A Clause-Based Heuristic for SAT Solvers

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

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

Abstract. We propose a new decision heuristic for DPLL-based propositional SAT solvers. Its essence is that both the initial and the conflict clauses are arranged in a list and the next decision variable is chosen from the top-most unsatisfied clause. Various methods of initially organizing the list and moving the clauses within it are studied. Our approach is an extension of one used in Berkmin, and adopted by other modern solvers, according to which only conflict clauses are organized in a list, and a literal-scoring-based secondary heuristic is used when there are no more unsatisfied conflict clauses. Our approach, implemented in the 2004 version of zChaff solver and in a generic Chaff-based SAT solver, results in a significant performance boost on hard industrial benchmarks.

1 Introduction

Propositional satisfiability (SAT) is the problem of determining, for a formula in the propositional calculus, whether there exists a satisfying assignment for its variables. This problem has numerous applications in Formal Verification (e.g., [14]), as well as in Artificial Intelligence (e.g., [8]). SAT solvers are widely used in these and other domains.

Modern complete SAT solvers (e.g., Chaff [10,13], Berkmin [5], Siege [15]) are based on the backtrack-search algorithm of Davis, Putnam, Loveland and Logemann (*DPLL*) [3]. A crucial factor influencing the performance of a DPLL-based SAT solver is its decision heuristic. This heuristic decides which variable to choose at each decision point during the search and what value to assign it first. This paper introduces a new decision heuristic that has been found to be efficient on real-world hard industrial benchmarks. It is designed to increase the likelihood that interrelated variables will be chosen in proximity.

In recent years, the field has been a witness to a breakthrough in the design of decision heuristics that are empirically successful on real-world industrial SAT instances. The key observation was that the decision heuristic must be dynamic, that is, it must re-focus the search on recently derived conflict clauses. VSIDS [13]—the first such dynamic heuristic—maintains a score for each literal. The score is increased when

the literal appears in a conflict clause; once in a while, scores are halved. This strategy ensures that the prover picks literals that were involved in the derivation of recent conflict clauses.

Another well-known decision heuristic, which proved to be even more successful than VSIDS on benchmark industrial instances, is Berkmin's [5]. Its authors claimed that VSIDS is not sufficiently dynamic, in the sense that it may still pick literals that are irrelevant to the currently explored branch. Instead, they proposed organizing all conflict clauses in a list and picking the next decision literal from the top-most unsatisfied clause on the list. If no such clause exists, a secondary VSIDS-like choice-heuristic is used. Berkmin's heuristic is indeed more dynamic than VSIDS, but we find another advantage of Berkmin's heuristic over VSIDS, in that it tends to pick interrelated variables, that is, variables whose joint assignment increases the chances of both quickly reaching a conflict in an unsatisfiable branch, as well as satisfying and removing "problematic" clauses in satisfiable branches. However, this potential advantage is diluted by the fact that Berkmin does not put the initial clauses on the clause list and applies a secondary VSIDS-like heuristic.

Our proposal, which we call the *clause-based heuristic* (*CBH*), maintains a clause list containing both the initial and the conflict clauses, thus increasing the chances of picking interrelated variables. The next decision literal is picked from the top-most unsatisfied clause. No secondary heuristic is required. We propose various methods of initially organizing the clause list and for moving clauses within it. Our approach results in a significant performance boost over both VSIDS and Berkmin's heuristic.

Basic definitions are provided in the next section. In Section 3, we review the DPLL algorithm [3], enhanced by Boolean constraint propagation [17] and conflict clause recording [12]. Readers familiar with this material—included in the paper to make it self-contained—may skip ahead. Some of the commonly used decision heuristics are summarized in Section 4. In Section 5, we present our clause-based heuristic. In Section 6, we report on experiments that show that CBH is superior to VSIDS, Berkmin's decision heuristic and the Berkmin-like decision heuristic of zChaff-2004, when tested on hard industrial benchmarks used in the SAT'04 competition [9]. This is followed by a brief conclusion.

2 Basic Definitions

In what follows, we denote negation by $\neg$, conjunction by $\wedge$ and disjunction by $\vee$. A *literal* is a variable or its negation. We use p, q, r, etc., for literals. A *clause* is a disjunction of literals, usually denoted by letters A, B, ... The number of literals in a clause A is denoted by $|A|$. A *conjunctive normal form (CNF)* formula is a conjunction of clauses, like $\varphi = (p) \wedge (\neg p \vee q) \wedge (\neg q \vee \neg r)$. It is sometimes convenient to represent a CNF formula as a set of clauses, rather than as a conjunction. For example, an alternative way to represent φ is as the set $\{ p, \neg p \vee q, \neg q \vee \neg r \}$. We denote an empty clause by $\varnothing$.

An (*partial*) *assignment* (σ) is a (partial) function assigning a truth value, 1 or 0, to all (some) variables. An example is the assignment $\tau(p) = \tau(r) = 1$, $\tau(q) = 0$. It is sometimes convenient to consider a (partial) assignment as a subset of literals. For

example, an alternative way to represent τ is as $\{p, \neg q, r\}$. A CNF formula φ evaluates under partial assignment σ to $\sigma[\varphi]$ by applying the following rules for each variable p:

1. If $\sigma(p) = 1$, remove $\neg p$ from φ's clauses and remove clauses containing p from φ.
2. If $\sigma(p) = 0$, remove p from φ's clauses and remove clauses containing $\neg p$ from φ.

If $\sigma[\varphi]$ contains no clauses, then $\sigma[\varphi] = 1$ and σ is a *satisfying assignment* or a *model* for φ, and we say that $\varphi=1$ *under* σ. If $\varnothing \in \sigma[\varphi]$, then σ is an *unsatisfying assignment* for φ, and we say that $\varphi=0$ *under* σ. Formula φ is *satisfiable* if there exists an assignment σ, such that $\varphi=1$ under σ; otherwise, it is *unsatisfiable.*

3 Enhanced DPLL

The DPLL algorithm [3] is the basic backtrack-search algorithm for SAT. DPLL is based upon the validity of the following two rules:

1. *Splitting Literal Rule*: Let $\sigma = \{p\}$ and $\tau = \{\neg p\}$. Formula φ is satisfiable iff $\sigma[\varphi]$ or $\tau[\varphi]$ is satisfiable.
2. *Unit Clause Rule:* Let $\sigma = \{p\}$. If there is a clause $A \in \varphi$ such that $A = p$, then φ is satisfiable iff $\sigma[\varphi]$ is.

Boolean constraint propagation (BCP) [17] is a process of extending a partial assignment by repeatedly applying the unit-clause rule. A *decision variable* is a variable assigned as a result of an application of the splitting rule. A *decision literal* is the assigned literal of a decision variable. An *implied variable* is a variable assigned as a result of the BCP process. An *implied literal* is the assigned literal of an implied variable. The *decision level* of an assigned variable p is one more than the number of decision variables assigned prior to p.

Let φ be an input formula. Modern complete SAT solvers implement the DPLL algorithm in the following way: A partial assignment σ, initialized to the empty partial assignment, is maintained. The partial assignment σ is extended by executing the BCP process. Then, if the input formula φ is neither 1 nor 0 under σ, the splitting rule is used, that is, the satisfiability of φ is recursively checked under $\sigma \cup \{p\}$ and then under $\sigma \cup \{\neg p\}$, as required. The literal p is picked using some *decision heuristic.*

Next we describe a powerful technique, used by all the modern SAT solvers, namely *conflict clause recording* [12]. A *conflict* is a situation during DPLL invocation when a formula φ evaluates to 0 under the current assignment σ. After such a conflict, it is possible to identify an assignment $\tau \subseteq \sigma$, such that if χ is an assignment containing τ, extended by invoking BCP on $\tau[\varphi]$, then φ is 0 under χ. In other words, applying τ to φ is sufficient to create the same conflict as applying s to φ up to BCP. We refer to such a partial assignment τ as a *conflict-sufficient assignment.* Let τ be a conflict-sufficient assignment. Then a clause A consisting of the negations of all the literals contained in τ is referred to as a *conflict clause*. The process of adding conflict clauses to the formula after each conflict is referred to as *conflict clause recording*. Conflict-clause recording prevents DPLL from re-exploring the same sub-

space during the subsequent search. Different *conflict recording schemes* add conflict clauses corresponding to different conflict-sufficient assignments. The 1UIP conflict recording scheme [12,13] has empirically been shown to be the most efficient [15,18]. In what follows, we describe the 1UIP conflict clause identification.

First, we define an "implication clause" for an assigned literal p. Let σ be a partial assignment held by DPLL prior to assigning p. The *implication clause* of an implied literal p, denoted by $ic(p)$, is the clause that became a unit under σ and made the implication of p possible, during BCP. The implication clause of a decision literal p is the empty clause $\emptyset$. After a conflict, a *conflicting variable* r is the variable that had been assigned last, before it was identified that σ is an unsatisfying assignment. Both literals of a conflicting variable are referred to as *conflicting literals*. According to the definition of an unsatisfying assignment, an empty clause must belong to $\sigma[\varphi]$. It means that there is a clause $F \in \varphi$ that is 0 under σ. One of the conflicting literals must belong to F, otherwise r would not have been the last variable that was assigned a value. We denote a conflicting literal that belongs to F by $ecl(r)$. For example, if $\varphi = (p \vee r) \wedge (p \vee \neg r)$; $\sigma = \{\neg p, r\}$, then $\sigma[\varphi] = 0$; $p \vee \neg r$ is the 0-clause under σ; r is the conflicting variable and $ecl(r) = \neg r$. Observe that the implication clause is well defined for $\neg ecl(r)$, since $\neg ecl(r)$ is an assigned literal. However, the implication clause is not defined for $ecl(r)$, since $ecl(r)$ is not an assigned literal. We extend the definition of the implication clause by setting $ic(ecl(r)) = F$. Note that by the extended definition, the conflicting variable has two implication clauses (one for each literal.)

An *implication graph* is a directed acyclic graph, in which each vertex is associated with a literal corresponding to an assignment along with its decision level. There is a directed edge from vertex p to vertex q if $ic(q) = A$ and $\neg p \in A$. Consider the example in Fig. 1, which illustrates concepts presented in this section. DPLL has been invoked on formula φ and the situation at the time of the first conflict is shown in Table 1. The implication graph is shown at the bottom. The decision literals $\neg y$, $\neg p$ and $\neg s$ do not have incoming edges. Any implied literal y has $|\ ic(y)\ | - 1$ incoming edges.

Now we are ready to describe the concept of conflict recording schemes. Any conflict clause is generated by a *conflict cut* of the implication graph. All the decision literals are on one side of the conflict cut (called the *reason side*). Both conflicting literals are on the other side of the conflict cut (called the *conflict side*). All vertices on the reason side that have at least one edge to the vertices of the conflict side comprise the conflict's *reason*. Let $\tau \subseteq \sigma$ be a partial assignment containing only the assigned literals corresponding to the reason of a conflict cut. The partial assignment τ is a conflict-sufficient assignment.

Next, we describe the *1UIP learning scheme*. Let p and q be two literals assigned at the same decision level dl. Then, in an implication graph, p is said to *dominate* q, if and only if any path from the decision variable corresponding to level dl to q contains p. A *unique implication point* (*UIP*) [12] is an assigned literal of the highest decision level dominating both literals of the conflicting variable. Observe that the decision literal corresponding to the highest decision level is always a UIP. The 1UIP learning scheme requires that any literal assigned after the first UIP (counting from the conflicting literals) will be on the conflict side of the corresponding conflict cut and all others will be on the reason side. Consider again the example in Fig. 1. The literals t

and $\neg s$ are UIP's, since both dominate x and $\neg x$. The literal t is the first UIP, since it is closer than $\neg s$ to the conflicting literals. The 1UIP cut is shown on Fig. 1. The corresponding conflict clause is $\neg q \vee \neg t$.

$$A_1 = \neg w \vee s \vee v \vee \neg q;\ A_2 = p \vee q;\ A_3 = \neg q \vee r \vee s \vee \neg w;\ A_4 = \neg q \vee \neg t \vee x; \quad \textbf{(1)}$$
$$A_5 = s \vee \neg r \vee \neg v \vee t;\ A_6 = \neg t \vee \neg x;\ A_7 = y \vee w;$$
$$\varphi = A_1 \wedge A_2 \wedge A_3 \wedge A_4 \wedge A_5 \wedge A_6 \wedge A_7.$$

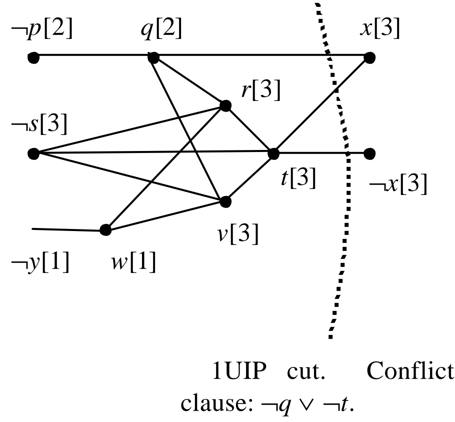

Fig. 1. An example of an implication graph and 1UIP conflict clause identification. The implication graph corresponds to a DPLL invocation on Formula (1). The related variable values, implication clauses and decision level of the variables are shown on Table 1

Table 1. The implication graph and 1UIP conflict clause identification principles. The value, decision level and implication clause for each variable of formula (1) are shown. The corresponding implication graph and 1UIP conflict cut are displayed in Fig. 1

	p	q	r	s	t	v	w	x	y
σ	0	1	1	0	1	1	1	1	0
DL	2	2	3	3	3	3	1	3	1
ic	∅	2	3	∅	5	1	7	4	∅

4 Existing Decision Heuristics

In this section, we describe the most widely used decision heuristics, known to be efficient on real-world industrial benchmarks.

Early *static* heuristics (e.g. Jeroslaw-Wang [7], Literal Count [11]) picked the next variable based on the number of appearances (scores) of different variables in unsatisfied clauses. A major drawback of such an approach is that score calculation requires visiting all the clauses at each node of the search tree, which implies a very significant

overhead. Another disadvantage of static heuristics is that they do not consider information that can be retrieved after a conflict during implication graph analysis. Heuristics based upon such analyses were found to be several orders of magnitude faster [5,13].

The first *dynamic* heuristic is called Variable State Independent Decaying Sum (VSIDS) [13]. According to VSIDS, each literal is associated with a counter $cl(p)$, whose value is increased once a new clause containing p is added to the database. Counters are initialized to 0. Every once in a while, all counters are halved. The next literal to be picked is the one with the largest counter. Ties are broken randomly. Two major advantages of VSIDS over the previous heuristics are that: (1) VSIDS is characterized by a negligible computational cost; (2) VSIDS gives preference to literals that participate in recent conflicts, i.e. it is dynamic. MiniSat SAT solver [4] implements a variant of VSIDS. Instead of infrequent halving of the scores, MiniSat multiplies the scores after each conflict by 0.95. This makes the heuristic more dynamic.

The authors of the Berkmin SAT solver [5] proposed a successful decision heuristic that has been partially or fully adopted by the most modern SAT solvers, such as the 2004 version of zChaff [10], Satzoo [4], and Oepir [1]. We show, in the experimental section, that Berkmin's heuristic is indeed faster than VSIDS on hard industrial benchmarks. The main difference of Berkmin's heuristic, when compared to VSIDS, is as follows: Conflict clauses are organized in a list, and every new conflict clause is appended to the head of the list. The next decision variable is picked from the top-most unsatisfied clause. If no such clause exists, the next decision variable is chosen according to a VSIDS-like heuristic. We will describe Berkmin's heuristic in detail and analyze why it is advantageous over VSIDS.

Berkmin maintains a counter $cl'(p)$ measuring the contribution of each literal to the search. Unlike VSIDS, Berkmin augments $cl'(p)$, not only for literals that belong the conflict clause itself, but also for literals that belong to one of the clauses that were traversed in the implication graph during 1UIP conflict clause identification. At intervals, Berkmin divides all the counters by 4 (compared to 2 for VSIDS). Let $cv'(p)$ be a counter measuring the contribution of each variable to the conflicts, defined as $cl'(p) + cl'(\neg p)$. Berkmin maintains all conflict clauses in a list. After each conflict, the new conflict clause is appended to the top of the list. The next decision variable is one with the highest $cv'(p)$ out of all the variables of the topmost unsatisfied clause. If no conflict clauses exist yet, or if all the conflict clauses are satisfied, then the variable with the highest $cv'(p)$ of all unassigned variables is chosen.

Next, we describe how Berkmin decides which literal, out of the two possible literals of the already chosen variable, to pick. Berkmin maintains a counter $gcl(p)$, measuring the global contribution of each literal to the conflicts. The counter $gcl(p)$ is initialized to 0 and is increased whenever $cl'(p)$ is increased, but is not divided by a constant. If a topmost unsatisfied clause exists, Berkmin picks a literal with the highest global score $gcl(p)$. Ties are broken randomly. If there is no unsatisfied topmost clause, then Berkmin picks the literal with the highest value of $two(p)$, where $two(p)$ approximates the number of binary clauses in the neighborhood of literal p. The function $two(p)$ is computed as follows: First, the number of binary clauses containing p is calculated. Then, for each binary clause B, containing p, the number of binary clauses containing $\neg q$ is computed, where q is the other literal of B. The sum of all computed

numbers gives the value of $two(p)$. To reduce the amount of time spent computing $two(p)$, a threshold value of 100 is used. As soon as the value of $two(p)$ exceeds the threshold, its computation is stopped. Once again, ties are broken randomly.

The most important advantage of Berkmin's approach over VSIDS, as stated by the authors of Berkmin, is its additional dynamicity. It quickly adjusts itself to reflect changes in the set of variables relevant to the currently explored branch. Indeed, Berkmin picks variables from fresh conflict clauses and thus uses very recent data. Our understanding is that there is another important advantage of Berkmin's heuristic over VSIDS: newly assigned variables tend to embrace more interrelated variables. By "interrelated," we mean variables whose joint assignment increases the chances of both quickly reaching a conflict in an unsatisfiable branch and satisfying out "problematic" clauses in satisfiable branches. According to Berkmin's heuristic, a series of new decision variables appear in recent conflict clauses. Hence, these variables were recently traversed during conflict analysis and consequently contributed to conflict derivation. Moreover, even if the top-most conflict clauses were recorded a long time ago, the fact that their variables appeared closely together during conflict analysis, hints that they are interrelated. However, the impact of this advantage is diluted by the fact that Berkmin does not put the initial clauses on the list, but instead uses VSIDS as a secondary heuristic. The novel CBH heuristic, described in the next section, takes advantage of this observation.

5 The Clause-Based Heuristic

In our clause-based heuristic (CBH), all clauses (both the initial and the conflict clauses) are organized in a list. After each conflict, the conflict clause is prepended to the top of the list. *Conflict-responsible* clauses, that is, clauses visited during 1UIP conflict-clause identification, are placed just after the new conflict clause. The next decision literal is picked from the topmost unsatisfied clause of the list. One can see that CBH is highly dynamic, since recently visited clauses are placed at the top of the list. Also, CBH organizes the list in such way that clauses that were responsible for a recent conflict are placed together. Hence, when one picks a series of decision variables after backtracking, it will tend to embrace interrelated variables. Indeed, when literals are picked from the same clause they must be related, even if the clause is an initial clause. When literals are picked from closely-placed clauses, they also tend to be related, since the list is organized in such a way that interrelated clauses are near each other, by placing conflict clauses at the top and moving conflict-responsible clauses towards the top.

As a variant, CBH can also move clauses found to have exactly two unassigned literals during BCP to the top of the list. We refer to this strategy as *2LitFirst*. The added value of this strategy is that: (1) more implications are learned during BCP; (2) short and potentially contradictory clauses tend to be immediately satisfied. The first point guides the solver to find conflicts in an unsatisfiable area, and the second one is useful to eliminate conflicts in a satisfiable area. The disadvantages of *2LitFirst* are that: (1) it tends to separate between clauses that contain interrelated variables; (2) it may promote clauses that have never been responsible for conflicts.

We found experimentally that while usually *2LitFirst* hurts performance, it may be helpful for instances having high clause/variable ratio. This can be explained by the fact that, in instances having a high clause/variable ratio, variables tend to appear in a greater number of clauses, since there are fewer variables per clause overall. Hence, two chains of decisions taken using different decision strategies tend to contain more common variables. This gives more weight to the order between variables and the local context of the search. One should prefer variables whose assignment can have an immediate impact; this is exactly what *2LitFirst* does. The default version of CBH invokes *2LitFirst* on instances where the clause/variable ratio exceeds 10.

One can see that the major differences of CBH comparing with Berkmin's heuristic are the following:

(1) Both the initial and the conflict clauses, rather than only conflict clauses, are organized in a list; therefore, a second choice heuristic is not required. Thus, any set of decision variables picked by CBH tends to contain more variables from the same clause.

(2) After a conflict, in addition to the conflict clause, a number of clauses responsible for the conflict (including initial clauses) are moved towards the head of the list. Thus, clauses that are placed nearby are likely interrelated.

(3) As a variant, clauses that were discovered to have two unassigned literals are moved towards the top of the list.

CBH can be easily implemented using a doubly-linked list. A pointer to the currently watched clause *C*, initialized to the top-most clause, is maintained. When a decision is required, we seek the top-most unsatisfied clause *A*, starting from *C* towards the bottom of the list, and pick a literal from *A* (as described in Section 5.1). Observe that if no top-most unsatisfied clause exists, then we have a satisfying assignment, since all the clauses, including the original ones are satisfied. After each conflict, the solver updates the clause list and sets the currently watched clause to point to the top of the list.

The next subsection explains how CBH chooses the decision literal from the top-most unsatisfiable clause. Subsection 5.2 explains the initial organization of the clause list. Subsection 5.3 describes conflict-responsible clause identification in greater detail.

5.1 Choosing the Decision Literal from the Top-Most Clause

CBH maintains two counters, $lcl(p)$ and $gcl(p)$, measuring the local and global contributions of each literal to the conflicts, respectively. The counter $lcl(p)$ is initialized to 0 for each p, while $gcl(p)$ is initialized with the number of p's appearances in initial clauses. Both counters are incremented whenever a literal belongs to one of the clauses traversed in the implication graph during 1UIP conflict-clause identification. Occasionally, the value of $lcl(p)$ is divided by 2.

CBH also maintains two counters for variables $lcv(p)$ and $gcv(p)$, measuring the contribution of each variable to the conflicts. We define:

$$lcv(p) = (lcl(p) + lcl(\neg p)) + 3 \cdot min(lcl(p), lcl(\neg p)). \qquad (2)$$

The first term gives preference to variables for which both literals are important, and the second term eliminates variables where only one literal is important. In a similar manner, we have:

$$gcv(p) = (gcl(p) + gcl(\neg p)) + 3 \cdot min(gcl(p), gcl(\neg p)). \quad (3)$$

CBH chooses the decision variable from the topmost unsatisfied clause using the following algorithm: A variable p with maximal $lcv(p)$ is chosen, so as to give preference to variables that participated in recent conflicts. Ties are broken by preferring variables with maximal global score $gcv(p)$. According to the next criterion, variables that used to have the maximal decision level when assigned the last time are preferred. (If there still is a tie, it is broken by picking the lexicographically smallest variable.)

CBH chooses the decision literal out of the two possible, based on the global contribution value $gcl(p)$.

5.2 Initial Clause-List Organization

In general, we aim to:

(1) place clauses containing frequently appearing literals near the top of the list, and

(2) place clauses containing common literals nearby.

Point (1) guides the solver to start the search using frequent literals, and the second point increases the chances of picking interrelated literals.

First, the initial global score $igs(p)$ is calculated for each literal p. The function $igs(p)$ is initialized to 0 and is augmented for each clause that contains the literal p. The initial global score reflects the overall frequency of a literal.

In the process of clause list construction, we also maintain the initial bcal score $ils(p)$ for each literal p. It is calculated similarly to $igs(p)$, except that only clauses that were already placed on the clause list are considered. The local score reflects the involvement of p in clauses that have been already appended to the clause list. Initially, no clauses are included in the clause list, hence $ils(p)$=0 for each literal p. We also define the initial overall score $ios(p) = igs(p) + ils(p)$ for each literal p. The initial overall score takes into consideration both the local and global influence of each literal.

So far, we have defined three functions for each literal reflecting its global, local and overall influence. Now, we can define the initial variable overall score for each variable p:

$$iosv(p) = (ios(p) + ios(\neg p)) + 3 \cdot \min(ios(p), ios(\neg p)). \quad (4)$$

The clause list is constructed by repeating the following procedure until all the clauses are placed on the clause list: Let p be the variable having the maximal variable overall score amongst all variables that have not already been picked. (Ties are broken by preferring the smaller variable according to lexicographical order.) We append clauses containing the variable p that have not yet been appended to the end of the

clause list. Local and overall scores are updated for each literal participating in clauses that have been appended to the list. Such dynamic update of scores does not require any overhead, given that we use a priority queue, indexed by the scores. Literals can be moved within the queues in constant time.

5.3 Conflict-Responsible Clauses

Recall that after a conflict, conflict-responsible clauses are placed at the head of the list, after the new conflict clause. A clause is responsible for a conflict (in the context of CBH) if it appears in the implication clause of either a literal that appears on the conflict side of the 1UIP cut, or else of a literal whose negation appears in the 1UIP conflict clause.

Consider the example in Fig. 1. Clauses A_4, A_6, A_2 and A_5 are responsible for the conflict. Indeed, clauses A_4 and A_6 are the implication clauses of the two literals x and $\neg x$ that appear on the conflict side of the 1UIP cut, and clauses A_2 and A_5 are the implication clauses of the literals q and t, whose negation appear in the conflict clause.

From a practical point of view, it is not hard to identify clauses that are responsible for the conflict, since these are all the clauses that were visited during 1UIP conflict-clause construction.

6 Experimental Results

We implemented CBH within two SAT solvers. The first is the newest version of the famous zChaff solver, namely zchaff_2004.11.15 [10]. zChaff won first place in the Industrial-Overall category of the SAT'04 competition [9]. This new version of zChaff implements Berkmin's heuristic, in contrast with the old version of zChaff [13] which used VSIDS. The performance of zChaff was measured on a machine with 4Gb of memory and two Intel® Xeon™ CPU 3.06GHz processors with hyper-threading. The second solver we used in our experiments is SE—a Chaff-like solver, implementing 1UIP conflict-clause recording [13], non-chronological backtracking [12], frequent search restarts [5,6,10] and aggressive clause deletion [2,5,10] strategies. We designed SE to implement the aforementioned enhancements of DPLL, since they play a crucial role for the performance of a modern SAT solver, tuned for industrial benchmarks, as explained in papers on Berkmin [5] and the newest zChaff [10]. We did not check whether CBH boosts performance when one or more of the above mentioned techniques is not used. The performance of SE was measured on a stronger machine with 4Gb of memory and two Intel® Xeon™ CPU 3.20GHz processors with hyper-threading.

In what follows, we first analyze the overall performance of CBH versus Berkmin's heuristic, VSIDS and Berkmin-like new zChaff's heuristic. We show that CBH outperforms both Berkmin's heuristic and VSIDS within the SE SAT solver, and also that CBH significantly outperforms zChaff's new Berkmin-like heuristic. Then, we analyze how various strategies used by CBH contribute to its performance. We tested CBH inside two solvers to ensure that its measured impact on performance is inde-

pendent of the implementation details of a particular solver. The main measure for success in our experiments is the number of solved instances within an hour on hard industrial families used during SAT'04 competition [9]. We find this measure, which was used during the SAT'04 competition, more convincing than a comparison of the number of decisions or conflicts (omitted for lack of space), since reducing the running time is the final goal of any practical heuristic. Our experiments required approximately 35 days of computation.

Table 3 compares the performance of CBH, VSIDS, VSIDSM—MiniSat-like VSIDS with frequent scores decay and Berkmin's heuristic, implemented in SE, on eight hard industrial families used during the SAT'04 competition (downloaded from the competition's website [16]). The description of these families is provided in Table 2. VSIDSM multiplies the score by 0.95 after every 10 conflicts, rather than with each conflict. The latter rate is used within MiniSat, but the former is preferable within SE. Other heuristics decay the scores each 6000 conflicts. CBH solves at least as many instances within each family, when compared to either Berkmin's heuristic or either version of VSIDS. CBH solves more instances than both version of VSIDS in 7 out of 8 cases, and solves more instances than Berkmin's heuristic in 5 out of 8 cases.

Table 4 shows the performance of CBH within the new version of zChaff. The performance of a version of CBH that does not use *2LitFirst* is also provided. One can see that zChaff, enabled with CBH, outperforms zChaff in a very convincing manner for 6 out of 8 families and is inferior in only once case. Moreover, if the *2LitFirst* strategy is not used, CBH is never inferior.

One can conclude that CBH definitely contributes to modern SAT-solver performance, outperforming both VSIDS and Berkmin's heuristic. Our experiments also confirm that Berkmin's heuristic is preferable to VSIDS, though the gap narrows if VSIDS uses frequent scores decay. This is to be expected, but—to the best of our knowledge—has never been reported, despite the fact that Berkmin's heuristic has been partially or fully adopted by the most modern SAT solvers [1,4,5,10].

What remains is to analyze the performance of CBH when disabling some of its specific strategies. Accordingly, we consider CBH_NM, a version that does not move conflict-responsible clauses to the top of the list (but still appends the conflict clause itself to the top of the list), and CBH_NI, which does not use the initial strategy, described in Section 5.2, but rather appends all clauses to the list in the order of appearance in the input instance. We also experimented with CBH_2L_A, which always uses *2LitFirst*, and with CBH_2L_N, which never does. Table 5 and 6 compare the performance of CBH within SE and zChaff respectively.

Switching off the initial strategy results in a performance degradation for 3 families within SE, and in performance gain for one family. In zChaff, switching off the initial strategy resulted in performance degradation for 2 families. In general, the initial strategy improves the performance within both SE and zChaff, although it is not the most crucial factor contributing to CBH's performance. Even if the initial strategy is switched off, CBH performs better than other decision heuristics. This can be explained by the fact that during the search, CBH quickly reorganizes the clause list to contain groups of interrelated clauses.

Switching off the moving of conflict-responsible clauses to the top of the list results in a performance degradation for 4 families and performance gain on 3 families

in zChaff. The overall number of solved instances is higher when the strategy is switched on. Switching off the moving of conflict-responsible clauses to the top of the list leads to mixed results in the case of SE. Performance seriously degrades for the SCH family, and also for the ST2 family; however, there is a performance-boost for the GR, ST2B and VUN families. The overall number of solved instances remains the same. One can conclude that moving the conflict-responsible clauses to the top of the list can be useful for some families, but hurt others. We recommend invoking it by default, since it results in an overall performance boost in the case of zChaff and does not hurt overall performance of SE.

Table 2. Description of the hard industrial benchmark families used in our experiments. Family name, number of instances in each family as well as the average, and maximal and minimal clause/variable ratios are provided

Abbreviation	*Family Name*	*Inst. Num.*	*Cls/Var Avrg*	*Cls/Var Max*	*Cls/Var Min*
HEQ	goldberg03-hard_eq_check	13	6.4	6.7	6.1
GR	maris03-gripper	10	9.1	9.7	8.5
SCH	schuppan03-l2s	11	3.2	3.3	3.0
ST2	simon03-sat02	9	3.3	4.2	2.7
ST2B	simon03-sat02bis	10	23.9	71.9	2.9
CLR	vangelder-cnf-color	12	42.9	195.4	4.0
PST	velev-pipe-sat-1-1	10	33.7	33.8	33.7
VUN	velev-vliw_unsat_2.0	8	15.6	20.1	10.7

Table 3. Performance of CBH vs. two versions of VSIDS and Berkmin's heuristic, implemented in the SE SAT solver, on eight hard industrial families. The first column contains an abbreviated family name. Each pair of subsequent columns is dedicated to a specific heuristic,. The number of instances, solved within an hour, is provided

Family	*CBH solved*	*Berkmin heur.*	*VSIDSM*	*VSIDS*
HEQ	5	4	4	3
GR	1	1	0	1
SCH	5	2	2	0
ST2	5	4	4	2
ST2B	2	2	2	1
CLR	6	4	4	4
PST	10	10	4	5
VUN	4	2	1	0
ALL	***38***	***29***	***21***	***16***

Regarding the impact of *2LitFirst* strategy, first observe that according to Tables 5 and 6, invoking *2LitFirst* on every instance does not pay off. Note that the default strategy used by CBH invokes *2LitFirst* if the clause/variable rate is greater than 10. The motivating experimental observation for designing CBH in this manner is that SE without *2LitFirst* performs the same as the default version on all families, except VUN, where performance seriously degrades. VUN is the only family, other than PST, for which the clause/variable ratio is greater than 10 for all instances. To con-

firm that SE performs better when *2LitFirst* is invoked on instances with high clause/variable ratio, we launched SE on 26 handmade families that were submitted to SAT'04 (and are available at [16]). SE was able to solve at least 1 instance from 13 families. The default CBH strategy performed better than a strategy with *2LitFirst* disabled on 2 out of 13 families, and it performed the same on other families. In the case of zChaff, we found that *2LitFirst* invocation hurts performance in a dramatic manner on families with low clause/variable ratio, and leaves the performance the same or slightly degraded on families having high clause/variable ratio. The families HEQ, GR, SCH and ST2 are those with a ratio for all its instances lower than 10. If *2LitFirst* is always used, zChaff is able to solve only 4 instances of these 4 families, comparing to 20 instances solved by the version that never uses *2LitFirst*. The other 4 families have either mixed or high clause/variable ratio. On these families, when *2LitFirst* is always used, zChaff is able to solve 12 instances, compared to 16 by a version that never uses *2LitFirst*. One can conclude that within zChaff, *2LitFirst* usage is not justified. Overall, *2LitFirst* performs much better on instances having a high clause/variable ratio.

Table 4. CBH vs. the default heuristic within zChaff_2004.11.15. CBH_2L_N is a version of CBH that does not use the *2LitFirst* strategy

Family	*zChaff + CBH*	*zChaff+ CBH_2L_N*	*zChaff default*
HEQ	8	8	4
GR	3	3	0
SCH	5	5	2
ST2	4	4	1
ST2B	2	2	1
CLR	3	4	1
PST	5	8	8
VUN	2	2	2
ALL	***32***	***36***	***19***

Table 5. Performance of different configurations of CBH in terms of solved instances within an hour in SE solver

Family	*CBH*	*CBH_NM*	*CBH_NI*	*CBH_2L_A*	*CBH_2L_N*
HEQ	5	4	5	3	5
GR	1	2	2	0	1
SCH	5	2	4	5	5
ST2	5	4	5	2	5
ST2B	2	3	1	2	2
CLR	6	7	5	5	6
PST	10	10	10	10	10
VUN	4	6	4	4	1
ALL	***38***	***38***	***36***	***31***	***35***

Table 6. Performance of different configurations of CBH in terms of solved instances within an hour in zChaff solver

Family	*CBH*	*CBH_NM*	*CBH_NI*	*CBH_2L_A*	*CBH_2L_N*
HEQ	8	4	8	4	8
GR	3	1	3	0	3
SCH	5	2	4	0	5
ST2	4	2	4	0	4
ST2B	2	3	2	2	2
CLR	3	3	3	3	4
PST	5	10	3	5	8
VUN	2	3	2	2	2
ALL	***32***	***28***	***29***	***16***	***36***

7. Conclusions

We have presented a novel clause-based heuristic, CBH. It maintains a clause list organized in a manner that allows the algorithm to choose sequences of interrelated variables that were responsible for recent conflict derivation.

CBH maintains both the initial and the conflict clauses in a single list. The next decision literal is picked from the topmost unsatisfied clause of the list. After each conflict, the conflict clause is prepended to the top of the list. Clauses visited during conflict-clause identification, are placed just after the new conflict clause. As a variant, if the clause/variable ratio of the input instance is greater than a predefined value (10 is a reasonable choice), newly identified binary clauses are moved to the top of the list.

We have demonstrated that using CBH results in a significant performance boost on hard industrial families, when compared with Berkmin's heuristic or VSIDS.

Acknowledgements

The work of the third author was carried out in partial fulfillment of the requirements for a Ph.D. We thank Ranan Fraer for his careful reading and helpful comments.

References

[1] J. Alfredsson. The SAT solver Oepir. URL http://www.lri.fr/~simon/contest/results/ONLINEBOOKLET/OepirA.ps (viewed January, 16 2005).

[2] R. Bayardo, Jr., and R. Schrag. Using CSP look-back techniques to solve real-world SAT instances. In Proceedings of the National Conference on Artificial Intelligence, pp. 203-208, 1997.

[3] M. Davis, G. Logemann and D. Loveland. A machine program for theorem proving. In Communications of the ACM, (5): 394-397, 1962.

[4] N. Eén and N. Sörensson. An extensible SAT-solver. In Proceedings of Sixth International Conference on Theory and Applications of Satisfiability Testing (SAT2003), May 2003.

[5] E. Goldberg and Y. Novikov. BerkMin: A fast and robust SAT-solver. In Design, Automation, and Test in Europe (DATE '02), p. 142-149, March 2002.

[6] C.P. Gomes, B. Selman and H. Kautz. Boosting combinatorial search through randomization. In Proceedings of the National Conference on Artificial Intelligence, July 1998.

[7] R.G. Jeroslaw and J. Wang. Solving propositional satisfiability problems. In Annals of mathematics and Artificial Intelligence, (1):167-187, 1990.

[8] H. Kautz, B. Selman. Planning as satisfiability. In Proceedings of the 10th European conference on Artificial intelligence, 1992.

[9] D. Le Berre and L. Simon. Fifty-five solvers in Vancouver: The SAT 2004 competition. In Proceedings of the Seventh International Conference on Theory and Applications of Satisfiability Testing (SAT2004), volume Lecture Notes in Computer Science, 2004. Accepted for publication.

[10] Y.S. Mahajan, Z. Fu, S. Malik. ZChaff2004: an efficient SAT solver. In Proceedings of the Seventh International Conference on Theory and Applications of Satisfiability Testing (SAT2004), volume Lecture Notes in Computer Science, 2004. Accepted for publication.

[11] J.P. Marques-Silva. The impact of branching heuristics in propositional satisfiability algorithms. In Proceedings of the 9th Portuguese Conference on Artificial Intelligence (EPIA), September 1999.

[12] J.P. Marques-Silva and K.A. Sakallah. GRASP: A search algorithm for propositional satisfiability. IEEE Transactions on Computers, (48):506-521, 1999.

[13] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang and S. Malik. Chaff: Engineering an efficient SAT solver. In Proceedings of the Design Automation Conference, 2001.

[14] M. R. Prasad, A. Biere, A. Gupta. A survey of recent advances in SAT-based formal verification, Intl. Journal on Software Tools for Technology Transfer (STTT), (7):156-173, January 2005.

[15] L. Ryan, Efficient algorithms for clause-learning SAT solvers. Masters thesis, Simon Fraser University, February 2004.

[16] Seventh International Conference on Theory and Applications of Satisfiability Testing (SAT2004), May 2004. URL http://satlive.org/SATCompetition/2004/ (viewed September 1, 2004).

[17] R. Zabih and D.A. McAllester. A rearrangement search strategy for determining propositional satisfiability. In Proceedings of National Conference on Artificial Intelligence, p. 155-160, 1988.

[18] L. Zhang, C.F. Madigan, M.H. Moskewicz and S. Malik. Efficient conflict driven learning in a boolean satisfiability solver. International Conference on Computer-Aided Design (ICCAD'01), p. 279-285, November 2001.