do
 8:       v ← var(lit);
          if v is not marked then
10:         Mark v;
            if dlevel(v) = CurrentLevel then
12:           ++ ResolveNum;
            else
14:           NewClause ← NewClause ∪ {lit};
            end if
16:       end if
        end for
18:     u ← last marked literal on the assignment stack;
        Unmark var(u);
20:     activity(var(u)) += wght;
        sign_score(var(u)) −= wght · sign(u);
22:     −− ResolveNum;
        ResolveCl ← Antecedent(u);
24:     wght+= W(ResolveCl);
        currentClause ← ResolveCl \ {u};
26:   until ResolveNum = 0;
      Unmark literals in NewClause;
28:   NewClause ← NewClause ∪ {ū};
      W(NewClause) ← wght ;
30:   Add NewClause to the clause database;
    end procedure
```

5 Experiments

Figure 7 shows experiments on an Intel 2.5Ghz computer with 1GB memory running Linux, sorted according to the winning strategy, which is CMTF combined with the RBS scoring technique. The benchmark set is comprised of 165

industrial instances used in various SAT competitions. In particular, *fifo8, bmc2, CheckerInterchange, comb, f2clk, ip, fvp2, IBM02 and w08* are hard industrial benchmarks from SAT02; *hanoi* and *hanoi03* participated in SAT02 and SAT03; *pipe03* is from SAT03 and *01_rule, 11_rule_2, 22_rule, pipe-sat-1-1, sat02, vis-bmc, vliw_unsat_2.0* are from SAT04 each instance was set to 3000 seconds. If an instance could not be solved in this time limit, 3000 sec. were added as its solving time. All configurations are implemented on top of HaifaSat, which guarantees that the figures faithfully represent the quality of the various heuristics, as far as these benchmarks are representative. The results show that using CMTF instead of Berkmin's heuristic for choosing a clause leads to an average reduction of 10% in run time and 12-25% in the number of fails (depending on the score heuristic). It also shows a 23% reduction in run time when using RBS rather than VSIDS as a score system, and a corresponding 20-30% reduction in the number of fails. The differences in run times between HaifaSat running the berkmin heuristic and Berkmin561 are small: the latter solves these instances in 210793 sec. and 53 timeouts. We also ran zChaff2004.5.13 on these formulas: it solves them in 210395 sec, and has 53 timeouts.

		BERKMIN+RBS		BERKMIN+VSIDS		CMTF+RBS		CMTF+VSIDS	
Benchmark	instances	time	fails	time	fails	time	fails	time	fails
hanoi	5	389.18	0	530.62	0	130.72	0	74.55	0
ip	4	191.02	0	395.52	0	203.24	0	324.27	0
hanoi03	4	1548.25	0	1342.1	0	426.87	0	386.28	0
CheckerI-C	4	1368.25	0	3323.16	0	681.56	0	3457.78	0
bmc2	6	1731.96	0	1030.9	0	1261.97	0	1006.94	0
pipe03	3	845.97	0	6459.62	2	1339.29	0	6160.12	1
fifo8	4	1877.57	0	3944.31	0	1832.65	0	3382.61	0
fvp2	22	1385.64	0	8638.63	1	1995.17	0	11233.7	3
w08	3	2548.62	0	5347.62	1	2680.96	0	4453.28	0
pipe-sat-1-1	10	1743.23	0	3881.49	0	3310.41	0	6053.84	0
IBM02	9	7083.55	1	9710.52	1	3875.64	0	7163.95	0
f2clk	3	4389.04	1	5135.25	1	4058.62	1	4538.15	1
comb	3	3915.15	1	3681.45	1	4131.05	1	4034.53	1
vis-bmc	8	15284.45	3	7905.9	1	13767.52	3	10119.34	2
sat02	9	17518.09	4	22785.77	5	17329.64	4	21262.25	4
01_rule	20	22742.11	4	33642.33	9	19171.5	2	23689.37	5
vliw_unsat_2.0	8	16600.67	4	24003.62	8	19425.41	5	22756.03	7
11_rule_2	20	31699.69	8	34006.97	10	22974.7	6	28358.05	6
22_rule	20	28844.07	8	33201.87	10	27596.78	8	30669.91	8
Total:	165	161706.5	34	208967.7	50	146193.7	30	189125	38

Fig. 7. A comparison of various configurations, showing separately the advantage of CMTF, the heuristic for choosing the next clause from which the decided variables will be chosen, and RBS, the heuristic for choosing the variable from this clause and its sign

6 Summary

We presented an abstraction/refinement model for analyzing and developing SAT decision heuristics. Satisfying a conflict clause before satisfying the clauses from which it was resolved, can be seen according to our model as satisfying an abstract model before satisfying a more concrete version of it. Our Clause-Move-To-Front decision heuristic, according to this model, attempts to satisfy clauses in an order associated with the resolve-graph. CMTF does not require to maintain the resolve-graph in memory, however: it only exploits the connection between each conflict clause and its immediate neighbors on this graph. Perhaps future heuristics based on this graph will find a way to improve the balance between the memory consumption imposed by saving this graph and the quality of the decision order. We also presented a heuristic for choosing the next variable and sign from the clause chosen by CMTF. Our Resolution-Based-Scoring heuristic scores variables according to their involvement ('activity') in refuting recent decisions. Our experiments show that CMTF and RBS either separately or combined are better than Berkmin and the VSIDS decision heuristics.

Acknowledgments

We thank Maya Koifman for helpful comments on an earlier version of this paper.

References

1. P. Beame., H. Kautz, and A. Sabharwal. Towards understanding and harnessing the potential of clause learning. *Journal of Artificial Intelligence Research*, 22:319–351, 2004.
2. E. Goldberg and Y. Novikov. Berkmin: A fast and robust sat-solver. In *Design, Automation and Test in Europe Conference and Exhibition (DATE'02)*, page 142, Paris, 2002.
3. L.Ryan. Efficient algorithms for clause-learning SAT solvers. Master's thesis, Simon Fraser University, 2004.
4. M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *Proc. Design Automation Conference (DAC'01)*, 2001.
5. J.P.M. Silva and K.A. Sakallah. GRASP - a new search algorithm for satisfiability. Technical Report TR-CSE-292996, Univerisity of Michigen, 1996.
6. L. Zhang, C. Madigan, M. Moskewicz, and S. Malik. Efficient conflict driven learning in a boolean satisfiability solver. In *ICCAD*, 2001.
7. L. Zhang and S. Malik. Extracting small unsatisfiable cores from unsatisfiable boolean formula. In *Theory and Applications of Satisfiability Testing*, 2003.