Main Techniques for Solving Real-World SAT Instances

Ilya Shlyakhter

I. Introduction

THE satisfiability (SAT) problem is: given a Boolean function $f(x_1, ..., x_n)$, find an assignment of Boolean values to the x_i such that f evaluates to true, or prove that no such assignment exists. Many important real-world problems – from software and hardware verification to AI planning – reduce naturally to the satisfiability problem [3]. This fact, together with advances in SAT solving technology, have established SAT as a de facto "virtual machine" for constraint problems.

While SAT is NP-complete, many instances that arise in practice are tractable. This happens because these instances have a regular structure that can be exploited by solvers. Large swaths of search space can be excluded quickly, and information gathered while searching one part of the search space may be used to speed up the search through other similarly-structured parts.

In this paper, I will review the principal optimization techniques used by SAT solvers to speed up the solution of SAT instances. I will talk about the relationship of these techniques to each other, and the rationale behind these techniques. I will point out some possible directions for further exploration.

II. Preliminaries

A. CNF formulas

A Boolean formula φ in CNF format is a conjunction of clauses, where each clause is a disjunction of literals, and each literal is either a Boolean variable or its negation. For instance, the formula $(x_1 \vee \overline{x_2} \vee \overline{x_3}) \wedge (\overline{x_3} \vee x_2 \vee x_3)$ is in CNF format. An *assignment* is a set of pairs (or *settings*) $< x_i, v >$ where $v \in \{0, 1\}$, giving values to some Boolean variables: e.g. $\{< x_1, 1 >, < x_3, 0 >\}$. An assignment that includes a setting for every variable used in the formula is a *full assignment*, one that leaves some variables unset is a *partial assignment*. Assignment A_2 extends (or is an extension of) assignment A_1 iff $A_1 \subseteq A_2$. A partial assignment makes each clause of the CNF either satisfied, falsified, or unresolved. A full assignment makes each clause either satisfied or falsified. A satisfying assignment satisfies all clauses of the CNF formula.

Each CNF clause thus prevents a single cube of the n-dimensional search space from containing satisfying assignments. For example, the clause $(x_1 \vee \overline{x_2} \vee \overline{x_3})$ excludes all assignments that extend the partial assignment $\{< x_1, 0 >, < x_2, 1 >, < x_3, 1 >\}$. The SAT problem is to find a complete assignment that is outside the union of exclusion cubes defined by the CNF clauses. A *constraint graph* of the CNF is a graph whose nodes are the Boolean variables of the CNF, and two nodes are connected by an edge iff they appear together in one of the CNF clauses.

A partial assignment is a *nogood* if it cannot be extended to a satisfying assignment [22]. Note that a nogood is a property of the Boolean function represented by the CNF, not of the particular CNF formula. If the CNF formula is unsatisfiable, any partial assignment (including the empty assignment) is a nogood. Some nogoods are easily derivable from the syntactic structure of the CNF formula; we call such nogoods "shallow nogoods". For example, the negation of any clause is a shallow nogood. Other nogoods are not obvious from the formula, and can only be discovered through search. We call such nogoods "deep nogoods".

The CNF format is much more restricted than general Boolean formulas with standard connectives. General Boolean formulas can be converted to CNF in polynomial time by adding auxiliary variables [6]. The addition of extra variables increases the size of the search space exponentially. Fortunately, raw size of the search space is less important that problem structure, which determines the size of the search tree and the actual search time. Translation to CNF preserves problem structure.

While restrictive, the CNF format enables efficient implementation of many critical operations in SAT solving, as we will see in subsequent sections.

B. n-queens problem

The algorithms in the subsequent sections use the n-queen problem to illustrate some points. The n-queen problem is to place n queens on an $n \times n$ chessboard, so that they do not threaten each other. To encode this problem in CNF, we allocate a Boolean variable to each square of the chessboard; the variable is 1 iff the square contains a queen. We will speak interchangeably of the square and the Boolean variable that represents it.

We encode the constraints with two groups of CNF clauses. The first group ("group 1") enforces the "do not threaten" constraint. If a, b are Boolean variables allocated to two different cells on the same row, column or diagonal, we include clause $\overline{a} \vee \overline{b}$. The second group ("group 2") states that each row and each column must contain a queen. For each row/column, we add a clause that is an OR of Boolean variables representing the squares of that row/column.

C. Backtracking search

All SAT algorithms considered here use backtracking as the basic strategy for organizing the search. Backtracking is a method for exhaustive systematic traversal of the search space. Backtracking organizes the search space into a *search tree*, and performs a depth-first traversal of that tree, generating nodes on the fly as necessary.

Each node of the decision tree contains a decision variable, and splits the search space into assignments settings this variable to 0 and to 1. The path from the root to each node thus corresponds to a partial assignment, and the node represents the portion of the search space consisting of extensions of this partial assignment.

The search starts at the root of the decision tree with an empty current partial assignment, and repeatedly executes three steps [1]:

1. Extend the current partial assignment with any necessary variable settings that can be deduced without search. Settings derived in this step are called *implications.* If all clauses are satisfied, print the current partial assignment and quit.
2. If the extended current partial assignment does not falsify any CNF clauses, choose an unassigned variable and tentatively set it to a truth value. The tentative setting is a *decision* and the variable set in this step is a *decision variable.* The new decision is marked as *open*, meaning that only one of two possible settings of the decision variable has been tried so far.
3. Otherwise (if a CNF clause was falsified), backtrack to the last open decision setting. Flip that setting to the opposite Boolean value, and undo any settings made later than that setting. Mark the decision to which we backtracked as *closed.* If we were unable to find an open decision to which to backtrack, report that the formula is unsatisfiable.

Step 1 is implemented by a procedure called *unit propagation* or Boolean Constraint Propagation (BCP). When all literals except one in some clause are falsified by the current partial assignment, and the remaining literal is unassigned, such clause is called a *unit clause* with respect to the current partial assignment. The sole unassigned literal in a unit clause must be set to true in any satisfying assignment that extends the current partial assignment. The unit clause that caused an implication setting is called the *antecedent clause* of that setting, and the individual settings that turned the antecedent clause of an implication into a unit clause are called *antecedent settings* of that implication. For example, if the CNF contains the clause $(x_1 \vee x_2 \vee \overline{x_3})$ and the current partial assignment includes the settings $x_1 = 0, x_2 = 0$, then the clause is a unit clause that implies the setting $x_3 = 0$. The clause $(x_1 \vee x_2 \vee \overline{x_3})$ is the antecedent clause of the setting $x_3 = 0$, and the setting $x_1 = 0$ and $x_2 = 0$ are the antecedent settings of the setting $x_3 = 0$. When a previously unassigned variable becomes implied by a unit clause, it may turn new clauses into unit clauses. For example, the implication $x_3 = 0$ turns the clause $x_3 \vee x5$ into a unit clause, causing a new implication $x5 = 1$. Unit propagation continues until no CNF clauses are unit clauses and no clause is falsified. If a clause is falsified, it is called a *conflicting clause* (not to be confused with *conflict clause* defined later). Backtracking search with unit propagation is called the Davis-Putnam algorithm [1].

Step 2 is based on the observation that if there exists a satisfying assignment, there either exists one in which the decision variable is true or one in which it is false. Backtracking explores one of these two possibilities first by extending the current partial assignment with it, and if that extension is later shown through search to be a nogood, Step 3 ensures that the remaining (opposite) possibility is then explored. The rule for choosing an unassigned variable in Step 2 is called the *decision heuristic* or the *splitting heuristic.* The sequence of decisions made in Step 2 is called the *decision stack.* Variables of the current partial assignment that are not set to a value are called *unassigned* or *unresolved* variables.

We associate each setting with the *decision level* at which it was made. The decision level of a decision setting is its distance from the root of the searh tree, plus one. The first decision has decision level one. Each time a decision is made in Step 2, the current decision level is incremented; each time Step 3 is executed, the current decision level is set to the decision level of the decision to which we backtrack. The decision level of an implication is set to the current decision level at the time the implication was derived in Step 1. Thus for every decision level, there is one decision and zero or more implications at that decision level. The current decision level tracks the depth of the decision stack. We use the notation $x_1 = 0@3$ to write "x_1 was set to 0 at decision level 3".

III. Optimization techniques

A number of improvements to the basic Davis-Putnam implementation are used by modern SAT solvers. Some improve the efficiency of the basic Davis-Putnam steps, such as propagation. Others reduce the size of the search tree by adding steps to the basic algorithm.

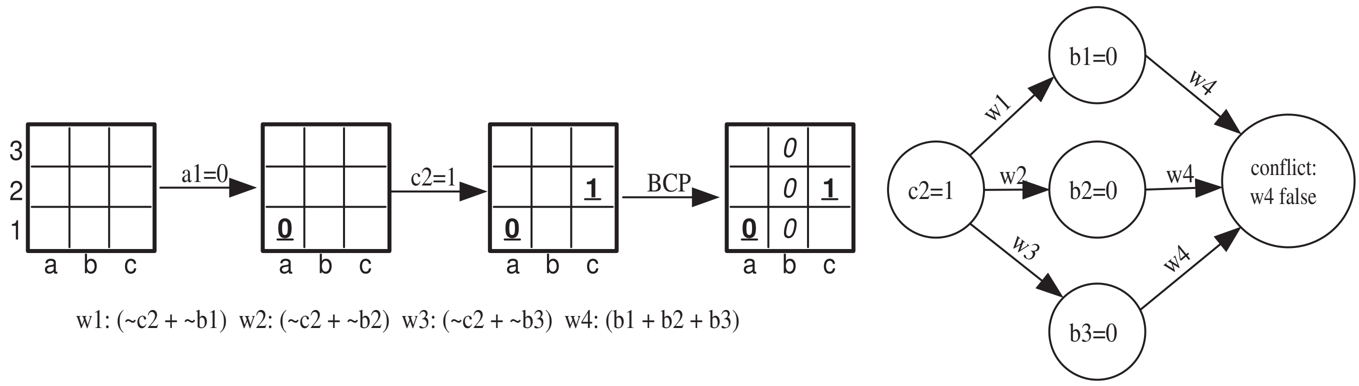

Fig. 1. Basic idea behind learning. The conflict created by decision sequence $a1 = 0, c2 = 1$ is due solely to $c2 = 1$. Therefore we can learn the clause $\overline{c2}$ from the conflict, as opposed to $a1 \vee \overline{c2}$ without conflict analysis.

A. Learning

A.1 Basic idea

When the backtracking search encounters a dead-end, the current decision stack is a nogood explicated by the search. It's possible that the not all of the current decision settings play a role in producing the dead-end. That is, if some of the settings were unassigned, the remaining settings would still imply the same dead-end (falsify the same clause).

Standard backtracking's control strategy ensures only that extensions of the *entire* current (nogood) decision stack will not be examined in future search. If we could determine that a proper subset of the current decision stack is a nogood, we could rule out extensions of that smaller nogood from future examination by adding a negation of the nogood as an explicit clause to the CNF formula. Unit propagation would then ensure that whenever all but one literal of the nogood was set to true, the remaining literal is immediately propagated to false. We would therefore effectively exclude a larger chunk of the solution space than if we had only excluded extensions of the entire current decision stack.

The basic concept is illustrated by a trivial example in Figure 1. The figure shows one path from root to leaf through a search tree of the 3-queens problem. After the decision sequence $a1 = 0, c2 = 1$ we get a conflict: all variables in column b are implied to zero by clauses $w1, w2, w3$, violating the clause $w4 = (b1 \vee b2 \vee b3)$. Ordinary backtracking would learn from this only that the entire decision sequence $a1 = 0, c2 = 1$ leads to a conflict, and would not repeat this sequence. In effect, this corresponds to learning the clause $(a1 \vee \overline{c2})$. However, the setting $a1 = 0$ played no role in creating the conflict; if $a1$ was unassigned, the conflict would happen anyway. This can be determined by looking at the source of the three settings $b1 = 0, b2 = 0, b3 = 0$ which falsified the clause $(b1 \vee b2 \vee b3)$, and discovering that they were all directly implied by the decision $c2 = 1$ (via clauses $w1, w2, w3$). We're therefore entitled to learn the shorter clause $(\overline{c2})$. Unlike the clause $(a1 \vee \overline{c2})$, this shorter clause will continue generating implications even after $a1$ is unassigned or set to 1. Clauses learned from conficts are called *conflict clauses.*

A.2 Implementation

We can determine the subset of decisions sufficient to imply the current empty clause, by recursively tracing back the reasons for the falsification of each literal in the clause. Each setting in the current partial assignment came about for one of two reasons, either as a decision or as an implication. For each implication, we can record a pointer to its antecedent clause as we record the implication. We can therefore obtain, for each setting in the current partial assignment, the set of earlier settings immediately responsible for that setting. By tracing backwards through the antecedent clauses, we can determine the decision assignments ultimately responsible for falsifying each literal in the current conflicting clause.

Formally, we analyze the *implication graph* [1], a directed acyclic graph which captures the structure of the conflict. The graph has one node representing each setting in the current partial assignment, and one special conflict node representing the falsified clause. We will speak interchangeably of the implication graph nodes and the settings they represent. Figure 1 includes the implication graph for the conflict encountered after the decision sequence $a1 = 0, c2 = 1$. Only the connected component containing the conflict node is shown; in the full graph, the decision $a1 = 0$ would be represented by an isolated node since it caused no implications.

The incident edges of each node originate from the antecedent settings of that node. Each edge is labeled with the antecedent clause of its target node. Decision settings have indegree zero, e.g. the node $c2 = 1$ in Figure 1. The indegree of each implication setting is one less than the size of its antecedent clause. In Figure 1, the implication setting $b1 = 0$ was implied by the binary clause $w1$, and its indegree is one. The indegree of the conflict node equals the size of the conflicting clause; in our example, the ternary clause $w4$ was falsified, and the conflict node has indegree three. Each setting node together with negations of its immediate predecessors corresponds to a clause in which exactly one literal (the setting node) is satisfied. For example, the node $b1 = 0$ together with its sole predecessor $c2 = 1$ arise from the binary clause $(\overline{c2} \vee \overline{b1})$ in which only the literal $\overline{b1}$ is satisfied. The conflict node is the only one with outdegree zero. While the implication graph is defined over all settings in the current partial assignment, only the connected component containing the conflict node is used for conflict analysis.

Given any node in the implication graph, we can determine various sets of nodes sufficient to imply that node. Let us call such sets the antecedent sets of a node. The set of immediate predecessors is obviously one such set; let us call it the immediate antecedent set. For example, the set $\{b1 = 0, b2 = 0, b3 = 0\}$ is the immediate antecedent set of the conflict node. We can obtain new antecedent sets by replacing a node in the antecedent set with its immediate predecessors. For instance, we can replace $b1 = 0$ with its predecessor $c2 = 1$ to obtain another antecedent set $\{c2 = 1, b2 = 0, b3 = 0\}$ of the conflict node. Replacing $b2 = 0$ in this set with its predecessor $c2 = 1$ yields the set $\{c2 = 1, b3 = 0\}$. Finally, replacing $b3 = 0$ with its predecessor $c2 = 1$ yields the set $\{c2 = 1\}$. Indeed, this single assignment suffices to falsify the clause $w4$, and the set $\{c2 = 1\}$ is an antecedent set of the conflict node. Any antecedent set of the conflict node is a nogood that can be effectively determined from the implication graph. Adding the negation of the nogood as a new clause will not remove any solutions, but may help constrain further search. In our example, we can add the clause $(\overline{c2})$ to the clause set. Observe that since *each* antecedent set of the conflict node corresponds to a clause that can be learned, there is a choice of clauses to learn, and more than one clause can be learned. Section III-A.5 discusses considerations for choosing which clause(s) to learn.

Replacing a node in the antecedent set with its immediate predecessors corresponds to the standard logical operation of *resolution* [3]. Resolution involves taking two clauses containing opposite literals of the same variable, and deriving a new clause (implied by the two original clauses) which does not refer to that variable. The literals of the new clause are literals appearing in either of the original clauses, except for the two opposite literals on which we are resolving. For instance, given two clauses $(x_1 \vee x_2 \vee x_3)$ and $(\overline{x_1} \vee x_4 \vee x_5)$ containing opposite literals of $x1$, we can derive a new clause $(x_2 \vee x_3 \vee x_4 \vee x_5)$ which does not refer to $x1$. If all literals in this new clause were false, the two original clauses would become unit clauses that imply two opposite literals, causing a contradiction. Replacing a node in an antecedent set with its immediate predecessors yields a resolution of the antecedent set with the clause comprised of the node and negations of its immediate predecessors. For instance, when we replace $b1 = 0$ in the immediate antecedent set $\{b1 = 0, b2 = 0, b3 = 0\}$ of the conflict node with its immediate predecessor $c2 = 1$, we obtain another antecedent set $\{c2 = 1, b2 = 0, b3 = 0\}$ of the conflict node. This corresponds to resolving the clauses $(b1 \vee b2 \vee b3)$ and $(\overline{c2} \vee \overline{b1})$ on $b1$. Thus backtracing of the implication graph corresponds to a series of resolution operations, and the learned clause is the result of resolving a set of existing clauses. Properties of the resolution operation will prove helpful in understanding the intuition behind some decision heuristics (section III-E.3).

A.3 Why it helps

At the lowest level, nogood learning helps by causing additional unit propagations. Before the negation of the nogood is added as an explicit clause, the nogood only affects the search (by causing a backtrack) if *all* literals of the nogood become set to true. After adding the clause, the nogood affects the search (by causing a unit propagation) as soon as *all but one* literal of the nogood become true. For an n-literal nogood, this means having n partial assignments of length $n-1$ that affect the search, instead of one partial assignment of length n. This means that the search is constrained more frequently by the nogood, after the nogood's negation is added as an explicit clause.

Still, it might appear that not much is gained, because the learned nogood is "shallow". Assigning all literals in the nogood caused immediate conflict even before learning the clause. It would be more useful if we could learn "deep" nogoods, ones that did not cause immediate conflict in the original CNF, but nevertheless could not be extended to a satisfying assignment.

Fortunately, the learning mechanism outlined above permits gradual learning of deeper and deeper nogoods. Whenever a nogood is learned from a conflict, that nogood is shallow (causes immediate conflict) relative to the clause set *at the time of the conflict*. The clause set includes the clauses of the initial CNF, as well as any nogoods learned earlier. Therefore, later in the search we can learn "deeper" nogoods which would not have caused an immediate conflict in the original CNF. Previously learned nogoods combine with original CNF clauses to turn these "deep" nogoods into "shallow" nogoods that can be learned from conflicts.

This process is analogous to what happens in ordinary backtracking, which may be viewed as a special case of nogood learning. When ordinary backtracking hits a conflict, it learns a nogood comprised of the entire current decision stack. For example, if the current decision stack $\{x_1 = 0@1, x_2 = 0@2, x_3 = 0@3\}$ leads to a conflict, ordinary backtracking learns the clause $(x_1 \vee x_2 \vee x_3)$. When the search backtracks, unassigning x_3, the newly learned clause becomes a unit clause and enables us to infer $x_3 = 1$. If that leads to a conflict, we can then infer that $\{x_1 = 0, x_2 = 0\}$ is a nogood and learn the clause $(x_1 \vee x_2)$. (The setting $x_3 = 1$ is not a decision but an implication of $\{x_1 = 0, x_2 = 0\}$ via the newly learned clause $(x_1 \vee x_2 \vee x_3)$, so x_3 does not become part of the subsequently learned clause $(x_1 \vee x_2)$.) Note that the nogood $\{x_1 = 0, x_2 = 0\}$ did *not* cause an immediate conflict in the original CNF – otherwise we would not

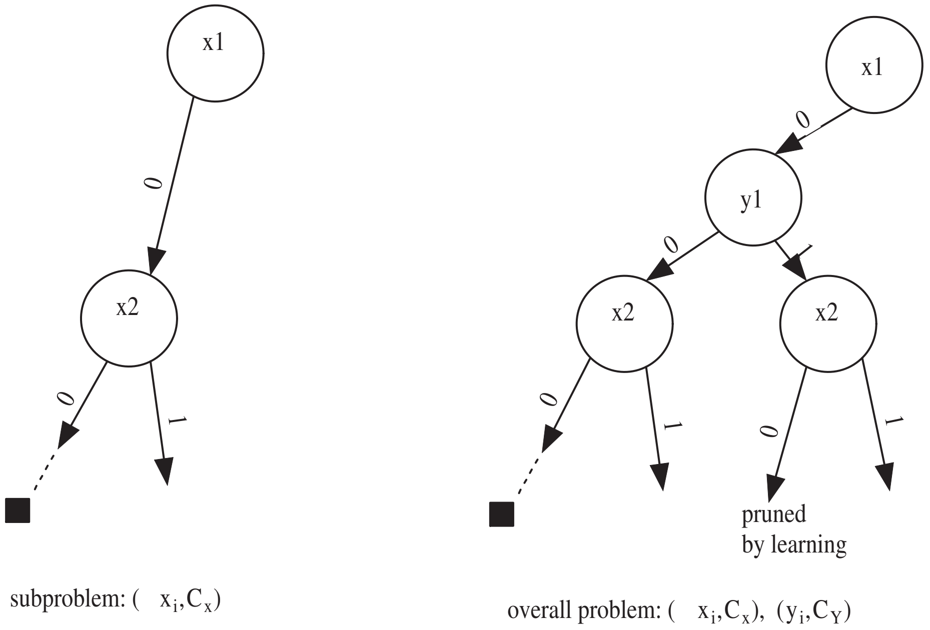

Fig. 2. Correspondence between search tree for the overall problem (right) and the search tree for the subproblem consisting of just the x_i variables constrained by C_X clauses (left). Learning ensures that in the overall problem, the branch $x1 = 0, x2 = 0$ is only tried once, as it is in the subproblem. Note: figure assumes that the 0 branches are tried before the 1 branches.

have extended it to a decision assignment $x_3 = 0@3$. However, it turned from a "deep" nogood into a "shallow" (and learnable) nogood after the clause $(x_1 \vee x_2 \vee x_3)$ was learned. The entire search may be viewed as a process of building on shallower nogoods to learn increasingly deeper nogoods, until either two opposite single-literal nogoods are learned or a solution is found [19].

Viewing backtracking as a special case of nogood learning helps us understand why learning improves performance on SAT instances that occur in practice. Such instances are often structured, comprising a combination of overconstrained and underconstrained subproblems [7]. When searching through a given subproblem as part of the overall search, learning makes that part of the search tree follow a search tree that would result from enumerating solutions to the subproblem in isolation. Without learning, search through the same unsatisfiable subproblem could be repeated many times.

Structured problems arising in practice typically represent SAT encodings of higher-level constraint problems [12], [14]. Clusters of Boolean variables represent higher-level objects such as graphs. Internal constraints on higher-level objects (e.g. that the graph is a tree) translate into intra-cluster CNF clauses. Constraints between higher-level objects (e.g. messages only sent on valid links) translate into inter-cluster CNF clauses among variables from the corresponding clusters.

To see the benefits of learning on structured problems, consider first a problem whose constraint graph separates into two connected components. (Such problems often arise as subproblems during search, as clauses are subsumed and the constraint graph becomes sparse [7].) The variables are $x_1, \ldots, x_p$ and $y_1, \ldots, y_q$. The clauses divide into groups: clauses in C_X refer only to the x_i variables; clauses C_Y refer only to the y_j variables; clauses in C_{XY} express constraints among x_i and y_j. For now, assume C_{XY} is empty.

When learning is used and all learned clauses are retained, the number of backtracks originating at C_X clauses is bounded by the number of backtracks in a search tree of just the variables x_i constrained by the clauses C_X. Since C_{XY} is empty, the implication graph constructed when a C_X clause becomes falsified contains only x_i settings. The number of backtracks caused by C_X clauses depends on the variable ordering among x_i, but not on the variable ordering among the y_j or on the particular interleaving of the x_i and y_j variables in the overall variable ordering for the problem. This situation is illustrated in Figure 2.

The number of backtracks caused by C_Y clauses is, similarly, bounded by the number of backtracks in a search tree over the variables y_j constrained by C_Y. Since each backtrack in the overall search tree is caused by either a C_X or a C_Y clause, this observation lets us bound the total number of backtracks in the search tree. The number of backtracks encountered during search correlates highly with actual search time [1], [9].

When C_{XY} is not empty, the above observations do not hold precisely. The implication graphs may contain both x_i and y_i settings, and the clauses learned from backtracks caused by C_X clauses may not correspond to clauses learned

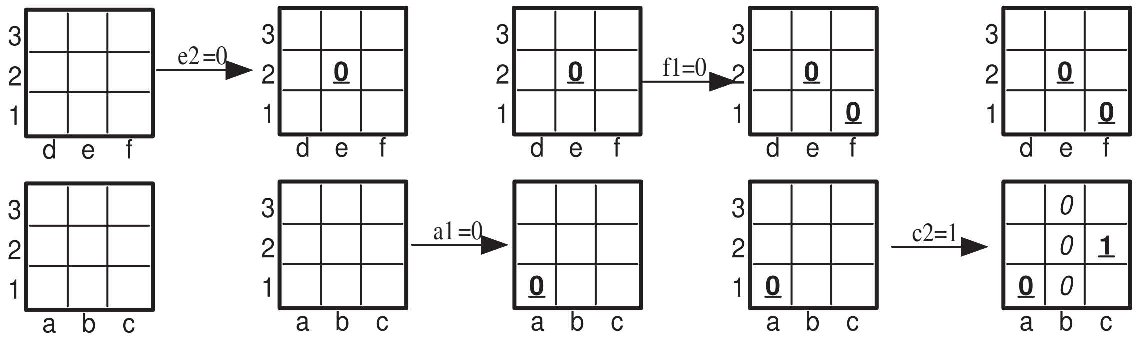

Fig. 3. Searching through the subproblem (bottom half) as part of searching through a larger overall problem. After decision sequence $e2 = 0, a1 = 0, f1 = 0, c2 = 1$ of the overall problem the same conflict clause is learned as after decision sequence $a1 = 0, c2 = 1$ of the subproblem.

when searching over isolated x_i constrained by C_X. Nevertheless, the correspondence may still hold over parts of the search tree. Also, if either the (x_i, C_X) problem or the y_j, C_Y problem is unsatisfiable, adding C_{XY} would typically speed up solving by increasing opportunities for unit propagation. [1] Hence, the bounds obtained assuming C_{XY} is empty should only improve when C_{XY} clauses are added.

Note that the situation where a large problem contains a small unsatisfiable subproblem is quite common in practical SAT instances [3]. This situation occurs particularly often in model checking applications of SAT. In these applications, an instance encodes a valid execution of a system. The constraints defining valid executions are typically a conjunction of many rules. It is easy for a rule or a small group of rules to accidentally exclude all possible executions. Learning ensures that once the subproblem defined by this small group of rules is determined to be unsatisfiable, the contradictory unit clauses learned from the subproblem quickly lead to declaring the overall problem unsatisfiable. This ability is particularly important when using model checking in an interactive setting, such as for design or refinement of algorithms.

Figure 3 illustrates searching through the 3-queens problem as part of a larger overall problem. In this case, the larger problem consists of two separate 3-queen problems. The decision heuristic alternately picks variables from the upper and lower boards. When the search encounters a conflict on the lower board, the implication graph is the same as in Figure 1. (More precisely, the connected component containing the conflict node is the same; the full implication graph for the overall problem contains nodes for $e2 = 0$ and $f1 = 0$ not present in the subproblem's implication graph. Only the connected component containing the conflict node is used in conflict analysis.) The same conflict clause $(\overline{c2})$ is learned when searching through the lower 3-queens problem as part of a larger problem, as when searching through the same 3-queens problem as a standalone problem. The interleaving of decision choices from upper and lower boards is made irrelevant by conflict analysis, which discovers the set of settings relevent to the lower-board subproblem in which the conflict occurred.

A.4 Costs

Learning imposes several extra costs on the search algorithm. First is the cost of maintaining the antecedent clause pointers from which the implication graph can be reconstructed. This cost is paid at every node, but is a small constant addition (in both time and space) to the already paid cost of setting and unsetting variable values in the current partial assignment.

Second is the cost of the actual conflict analysis. This cost is paid only at dead-ends, but can be non-negligible. In modern SAT solvers, conflict analysis takes up about 10% of running time [2].

The cost of conflict analysis for a particular conflict depends on the depth of the implication graph, as well as on the choice of learning scheme (see section III-A.5). The depth of the implication graph depends on the amount of unit propagation, as well as on the number of decision levels (depth of search). While a large number of unit propagations can lead to a deep implication graph and slow down conflict analysis, it also indicates that significant simplification is occurring during search. This can lead to a relatively short solution time even if conflict analysis is impeded.

Which portion of the implication graph actually gets examined depends on the learning scheme. Most learning schemes, when tracing the implication graph backwards, stop at nodes of decision levels earlier than the current level. This limits the cost of search to the amount of unit propagation at the latest (current) decision level. Two schemes from [4] use the entire implication graph (that is, the connected component containing the conflict node). Conflict analysis can be significantly more expensive for such schemes, which may be one reason they perform poorly in experiments.

[1] However, this is not guaranteed; see sections III-F.2 and III-F.5 on possible effects on search time when new clauses are added.

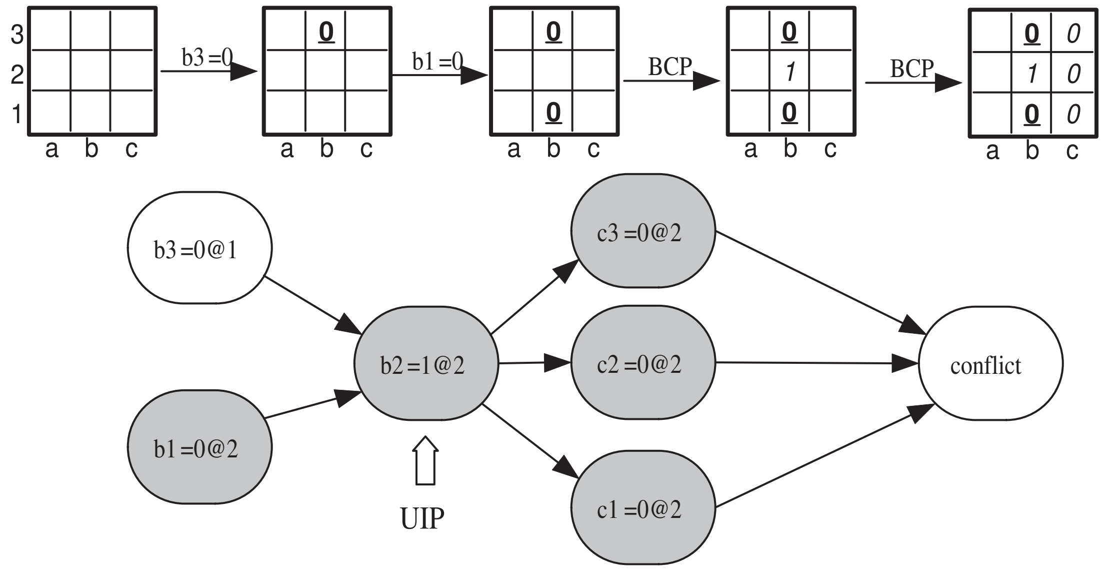

Fig. 4. Learning shorter clauses using a Unique Implication Points (UIP). The UIP $b2 = 1@2$ suffices to trigger the conflict. We may shorten the learned clause by replacing settings needed solely to imply $b2 = 1$ (here, $b3 = 0$ and $b1 = 0$) with the setting $b2 = 1$ itself.

The third cost of learning is that addition of clauses can increase memory usage of the solver, possibly causing virtual memory thrashing and killing performance. Moreover, addition of clauses can slow down unit propagation. More clauses need to be checked after each variable setting, to determine whether they have become unit clauses. This interaction is explored more fully in section III-F.5. Finally, learning can slow down computation of the decision heuristic, if the decision heuristic takes the entire clause database into account.

A.5 Design choices

One design choice in learning is deciding which clauses to learn. As noted above in section III-A.2, any antecedent set of the conflict node of the implication graph is a nogood of the original CNF, so its negation can be safely added to the clause set. We can therefore choose which nogood(s) to learn from a given implication graph.

It is reasonable to assume that we should strive to learn the shortest clauses possible. The shorter the clause, the larger region of search space is excluded by it. Also, shorter clauses will cause unit propagations more frequently, since fewer literals must be assigned before they cause unit propagations, and since they are less likely to become subsumed.

One technique used to obtain short nogoods from the implication graph is to find *Unique Implication Points* (UIP)s of the graph [1]. A UIP is a node at the current decision level (the level whose decision triggered the conflict), through which any path from the current decision node to the conflict node must pass. Intuitively, a UIP is a "bottleneck" of the implication graph: the current decision node may cause many implications but they all converge on the UIP node. If the current decision was replaced by the UIP, the same conflict would result; the UIP implies the nodes at the current decision level that lead to the conflict. The UIP may be included in the nogood in lieu of the implication nodes converging on it, leading to a shorter nogood. UIPs can be found by a single traversal of the implication graph [4].

Figure 4 illustrates the use of a UIP to shorten learned clauses. Shaded nodes show the implication chain triggered by the decision $b1 = 0@2$. The decision sequence leading to the conflict is $b3 = 0, b1 = 0$, so we could learn the clause $(b3 \vee b1)$. But the implication $b2 = 1$ is a UIP; if the decision at level 2 had been $b2 = 1$ instead of $b1 = 0$, the same conflict would result. We can therefore replace any settings needed solely to imply $b2 = 1$ (in this case, $b3 = 0$ and $b1 = 0$) with the UIP $b2 = 1$ itself. This allows us to learn the unary clause $\overline{b2}$, instead of the binary clause $(b3 \vee b1)$.

Note also that the clause $\overline{b2}$ is more "relevant" to the conflict. Learning the clause $(b3 \vee b1)$ would still allow essentially the same conflict to be reached by setting $a2 = 0, c2 = 0$: the UIP $b2 = 1$ would simply be implied in a different way. By contrast, learning the clause $\overline{b2}$ would prevent such recurrence of the conflict. Experiments show that learning the UIP that is closest to the conflict node leads to better performance than learning only decisions or learning UIPs farther from the conflict node [4].

Another design parameter in learning is deciding how long to keep the learned clauses. Keeping all learned clauses until the end of search is simple, and may be required to ensure completeness of search when using restarts (section III-C). However, unrestricted learning can exhaust memory and slow down unit propagation.

One solution called *bounded learning*, used in [1], is to keep only clauses shorter than some constant k. The number of distinct k-literal clauses over n variables is polynomial in n (for fixed k), so clause database growth becomes polynomial in the number of variables in the worst case. A generalization of this approach, called *relevance-bounded learning* [6], keeps a learned clause as long as the search remains in the general search space region in which the clause was learned, then deletes the clause. More precisely, relevance-bounded learning of order i deletes a learned clause when at least i of its literals become either flipped or unassigned since the clause was learned. The reasoning behind this heuristic is that the clause becomes less likely to cause unit propagations as its variables become unassigned, and becomes subsumed (and useless) as its variables become flipped. The tradeoff of any scheme that deletes clauses lies in missed opportunities for unit propagations that could have been caused by the clause. These unit propagations could have led to conflicts from which additional clauses, relevant to the current search region, could have been learned.

B. Non-chronological backtracking

B.1 Basic idea

When ordinary backtracking encounters a conflict, it backtracks to the last open decision and flips it. It's possible that the last open decision does not contribute to the conflict. In that case, the conflict will be re-encountered after the decision is flipped.

Non-chronological backtracking takes advantage of conflict analysis – the same procedure used for learning – to determine which settings in the current partial assignment are actually responsible for the conflict. It then determines how far it needs to backtrack in order to unassign one of these settings. It finds the latest decision on the decision stack which has resulted in one of the settings comprising the conflict, and backtracks to that decision. This saves us from examining partial assignments that extend the same nogood that led to the current conflict.

B.2 Implementation

Non-chronological backtracking shares the conflict analysis mechanism with learning (section III-A.2). Tracing back the implication graph from the conflict node, we can find all decision settings contributing to the conflict, and therefore determine the latest such assignment. In fact, tracing the graph all the way back to decision setting nodes is unnecessary. Observe that the decision level of an immediate predecessor (antecedent setting) of a node is never less than the decision level of the node. Therefore, when tracing backwards along any path in the implication graph, we can stop as soon as we hit a node with decision level earlier than the current decision level at the time of conflict: tracing further back will not find nodes with later decision levels.

Moreover, non-chronological backtracking is usually implemented in conjunction with learning [1], [2], [6]. Therefore, separate backtracing of the implication graph to determine the backtrack decision level is not needed. The correct level to which to backtrack can be determined directly from a learned clause. Since a learned clause represents a nogood, at least one literal needs to be unassigned before a solution can be found. We can therefore determine the latest decision level of the literals in the learned clause, and backtrack directly to that level.

B.3 Why it helps

Non-chronological backtracking saves search by eliminating solutionless branches that would be examied by ordinary backtracking. The amount of search saved depends on the decision heuristic [9], [10]. The interaction between non-chronological backtracking and decision heuristics is addressed in section III-F.4. Here we give some intuition on why opportunities for non-chronological backtracking are likely to arise in practice.

Recall the discussion of structured problems from section III-A.3. When a structured high-level constraint problem is encoded as a SAT instance, clusters of Boolean variables represent higher-level objects. Intra-cluster CNF clauses are added to restrict each cluster to valid representations of its high-level object. Inter-cluster CNF clauses are added to represent constraints between high-level objects.

The decision heuristic can organize the search either as a coarse-grained search or as a fine-grained search. In coarse-grained search, variables in each cluster occupy a continuous segment of the splitting order. In other words, the search chooses a complete value for one high-level object before proceeding to another object. By contrast, in fine-grained search the decision heuristic may assign only some of the variables in one cluster before proceeding to another cluster.

Fine-grained search permits greater flexibility, and has proven more effective in practice than coarse-grained search [17]. Often, the logic of inter-cluster constraints creates short intra-cluster nogoods that can be explicated by fine-grained search and used to prune the search space. For example, suppose one cluster of Boolean variables represents the adjacency matrix of a graph. If the logic of the overall problem, expressed with inter-cluster constraints, requires the graph to be connected for there to be a solution, fine-grained search can achieve effective pruning by assigning enough variables of the adjacency matrix to create one disconnected component, then assigning variables from other clusters to discover the contradiction and backtrack.

To see how non-chronological backtracking can help with fine-grained search, suppose we want to use SAT testing to check whether some heap operation preserves acyclicity of the heap. We allocate two square Boolean matrices X and Y to represent, respectively, the pre-state and the post-state of the heap; X_{ij} (Y_{ij}) is true iff object i points to object

j before (after) the operation. We add three sets of CNF clauses: a set C_X of clauses among X_{ij} constraining X to be acyclic; a set C_Y of clauses among Y_{ij} constraining Y to contain a cycle; and a set C_{XY} of clauses between X_{ij}, Y_{ij} constraining the (pre,post)-state pair to represent an execution that conforms to the specification of the operation.

Consider a splitting order of the form $x_1, \ldots, x_k, y_1, \ldots, y_m, x_{k+1}, \ldots, x_p, \ldots$ where x_i are variables from X and y_j are variables from Y. Suppose that for a particular partial assignment to $x_1, \ldots, x_k, y_1, \ldots, y_m$, the backtrack search has examined all extensions of this assignment to $x_{k+1}, \ldots, x_p$ that satisfy the intra-X_{ij} acyclicity constraints C_X among $x_1, \ldots, x_k, x_{k+1}, \ldots, x_p$, and has not found a solution. In this case, the current assignment to $x_1, \ldots, x_k$ is a nogood: no extension of it to $x_{k+1}, \ldots, x_p$ not yet examined will satisfy the C_X clauses. If we had been searching over only the variables of X (enumerating solutions to C_X) along the splitting order $x_1, \ldots, x_k, x_{k+1}, \ldots, x_p$, the backtrack search would backtrack to the latest open decision among $x_1, \ldots, x_k$ and change it.

However, we're searching over the variables of both X and Y, under the splitting order $x_1, \ldots, x_k, y_1, \ldots, y_m, x_k + 1, \ldots, x_p, \ldots$. The control strategy of standard backtrack search calls for backtracking to the latest open decision – which may be among $y_1, \ldots, y_m$ – even if the conflict is due entirely to the $x_1, \ldots, x_k$ portion of the current partial assignment. Standard backtracking may therefore waste time by examining extensions of a nogood assignment to $x_1, \ldots, x_k$. Non-chronological backtracking uses the implication graph to determine when a conflict is due to $x_1, \ldots, x_k$; if it is, no solutions will be found until one of the values assigned to $x_1, \ldots, x_k$ is changed. We may then backtrack far enough to change one of these values.

Observe that non-chronological backtracking on the overall problem (variables of X and Y constrained by clauses of C_X, C_Y and C_{XY}) mirrors the behavior of ordinary backtracking on the subproblem consisting of variables X_{ij} constrained by clauses C_X. A non-chronological backtrack on the overall problem corresponds to an ordinary backtrack on the (X, C_X) subproblem. Thus, non-chronological backtracking helps on structured problems for the same reason as learning (cf section III-A.3). When the search on the overall problem searches through a subproblem, non-chronological backtracking and learning make the search effort on the subproblem similar to what would be spent on searching that subproblem in isolation.

B.4 Costs

Non-chronological backtracking in isolation strictly reduces the number of nodes in the search tree [11]. However, when used (as it always is in practice) together with learning, non-chronological backtracking may lead to repeated exploration of partial assignments that would only be explored once if only learning was used. This interaction is addressed in section III-F.1.

C. Restarts

C.1 Basic idea

Search time for the same problem under different decision sequences can vary dramatically [16], [21]. For example, for any satisfiable problem there is a decision sequence that solves the problem in linear time (by choosing decisions from the solution, subsets of which cannot lead to backtracks); but many satisfiable problems take a long time to solve in practice. Decision heuristic which chooses the decision sequence can significantly affect search time [3], even though learning and non-chronological backtracking can reduce this variance [9].

Restarts address the problem of search time variance by running a number of shorter searches under varying decision sequences, instead of one long search. This can be viewed as "hedging the bet": reducing the probability of using a decision sequence that leads to an abnormally long search time. If the search has indeed stumbled into an unlucky decision sequence, the impact of the mistake will be limited by restarting with a different sequence. If there exist *some* decision orderings that lead to small search trees, restarts increase the probability of hitting such an ordering.

C.2 Implementation

Restarts are implemented simply by clearing the decision stack (resetting the decision level to 0) after every K backtracks, where K is a parameter of the restart strategy. To ensure that a different search tree is followed after each restart, a limited amount of randomness is introduced into the decision heuristic [16]. A "deterministic" decision heuristic can be randomized by introducing another parameter H, such that instead of returning the best-ranked decision the heuristic randomly chooses among the top H% of decision choices. Additional variation in decision-making can come from learned clauses. Since most decision heuristics consider the entire clause set in determining decision choices (and some weigh recently learned clauses more heavily), even without randomization the decision heuristic would make different choices after a restart, assuming learned clauses are kept. Besides contributing to variation in search, keeping clauses after a restart prevents the search from re-examining regions of space explored prior to the restart.

One issue that must be considered when using restarts is how to ensure completeness of the search algorithm. For a constant parameter K, the algorithm may not be complete. An unsatisfiable instance may require more than K backtracks to prove unsatisfiability, under any decision sequence. For instance, encodings of the pigeonhole problem require an exponential number of backtracks under all decision sequences [20]. One simple solution is to gradually increase K with each restart. Eventually K will be large enough to allow a search tree that exhausts the search space.

Another approach is to note that learning clauses during conflicts prevents those conflicts from recurring. Since there is a finite number of possible conflicts, unrestricted learning will eventually enable the solution to be found – or its absense proven – solely by unit propagation. A variation on this approach is to use restricted learning such as bounded or relevance-bounded learning, but gradually relax the restrictions as the search progresses [2]. Relaxing the restrictions ensures that any clause will eventually be learnable, so the set of clauses needed to allow exhausting the search space with fewer than K backtracks can eventually be learned.

C.3 Why it helps

Restarts help in several ways. First, they reduce the probability of an abnormally long search time. Second, they decrease the expected (average) runtime. Third, they increase the availability of good splitting decisions by allowing the search to explore only the smaller of the two subtrees under a decision node.

Experiments show that the distribution of search times on a given problem across decision sequences often exhibits a heavy right tail [16]. That is, given a randomly chosen decision sequence, there is a non-negligible propability that it leads to a very long solution time: significantly longer than the mean solution time across all random choices of decision sequence. This holds even when the random choice is not from all possible decision sequences, but from decision sequences returned by randomizing a reasonable decision heuristic. A deterministic decision heuristic corresponds to a single run of a randomized decision heuristic, and therefore suffers from the heavy right tail problem.

Restarts can decrease the probability of getting an abnormally long runtime, by taking advantage of a number of independent trials. Let p be the probability that the search completes (either finds a solution or exhausts the search space) within K backtracks. Then the probability that the search does *not* complete after R K-backtrack sessions is $(1-p)^R$. Reformulating slightly, the probability of not completing the search within S backtracks is on the order of $(1-p)^{S/K}$. This probability decays exponentially with increasing S, and the corresponding geometric distribution exhibits finite mean and variance. This holds even if the original probability $(1-p)$ of not completing within K backtracks decayed only polynomially in K, and its distribution exhibited infinite mean and variance [16].

Of course, this analysis assumes large enough K that the probability of completing the search within K backtracks is positive. If K is gradually increased during search, this condition will eventually be attained. Another assumption of the analysis is that the search sessions between restarts are all independent. If learned clauses are kept across restarts, this is not strictly true. Learned clauses change the problem that's being solved, and can affect the decision heuristic. However, the effect of learned clauses is generally beneficial. Learned clauses prevent conflicts from causing repeated backtracks, thereby reducing the number of backtracks required to complete the search. Therefore, learning should not worsen the bounds obtained for completely independent restart sessions. Experimental results appear to confirm that learning works in synergy with restarts [15]. Finally, note that the randomized algorithm may sometimes make suboptimal splitting decisions (within $H\%$ of the optimum), so its probability p of finishing within K backtracks may be somewhat lower than of the original algorithm. Since H is typically small, this should not significantly affect results.

In addition to decreasing the chance of abnormally long runtimes, restarts can also decrease the expected (average) runtime. The distribution of runtimes on many problems exhibits a heavy *left* tail, meaning there is a non-negligible probability of getting a significantly-below-average search time. In other words, the probability p of finishing the search in under K backtracks is non-negligible for very small K. Since each restart session for small K runs quickly, it becomes feasible to run sufficiently many independent sessions to achieve a high probability of finding a solution.

Note that the time taken by one K-backtrack session increases as more clauses are learned, since the speed of basic search operations (unit propagation, decision heuristic computation) depends on the total number of clauses. However, the above analysis is still valid as long as the total number of backtracks correlates well with overall search time. Experimental data supports this assumption [1].

Restarts help in one additional way: they permit the search to explore one branch of a decision without committing to exploring the other branch if the first branch does not lead to a solution. The two branches of a decision are often unbalanced, with one branch requiring much more search than the other [21], [24]. Without restarts, both branches must be explored. (This is obviously the case for unsatisfiable problems, but also holds for most branches in satisfiable problems [21].) The choice of branching variable must therefore take into account the size of both branches. A branching choice that produces significant simplification and leads to the learning of a short nogood may be precluded because its opposite branch produces little simplification and leads to a large search subtree. Such balancing considerations strongly influence the choice of real-life decision heuristics [21]. With restarts, it's possible to explore only the shorter of two branches, obtaining the benefit of learning a short nogood without the cost of exploring a deep subtree.

For example, restarts allow solving the 3-queen problem with just three one-node search trees. Figure 5 shows the three search trees. The bold cell indicates the decision node. In the n-queen problem, setting a cell to 1 (placing a queen) often generates many more implications than setting a cell to 0 (saying there is no queen). Thus, each of the settings $b2 = 1, b3 = 1, b1 = 1$ leads to an immediate contradiction by implying an entire row or entire column of

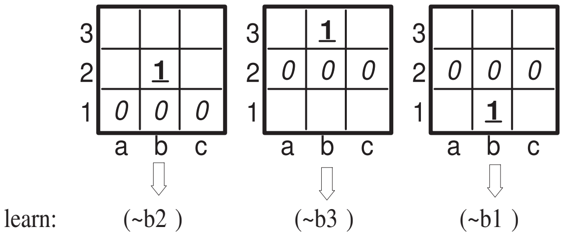

Fig. 5. Restarts allow solving the 3-queen problem with just three one-node search trees. Only the implications responsible for immediate backtracking are shown in each node. Each one-node search tree causes the learning of a unit clause; together the three unit clauses falsify the original CNF clause $(b_1 \vee b_2 \vee b_3)$.

zeroes (indicated by italics in Figure 5), contradicting the constraint that each row and each column has a queen. This enables the search to learn unit clauses $\overline{b2}, \overline{b3}, \overline{b1}$ which together create a queenless column b, creating an immediate contradiction. Without restarts, the search would be forced, after exploring the branch $b2 = 1$ and deriving the short learned clause $\overline{b2}$, to explore the larger branch $b2 = 0$.

C.4 Costs and Design Choices

One cost involved in using restarts is determining the appropriate values for parameters K and H. If K is so small that the probability of completing the search in K backtracks is zero or infinitesimal, the time spent running sessions of this length is essentially wasted. On the other hand, if K is too large, it will not be possible to run many K-backtrack sessions in a reasonable amount of time; the probability of hitting a difficult decision order will not be sufficiently reduced. The only suggested way to determine a good cutoff value is by trial-and-error, starting with a small K and gradually increasing it [16]. For some problems, such as the pigeonhole problem, all decision sequences lead to exponential-size search trees [20]. Therefore, the process of determining the optimal K may take considerable time. On the other hand, a single value of K may work for a class of similar problems, and would only need to be determined once for the class.

The choice of the parameter H, while not addressed in the literature, can also affect the search, although perhaps not as strongly as choice of K. Large H increases the possibility that the decision heuristic will make suboptimal splitting decisions (though within $H\%$ of the optimal). While restarts ensure that the impact of suboptimal splitting decisions remains limited, suboptimal splitting decisions can still adversely affect search performance. A completely random decision heuristic leads to very poor performance [21]. On the other hand, an insufficiently randomized decision heuristic (H too small) might not allow enough decision sequences leading to short searches.

D. Fast BCP

D.1 Basic idea

A basic operation of the Davis-Putnam algorithm involves determining, given the current partial assignment, which clauses are made unit clauses by that assignment, and whether any clauses are falsified. More specifically, we need to detect any new unit or empty clauses caused by assigning a literal (through either splitting or implication). The operation can be seen as having two stages. First, we need to identify the clauses that *might* become unit or empty as a result of setting a literal. Second, for each of these candidate clauses we need to determine whether the clause has actually become unit or empty.

A straightforward solution, used in [1], is to keep for each literal a list of clauses containing that literal, and for each clause keep counters of satisfied, falsified and unresolved literals in that clause. We will refer to this technique as per-clause counting. When a literal is set or unset, all clauses containing that literal would be visited and their counters updated. When the counter values indicate that the clause is a unit clause, all literals in the clause would be traversed to determine the sole unassigned literal that becomes implied. When the count of falsified literals matches clause size, we know we have a conflict. Checking the counters for a given clause takes constant time (independent of clause size). However, setting or unsetting a literal in this scheme takes time proportional to the number of clauses containing the literal. That number increases as new clauses are learned, increasing the cost of BCP and offsetting the benefits of search space pruning by learned clauses.

A simple observation lets us reduce the number of clauses visited when a literal is set, and avoid visiting any clauses when a literal is unset [2]. Observe that a clause only affects the search when its falsified literal count drops from two to one, or from one to zero. In other words, we do not need to look at a clause as long as it has at least two non-falsified (i.e. satisfied or unresolved) literals. Therefore, instead of keeping per-clause counters, we designate two non-falsified literals in every clause as *watched literals*. Until one of the watched literals is set to false, the clause cannot become unit or falsified, and we need not look at it.

D.2 Implementation

For every literal, we keep the list of clauses in which that literal is watched. When a literal is falsified by a decision or an implication, only the clauses in which it is watched need to be visited. For every such clause, we traverse the literal list to look for a non-falsified, unwatched literal. If such a literal is found, that literal becomes a new watched literal of the clause, replacing the literal that was falsified. If no such literal is found, and no literal in the clause is satisfied, the clause has become either unit or empty, depending on the whether the second watched literal is unassigned or false. During backtracking, nothing needs to be done (other than removing variable settings from the current partial assignment).

For example, suppose the CNF has a clause $(x_1 \vee x_3 \vee \overline{x_5} \vee \overline{x_7})$, in which the literals x_3 and $\overline{x_5}$ are randomly chosen to be watched. As long as only x_1 or x_7 become set or unset, this clause does not need to be considered. Suppose now that in the current partial assignment, x_1 is false and x_3, x_5, x_7 are unassigned. If x_3 becomes set to false, this clause will be visited (because it is on x_3's list of clauses in which x_3 is watched) and x_7 will become watched instead of x_3, making $\{\overline{x_5}, \overline{x_7}\}$ the watched set of literals in the clause. (Here, x_7 must be chosen because it is the only non-falsified, unwatched literal in the clause; if there were several such literals, any of them could have been chosen to be the new watched literal.) If x_5 becomes assigned to true while x_7 remains unassigned, we would visit this clause, determine that no unwatched unassigned literal exists and no literal is satisfied, and determine that the clause has become a unit clause forcing the implication of x_7 to false. The literals $\overline{x_5}$ and $\overline{x_7}$ remain the watched literals of the clause, but since the variables x_5 and x_7 are not unresolved, the clause will not be visited again until backtracking occurs. When backtracking does occur, x_5 and x_7 will be the first literals of the clause to become unassigned; therefore will not need to choose new watched literals for this clause.

D.3 Why it helps

Unit propagation is the bottleneck operation in most SAT solvers, consuming 80%-90% of the solver's time [2]. Speeding up this operation can therefore have significant impact on search time. The watched literals technique speeds up unit propagation by shortening the list of clauses that need to be considered when a given literal is set. Instead of looking at all clauses in which the literal appears, we only look at clauses in which it is watched. A given literal may be watched in only a small fraction of the clauses in which it appears – or in none.

Compared to keeping per-clause counters of satisfied and falsified literals, the cost that a given clause adds to unit propagation does not increase with clause length when watched literals are used. With counters, unit propagation incurs a constant cost due to a clause whenever *any* literal in the clause is either set or unset. With watched literals, a given clause only adds a cost to unit propagation when one of its two watched literals becomes set to false. Thus the savings are greatest on instances with large numbers of long clauses.

With counters, unassigning a literal takes time proportional to the number of clauses containing the literal. With watched literals, unassignment is constant-time. This is because unassignment, unlike assignment, cannot falsify any new literals. Therefore, any watched literals that were not falsified before an unassignment will not become falsified after, and can therefore remain valid watched literals. Note that on unsatisfiable instances, all assignments are eventually unassigned. On satisfiable instance, in practice most assignments are also eventually unassigned [21]. Therefore, constant-time unassignments should translate into a factor of two speedup for the watched-literal scheme compared to the clause-counter scheme.

The watched-literal scheme adapts well to the situation where a particular variable is unassigned and then reassigned again – a common situation during search. When a literal is set to false, it stops being the watched literal in any clause in which an unwatched, unfalsified literal can be found; another literal becomes watched in each such clause. Assigning a literal to false therefore sharply reduces the number of clauses in which that literal is watched. That number grows back only slowly, as other literals become falsified and force their watches to migrate to this literal. Unassignment does not increase the number of clauses in which the literal is watched. Therefore, when a literal is re-assigned shortly after being unassigned, the reassignment will happen faster than the initial assignment [2]. Reassignment after unassignment is made particularly common by the use of the VSIDS splitting heuristic (section III-E) which tends to assign variables appearing in recently learned conflict clauses. Since conflict clauses consist of variables that have recently been assigned, favoring these variables for splitting decisions increases the chance that these variables will be reassigned shortly after being unassigned.

D.4 Costs

While the watched-literal technique reduces the number of clauses that need to be considered when a literal is assigned, it may increase the work done at the clauses that are considered. With per-clause counters, a constant amount of work (checking counter values) suffices to determine whether the clause is unit or empty. With watched literals, in the worst case we need to look at all literals of the clause until we find an unwatched, non-falsified literal (or determine that none are left). The problem increases for longer clauses. In practice, however, the problem is mitigated since we need to search only as far as the first unwatched, non-falsified literal. And for any falsified literal that we step over during the search, recall that with per-clause counters we would have had to increment this clause's false-literal counter when the literal was set to false. This cost was avoided with watched-literal scheme because the literal was not watched in this clause. Experimental results consistently show that the watched-literal technique dominates per-clause counting on a variety of instances [2], [23].

Over the course of the search, a longer clause can add more cost to unit propagation even when watched literals are used. This is because a longer clause can stay in the database longer before becoming unit or empty. In other words, the operation "change the watched literals in this clause from one literal to another" can happen a larger number of times for a longer clause. While a longer clause has more opportunities to become satisfied, the basic watched literal scheme does not avoid visiting satisfied clauses.

D.5 Possible improvements

One source of inefficiency in the watched-literal scheme is that it visits satisfied clauses, even though they cannot cause conflicts or unit propagations. The inefficiency is especially large for clauses satisfied at early decision levels, since they're likely to remain satisfied for long periods of time. We could save computation by not watching literals in such clauses, until the clauses become unresolved due to backtracking. Observe that when one watched literal in a clause is satisfied, we do not need a second watched literal to know that the clause cannot become falsified or unit. Therefore, when a satisfied literal is designated a watched literal in a clause, we could remove the other watched literal (i.e. remove the clause from the other watched literal's list of clauses in which that literal is watched.) The removal would last until the satisfied watched literal was unassigned due to backtracking. Of course, at that point a second watched literal must be selected, unless another satisfied literal can be found. This approach does add work to the operation of unassigning a literal, but this cost may be offset by the savings from not visiting this clause when one of its watched literals is set. This approach would be particularly cost-effective for watching satisfied literals assigned at early decision levels, since they get unassigned least frequently. For such literals, the added cost of unassignment would be smallest, while the benefits of avoided visits to the clause would be most pronounced. It turns out that this suggestion has already been implemented in the solver Chaff, although it has not been published [5]. It may be extended further by going through clauses satisfied at early decision levels, and moving their watched literals to satisfied literals. Computational experiments would be needed to determine whether the savings outweigh the costs on practical instances.

A related optimization would be to choose watch literals that are least likely to become assigned. Watching literals least likely to become assigned means that literals that do become assigned are least likely to be watched, i.e. are watched in fewer clauses. This would lead to faster setting of literals, since fewer clauses would have to be visited. For example, literals with the lowest splitting heuristic scores (i.e. least likely to be assigned through decisions) could be chosen. This suggestion has been accepted for implementation in the next version of SAT solver JQuest [5].

A different optimization would involve not doing unit propagation faster, but doing less unit propagation by dropping the current insistence on complete unit propagation at every search node. The effectiveness of conflict analysis [1] suggests that only a fraction of settings in the partial assignment participate in conflicts. In other words, it's possible that many of the implications computed by unit propagation play no role in conflict creation; most conflicts would still be obtained (and conflict clauses learned) if the solver used less-than-complete unit propagation. With unit propagation taking 80%-90% of solver time, the potential savings from avoiding unit some unit propagations could be significant. Of course, foregoing complete unit propagation means forgoing the discovery and learning of some nogoods. It has been shown that Previous work has explored performing *more* inference at each search node than standard unit propagation [26]; it would be interesting to explore the "more search vs. more inference" tradeoff in the other direction.

E. Decision heuristics

E.1 Basic idea

Decision heuristic is the rule for choosing the next decision variable from among the unassigned variables, used at every node of the search tree. Decision heuristic may be the most important component of a SAT solver, with the most pronounced effect on search time [21]. Unfortunately, it is also the least understood component. The only reliable way to evaluate the effectiveness of a particular decision heuristic on a class of real-world problems is empirical [2]. The difficulty of designing good decision heuristics comes from the fact that early decisions, which have the strongest influence on search time, must be made using the least amount of information.

Early decisions matter most because the largest amount of subsequent search is influenced by them. This is especially true for hard satisfiable problems, which may have a "backbone" subset of variables only one particular assignment to which is extendable to a solution. In other words, all other assignments to the backbone variables are deep nogoods. Such is the case, for example, with SAT encodings of model checking problems where the algorithm has a subtle bug that occurs only under very specific circumstances. The wrong assignment to a backbone variable early in the search will result in a large amount of fruitless search, adding significantly to the overall search time.

At the same time, making early decisions involves the largest amount of extrapolation. In deciding which decision will lead to subproblems resulting in least search, the decision heuristic deals with the largest (least simplified) subproblems when making early decisions. Given that the actual search time resulting from the early decisions takes the longest time to determine, it's reasonable to assume that the early decisions involve the greatest amount of uncertainty. A major purpose of other SAT optimizations – learning, non-chronological backtracking and restarts – is to mitigate the effect of poor splitting decisions, particularly those made early in the search.

Experimentally, the most successful heuristic to date is the VSIDS heuristic introduced by Chaff [2]. This heuristic makes sense only when the algorithm uses learning, which all modern algorithms do. The heuristic focuses on satisfying recently learned conflict clauses. Variables appearing in a large number of recently learned conflict clauses are preferred as decision variables; and the branch that satisfies the recently learned conflict clauses is attempted first.

E.2 Implementation

The VSIDS heuristic is implemented by keeping, for each literal, a counter that is incremented whenever that literal appears in a newly learned clause. All counters are periodically divided by a constant. This ensures that literals that have not appeared in recently learned clauses have relatively low counter values (scores).

At each splitting decision, the unassigned literal with the highest score is selected. To speed up the selection, a list of unassigned literals sorted by the score is kept. The sorted list is incrementally maintained in sorted order.

E.3 Why it helps

On satisfiable instances, VSIDS attempts to make splitting decisions that are likely to be extendable to a solution. To see why, recall from section III-A.2 that each learned clause represents a resolution of a number of earlier clauses (i.e. clauses that were in the clause database when the learned clause was added). The resolvent of two clauses has the property that a partial assignment satisfying the resolvent can always be extended to satisfy the original two clauses [19]. For example, consider two clauses $(x_1 \vee x_2 \vee x_3)$ and $(\overline{x_1} \vee x_4 \vee x_5)$, resolvable in x_1. Resolving these clauses in x_1 gives the resolvent $(x_2 \vee x_3 \vee x_4 \vee x_5)$. Each literal of the resolvent comes from one of the two original clauses, so satisfying a literal of the resolvent will satisfy one of the original clauses. The other original clause can then be satisfied by setting its x_1 literal to true; while this falsifies x_1's literal in the first clause, that clause has already been satisfied. By a simple induction argument, this observation extends to a resolvent obtained by successively resolving several clauses; a partial assignment satisfying the resolvent can be extended to satisfy the original clauses.

As noted above, recently learned clauses are resolvents of sets of earlier clauses. Satisfying recent resolvents therefore "keeps our options open", allowing us to extend the current assignment to satisfy at least the clauses from which the resolvent was derived. This may help avoid deep nogoods, by preserving the possibility of extending the current partial assignment to a satisfying assignment – at least for the group of clauses from which the resolvent was derived. More recent resolvents are generally the result of longer resolution chains involving larger number of clauses; satisfying recent resolvents (as opposed to satisfying resolvents derived earlier, or clauses of the original database) preserves our ability to satisfy larger subsets of the current clause database. This can reduce the search time on satisfiable instances, by getting to a solution faster. Of course, that is a greedy assumption: we assume that by preserving the possibility of satisfying the largest number of clauses, we are more likely to preserve the possibilities for satisfying *all* clauses.

Another possible reason for the success of VSIDS (on both satisfiable and unsatisfiable instances) is that it leads to short learned clauses, which exclude large chunks of search space [13]. Observe that the resolvent of two clauses is shortest if the clauses share a large number of literals; each shared literal appears only once in the resolvent. By deciding on variables appearing in recently learned clauses, we increase the likelihood that future learned clauses will share literals with existing clauses, yielding shorter resolvents.

E.4 Costs

The computation costs of the VSIDS heuristic are relatively low. Note that the score of the variable does not depend on the current truth values of any variables, or on the subsumption status of any clauses. The costs are only updated at the time of a conflict, when a new clause is learned. Only the literals appearing in the new clause must have their counters updated; recall that learning strategies generally attempt to learn shorter clauses, so the number of literals to be updated for each conflict is minimized. For each literal, a counter is incremented (constant-time operation), and the literal's position in the array of sorted scores is updated (operation logarithmic in the number of variables). Finally, note that these costs are incurred only whenever the costs of conflict analysis are already being incurred. Experimentally, the cost of computing the VSIDS heuristic takes about 10% of solving time [2].

The danger of the VSIDS heuristic is that it may exacerbate poor splitting decisions made early in the search. Suppose the CNF formula has a subproblem that takes an exponential time to prove unsatisfiable, e.g. an instance of the pigeonhole problem. (Such instances often occur in practice, e.g. in circuit board design where the reason for unsatisfiability may be that n wires cannot be routed through the available $n-1$ spaces.) Initially all variables' scores are zero and ties are broken randomly by default [2], so it is possible that the first variables picked will be from this subproblem. This means that the first conflict clauses will likely consist of variables from the same subproblem, increasing the probability that the search will stay with the subproblem. In other words, the VSIDS heuristic may encourage coarse-grained search and create a heavy right tail of the distribution of search times – increasing the probability that the search will take an abnormally long time. In practice, the VSIDS heuristic is always use with restarts [16], [2], [3], which remove the heavy right tail (see section III-C.3). Also, the VSIDS heuristic simultaneously increases the benefits of good splitting decisions made early in the search, adding a heavy left tail of the distribution of search times. Restarts take advantage of heavy left tails to decrease the expected search time [16]. However, the VSIDS heuristic may reduce the independence of different restart sessions, by biasing splitting decisions in a fresh restart session towards the same variables used in decisions during the previous restart session. This may diminish the effectiveness of restarts in removing the heavy right tail of the search time distribution.

F. Interactions between techniques

This section describes some interactions between SAT optimizations. Descriptions of some other interactions are included with the descriptions of the optimizations in earlier sections.

F.1 Learning and non-chronological backtracking

Learning and non-chronological backtracking share the conflict analysis mechanism. This means that once one of these optimizations is implemented, the incremental cost of implementing the other is reduced. In all reported SAT solver implementations, either neither or both of these optimizations are present [1], [2], [3].

On the other hand, the incremental benefits of each of these two optimizations may be less when they're used together than when they're used individually. Since each of them prunes portions of the search tree, it deprives the other of an opportunity to save computation on the pruned portion.

Moreover, non-chronological backtracking may eliminate opportunities for learning that would have pruned a substantial amount of future search. In particular, consider the separated constraint graph problem from section III-A.3. Suppose a partial assignment to some subset of the x_i is a deep nogood, but some initial portion of that assignment produces no backtracks. (Heuristics such as VSIDS, which attempt to produce satisfiable subproblems, may tend to such conflictless partial assignments.) If non-chronological backtracking is used, an intra-y_i conflict can backtrack out of such conflictless partial assignment to the x_i before the deep nogood's conflict can be detected and learned. This means that the conflictless partial assignment can be repeated in the future. By contrast, chronological backtracking would have backtracked to the last open x_i decision, allowing the deep conflict to be learned and not repeated again.

F.2 Learning and decision heuristic

Learning affects decision heuristics by changing the clause set, on which most decision heuristics depend. Learning reduces the usefulness of static decision heuristics (those that determine a fixed decision order before the search, e.g. [25]), by substantially changing the clause set on which the search is done from the initial clause set on which the static decision order was computed. On the other hand, learning can gather useful information for dynamic decision heuristics, by identifying variables involved in conflict-making in the current portion of the search space. The most successful dynamic decision heuristic, VSIDS, is based primarily on statistics gathered from recently learned clauses.

Learning can give the search time a degree of independence from the decision heuristic, mitigating the effects of suboptimal splitting decisions. This is illustrated by the discussion of the separated constraint graph problem in section III-A.3. Learning bounds the number of backtracks caused by a given subproblem, regardless of how the decision heuristic interleaves variables of that subproblem with variables from the rest of the problem.

On the other hand, the decision heuristic determines the opportunities that are available for learning. The VSIDS heuristic attempts to provide opportunities for learning of shorter clauses, by splitting on variables from recently learned clauses to encourage shorter resolvents, as explained in section III-E.3.

F.3 Learning and restarts

Learning works in synergy with restarts; together these techniques can solve problems not amenable to each technique separately [15]. There can be several explanations for this. Learning prevents the search from repeatedly exploring the same parts of the search space after a restart [4]. Thus, deep nogoods learned during earlier restart sessions can significantly simplify the problem and prune search in later restart sessions. The addition of deep nogoods increases the probability of finishing a search session within a short time; in other words, it creates a heavy left tail of the distribution of search times. Restarts take advantage of this tail to decrease expected search time [16]. Furthermore, learning can increase the variety of decision orders available for restarts. While randomization of the decision heuristic ensures that different restart sessions use different decision orders, the amount of randomization must be limited to avoid diminishing the effectiveness of the splitting heuristic. Learning changes the set of clauses on which the decision heuristic is based, introducing an additional source of variation. For their part, restarts help learning by creating opportunities for learning short nogoods. For example, in Figure 5, restarts create opportunities for learning of unit clauses.

F.4 Decision heuristic and non-chronological backtracking

A "perfect" decision heuristic can obviate the need for non-chronological backtracking [10]. Given a search tree that uses non-chronological backtracking, a search tree of equal or smaller size can be constructed which uses only chronological backtracking. In practice, experiments show that the decision heuristic can reduce but not eliminate the opportunities for non-chronological backtracking [9]. In other words, non-chronological backtracking partially makes up for poor choices of splitting decisions.

F.5 Learning and Fast BCP

Learning can in general slow down BCP, since more clauses need to be checked for being unit or falsified after each variable setting. Also, additional clauses increase opportunities for unit propagation, increasing the number of settings that need to be unit-propagated. Fast BCP can make learning more tolerable, by decreasing the cost of an additional learned clause. The new clause is visited only when one of its two watched literals is falsified. The ability to keep a larger number of learned clauses translates into ability to prune more search through learned clauses.

IV. Conclusion

We have described five optimizations used in modern SAT solvers. Four of them – learning, non-chronological backtracking, restarts and decision heuristics – aim to reduce the size of the search tree. The fifth one – watched literals – speeds up generation of new tree nodes. While decision heuristics are poorly understood and introduce variability into the search time, other techniques – learning, non-chronological backtracking and restarts – mitigate the effects of this variability.

One area in which SAT solvers could improve is adding ability to use problem-specific information. For example, current SAT solvers cannot make direct use of symmetries of the problem. While some symmetry information can be given to the solver by adding symmetry-breaking CNF clauses [20], this method is not practical for problems with many symmetries. The ability to specify generators for a problem's symmetry group, and have the solver make direct use of symmetry to prune search, would be advantageous. Also, problems in many areas – including planning [14] and bounded model checking [12] – typically include many instantiations of the same formula template with different Boolean variables. Solvers could exploit this structure, for instance, to replicate deep nogoods learned about one copy of the structure onto other copies.

References

[1] J.P. Marques-Silva and K.A. Sakallah, *GRASP: A Search Algorithm for Propositional Satisfiability*, IEEE Transactions on Computers, vol. 48, no. 5, May 1999.

[2] C. Madigan, Y. Zhao, L. Zhang, S. Malik, *Chaff: Engineering an Efficient SAT Solver*, 39th Design Automation Conference, Las Vegas, June 2001.

[3] E. Goldberg and Y. Novikov, *BerkMin: a Fast and Robust Sat-Solver*, Design Automation and Test in Europe (DATE 2002)

[4] L. Zhang, C. Madigan, M. Moskewicz, S. Malik, *Efficient Conflict Driven Learning in a Boolean Satisfiability Solver*, Proceedings of ICCAD 2001, San Jose, CA, Nov. 2001.

[5] J.P. Marques-Silva Private communication

[6] R.J. Bayardo Jr and R.C. Schrag, *Using CSP Look-Back Techniques to Solve Real-World SAT Instances*, Proc. of the 14th Nat'l Conf. on Artificial Intelligence, pp. 203-208, 1997

[7] R.J. Bayardo Jr and J. D. Pehoushek *Counting Models using Connected Components*, Proc. of the Seventeenth Nat'l Conf. on Artificial Intelligence, 2000

[8] A. Plaisted and S. Greenbaum, *A Structure-preserving Clause Form Translation*, Journal of Symbolic Computation, 2:293–304, 1986

[9] J.P.Marques-Silva, *The Impact of Branching Heuristics in Propositional Satisfiability Algorithms*, Proceedings of the 9th Portuguese Conference on Artificial Intelligence (EPIA), September 1999

[10] X. Chen and P. van Beek, *Conflict-Directed Backjumping Revisited*, Journal of Artificial Intelligence Research 14 (2001) 53-81.

[11] G. Kondrak and P. van Beek, *A Theoretical Evaluation of Selected Backtracking Algorithms*, Artificial Intelligence, 89:365-387, 1997.

[12] E. Clarke, A. Biere, R. Raimi and Y. Zhu, *Bounded Model Checking Using Satisfiability Solving*, Formal Methods in System Design, vol. 19, issue 1, July 2001

[13] M. Moskewicz Private communication

[14] H. Kautz and B. Selman, *Planning as Satisfiability*, Proceedings ECAI-92

[15] L. Baptista and J. Marques-Silva, *The Interplay of Randomization and Learning on Real-World Instances of Satisfiability*, AAAI Workshop on Leveraging Probability and Uncertainty in Computation (AAAI-LPUC), July 2000.

[16] C.P. Gomes, B. Selman, H. Kautz *Boosting Combinatorial Search Through Randomization*, Proc. AAAI98, Madison, WI, July 1998

[17] D. Jackson *Automating First-Order Relational Logic* , Proc. ACM SIGSOFT Conf. Foundations of Software Engineering. San Diego, November 2000

[18] L. Baptista, I. Lynce and J. Marques-Silva *Complete Search Restart Strategies for Satisfiability* , IJCAI'01 Workshop on Stochastic Search Algorithms (IJCAI-SSA), August 2001

[19] E. Goldberg Private communication

[20] J.M. Crawford, M. Ginsberg, E. Luks, A. Roy *Symmetry Breaking Predicates for Search Problems* , Fifth International Conference on Principles of Knowledge Representation and Reasoning (KR'96)

[21] J.N. Hooker, V. Vinay *Branching rules for satisfiability* , Journal of Automated Reasoning, vol. 15, no. 3, pp. 359-383, 1995

[22] R. Dechter and D. Frost *Backjump-based Backtracking for Constraint Satisfaction Problems* , Preprint submitted to Elsevier Science, 13 December 2001

[23] I. Lynce and J.P. Marques-Silva *Efficient Data Structures for Backtrack Search SAT Solvers* , Proceedings of SAT02 conference, Cincinnati, Ohio, May 6-9 2002

[24] D. Kokotov and I. Shlyakhter *Progress Bar for SAT Solvers* , http://sdg.lcs.mit.edu/satsolvers/progressbar.html (unpublished)

[25] F.A. Aloul, I.L. Markov and K.A. Sakallah *MINCE: A Static Global Variable-Ordering for SAT and BDD* , IWLS 2001
[26] C.M. Li and Anbulagan *Heuristics based on unit propagation for satisfiability problems*, in proceedings of 15th International Joint Conference on Artificial Interlligence (IJCAI'97), Morgan Kaufmann Publishers, ISBN 1-55860-480-4, Page 366-371, Japon, 1997.