The editors

Chapter 1

Satisfiability solvers

Carla P. Gomes, Henry Kautz, Ashish Sabharwal, and Bart Selman

The past few years have seen an enormous progress in the performance of Boolean satisfiability (SAT) solvers. Despite the worst-case exponential run time of all known algorithms, satisfiability solvers are increasingly leaving their mark as a general-purpose tool in areas as diverse as software and hardware verification [28, 29, 30, 204], automatic test pattern generation [125, 197], planning [118, 177], scheduling [93], and even challenging problems from algebra [214]. Annual SAT competitions have led to the development of dozens of clever implementations of such solvers [e.g. 12, 18, 65, 85, 99, 107, 137, 139, 148, 151, 155, 156, 158, 159, 167, 178, 189, 191, 212], an exploration of many new techniques [e.g. 14, 92, 136, 155, 159], and the creation of an extensive suite of real-world instances as well as challenging hand-crafted benchmark problems [cf. 104]. Modern SAT solvers provide a "black-box" procedure that can often solve hard structured problems with over a million variables and several million constraints.

In essence, SAT solvers provide a generic combinatorial reasoning and search platform. The underlying representational formalism is propositional logic. However, the full potential of SAT solvers only becomes apparent when one considers their use in applications that are not normally viewed as propositional reasoning tasks. For example, consider AI planning, which is a PSPACE-complete problem. By restricting oneself to polynomial size plans, one obtains an NP-complete reasoning problem, easily encoded as a Boolean satisfiability problem, which can be given to a SAT solver [117, 118]. In hardware and software verification, a similar strategy leads one to consider *bounded* model checking, where one places a bound on the length of possible error traces one is willing to consider [29]. Another example of a recent application SAT solvers is in computing stable models used in the answer set programming paradigm, a powerful knowledge representation and reasoning approach [76]. In these applications – planning, verification, and answer set programming – the translation into a propositional representation (the "SAT encoding") is done automatically and hidden from the user: the user only deals with the appropriate higher-level representation language of the application domain. Note that the translation

to SAT generally leads to a substantial increase in problem representation. However, large SAT encodings are no longer an obstacle for modern SAT solvers. In fact, for many combinatorial search and reasoning tasks, the translation to SAT followed by the use of a modern SAT solver is often more effective than a custom search engine running on the original problem formulation. The explanation for this phenomenon is that SAT solvers have been engineered to such an extent that their performance is difficult to duplicate, even when one tackles the reasoning problem in its original representation.[1]

Although SAT solvers nowadays have found many applications outside of knowledge representation and reasoning, the original impetus for the development of such solvers can be traced back to research in knowledge representation. In the early to mid eighties, the tradeoff between the computational complexity and the expressiveness of knowledge representation languages became a central topic of research. Much of this work originated with a seminal series of papers by Brachman and Levesque on complexity tradeoffs in knowledge representation, in general, and description logics, in particular [34, 35, 36, 132, 133]. For a review of the state of the art of this work, see Chapter 3 of this handbook. A key underling assumption in the research on complexity tradeoffs for knowledge representation languages is that the best way to proceed is to find the most elegant and expressive representation language that still allows for worst-case polynomial time inference. In the early nineties, this assumption was challenged in two early papers on SAT [154, 191]. In the first [154], the tradeoff between typical case complexity versus worst-case complexity was explored. It was shown that most randomly generated SAT instances are actually surprisingly easy to solve (often in linear time), with the hardest instances only occurring in a rather small range of parameter settings of the random formula model. The second paper [191] showed that many satisfiable instances in the hardest region could still be solved quite effectively with a new style of SAT solvers based on local search techniques. These results challenged the relevance of the "worst-case" complexity view of the world.[2]

The success of the current SAT solvers on many real world SAT instances with millions of variables further confirms that typical case complexity and the complexity of real-world instances of NP-complete problems is much more amenable to effective general purpose solution techniques than worst-case complexity results might suggest. (For some initial insights into why real-world SAT instances can often be solved efficiently, see [209].) Given these developments, it may be worthwhile to reconsider the study of complexity tradeoffs in knowledge representation languages by not insisting on worst-case polynomial time reasoning but to allow for NP-complete reasoning sub-tasks that can be handled by a SAT solver. Such an approach would greatly extend the expressiveness of representation languages. The work on the use of SAT solvers to reason about stable models is a first promising example in this regard.

---

[1] Each year the International Conference on Theory and Applications of Satisfiability Testing hosts a SAT competition or race that highlights a new group of "world's fastest" SAT solvers, and presents detailed performance results on a wide range of solvers [129, 130, 193, 128]. In the 2006 competition, over 30 solvers competed on instances selected from thousands of benchmark problems. Most of these SAT solvers can be downloaded freely from the web. For a good source of solvers, benchmarks, and other topics relevant to SAT research, we refer the reader to the websites SAT Live! (`http://www.satlive.org`) and SATLIB (`http://www.satlib.org`).

[2] The contrast between typical- and worst-case complexity may appear rather obvious. However, note that the standard algorithmic approach in computer science is still largely based on avoiding any non-polynomial complexity, thereby implicitly acceding to a worst-case complexity view of the world. Approaches based on SAT solvers provide the first serious alternative.

In this chapter, we first discuss the main solution techniques used in modern SAT solvers, classifying them as complete and incomplete methods. We then discuss recent insights explaining the effectiveness of these techniques on practical SAT encodings. Finally, we discuss several extensions of the SAT approach currently under development. These extensions will further expand the range of applications to include multi-agent and probabilistic reasoning. For a review of the key research challenges for satisfiability solvers, we refer the reader to [116].

1.1 Definitions and Notation

A propositional or Boolean formula is a logic expressions defined over variables (or atoms) that take value in the set $\{\text{FALSE}, \text{TRUE}\}$, which we will identify with $\{0,1\}$. A *truth assignment* (or assignment for short) to a set V of Boolean variables is a map $\sigma : V \to \{0,1\}$. A *satisfying assignment* for F is a truth assignment σ such that F evaluates to 1 under σ. We will be interested in propositional formulas in a certain special form: F is in *conjunctive normal form* (CNF) if it is a conjunction (AND, $\wedge$) of *clauses*, where each clause is a disjunction (OR, $\vee$) of *literals*, and each literal is either a variable or its negation (NOT, $\neg$). For example, $F = (a \vee \neg b) \wedge (\neg a \vee c \vee d) \wedge (b \vee d)$ is a CNF formula with four variables and three clauses.

The Boolean Satisfiability Problem (SAT) is the following: *Given a CNF formula F, does F have a satisfying assignment?* This is the canonical NP-complete problem [45, 134]. In practice, one is not only interested in this decision ("yes/no") problem, but also in finding an actual satisfying assignment if there exists one. All practical satisfiability algorithms, known as SAT solvers, do produce such an assignment if it exists.

It is natural to think of a CNF formula as a set of clauses and each clause as a set of literals. We use the symbol Λ to denote the *empty clause*, i.e., the clause that contains no literals and is always unsatisfiable. A clause with only one literal is referred to as a *unit* clause. A clause with two literals is referred to as a *binary* clause. When every clause of F has k literals, we refer to F as a k-CNF formula. The SAT problem restricted to 2-CNF formulas is solvable in polynomial time, which for 3-CNF formulas, it is already NP-complete. A *partial assignment* for a formula F is a truth assignment to a subset of the variables of F. For a partial assignment ρ for a CNF formula F, $F|_\rho$ denotes the *simplified* formula obtained by replacing the variables appearing in ρ with their specified values, removing all clauses with at least one TRUE literal, and deleting all occurrences of FALSE literals from the remaining clauses.

CNF is the generally accepted norm for SAT solvers because of its simplicity and usefulness; indeed, many problems are naturally expressed as a conjunction of relatively simple constraints. CNF also lends itself to the DPLL process to be described next. The construction of Tseitin [201] can be used to efficiently convert any given propositional formula to one in CNF form by adding new variables corresponding to its subformulas. For instance, given an arbitrary propositional formula G, one would first locally re-write each of its logic operators in terms of $\wedge, \vee$, and $\neg$ to obtain, say, $G = (((a \wedge b) \vee (\neg a \wedge \neg b)) \wedge \neg c) \vee d$. To convert this to CNF, one possibility is to add four auxiliary variables w, x, y, and z, construct clauses that encode the four relations $w \leftrightarrow (a \wedge b)$, $x \leftrightarrow (\neg a \wedge \neg b)$, $y \leftrightarrow (w \vee x)$, and $z \leftrightarrow (y \wedge \neg c)$, and add to that the clause $(z \vee d)$.

s

1.2 SAT Solver Technology – Complete Methods

A *complete* solution method for the SAT problem is one that, given the input formula F, either produces a satisfying assignment for F or proves that F is unsatisfiable. One of the most surprising aspects of the relatively recent practical progress of SAT solvers is that the best complete methods remain variants of a process introduced several decades ago: the DPLL procedure, which performs a backtrack search in the space of partial truth assignments. The key feature of DPLL is the efficient pruning of the search space based on falsified clauses. Since its introduction in the early 1960's, the main improvements to DPLL have been smart branch selection heuristics, extensions like clause learning and randomized restarts, and well-crafted data structures such as lazy implementations and watched literals for fast unit propagation. This section is devoted to understanding these complete SAT solvers, also known as *systematic* solvers.

1.2.1 The DPLL Procedure

The Davis-Putnam-Logemann-Loveland or DPLL procedure is a complete, systematic search process for finding a satisfying assignment for a given Boolean formula or proving that it is unsatisfiable. Davis and Putnam [55] came up with the basic idea behind this procedure. However, it was only a couple of years later that Davis, Logemann, and Loveland [54] presented it in the efficient top-down form in which it is widely used today. It is essentially a branching procedure that prunes the search space based on falsified clauses.

Algorithm 1, `DPLL-recursive(`F, ρ`)`, sketches the basic DPLL procedure on CNF formulas. The idea is to repeatedly select an unassigned literal ℓ in the input formula F and recursively search for a satisfying assignment for $F|_\ell$ and $F_{\neg\ell}$. The step where such an ℓ is chosen is commonly referred to as the *branching* step. Setting ℓ to TRUE or FALSE when making a recursive call is called a *decision*, and is associated with a *decision level* which equals the recursion depth at that stage. The end of each recursive call, which takes F back to fewer assigned variables, is called the *backtracking* step.

A partial assignment ρ is maintained during the search and output if the formula turns out to be satisfiable. If $F|_\rho$ contains the empty clause, the corresponding clause of F from which it came is said to be *violated* by ρ. To increase efficiency, unit clauses are immediately set to TRUE as outlined in Algorithm 1; this process is termed *unit propagation*. *Pure literals* (those whose negation does not appear) are also set to TRUE as a preprocessing step and, in some implementations, in the simplification process after every branch.

Variants of this algorithm form the most widely used family of complete algorithms for formula satisfiability. They are frequently implemented in an iterative rather than recursive manner, resulting in significantly reduced memory usage. The key difference in the iterative version is the extra step of *unassigning* variables when one backtracks. The naive way of unassigning variables in a CNF formula is computationally expensive, requiring one to examine every clause in which the unassigned variable appears. However, the *watched literals* scheme provides an excellent way around this and will be described shortly.

1.2.2 Key Features of Modern DPLL-Based SAT Solvers

The efficiency of state-of-the-art SAT solvers relies heavily on various features that have been developed, analyzed, and tested over the last decade. These include fast unit propa-

Algorithm 1.1: `DPLL-recursive`(F,ρ)

```
Input  : A CNF formula F and an initially empty partial assignment ρ
Output : UNSAT, or an assignment satisfying F
begin
    (F,ρ) ← UnitPropagate(F,ρ)
    if F contains the empty clause then return UNSAT
    if F has no clauses left then
        Output ρ
        return SAT
    ℓ ← a literal not assigned by ρ                    // the branching step
    if DPLL-recursive(F|ℓ, ρ ∪ {ℓ}) = SAT then return SAT
    return DPLL-recursive(F|¬ℓ, ρ ∪ {¬ℓ})
end

sub UnitPropagate(F)
begin
    while F contains no empty clause but has a unit clause x do
        F ← F|x
        ρ ← ρ ∪ {x}
    return (F,ρ)
end
```

gation using watched literals, learning mechanisms, deterministic and randomized restart strategies, effective constraint database management (clause deletion mechanisms), and smart static and dynamic branching heuristics. We give a flavor of some of these below.

Variable (and value) selection heuristic is one of the features that vary the most from one SAT solver to another. Also referred to as the *decision strategy*, it can have a significant impact on the efficiency of the solver (see e.g. [147] for a survey). The commonly employed strategies vary from randomly fixing literals to maximizing a moderately complex function of the current variable- and clause-state, such as the MOMS (Maximum Occurrence in clauses of Minimum Size) heuristic [110] or the BOHM heuristic [cf. 31]. One could select and fix the literal occurring most frequently in the yet unsatisfied clauses (the DLIS (Dynamic Largest Individual Sum) heuristic [148]), or choose a literal based on its weight which periodically decays but is boosted if a clause in which it appears is used in deriving a conflict, like in the VSIDS (Variable State Independent Decaying Sum) heuristic [155]. Newer solvers like `BerkMin` [85], `Jerusat` [156], `MiniSat` [65], and `RSat` [167] employ further variations on this theme.

Clause learning has played a critical role in the success of modern complete SAT solvers. The idea here is to cache "causes of conflict" in a succinct manner (as learned clauses) and utilize this information to prune the search in a different part of the search space encountered later. We leave the details to Section 1.2.3, which will be devoted fully to clause learning. We will also see how clause learning provably exponentially improves upon the basic DPLL procedure.

The watched literals scheme of Moskewicz et al. [155], introduced in their solver `zChaff`, is now a standard method used by most SAT solvers for efficient constraint propagation. This technique falls in the category of lazy data structures introduced earlier by Zhang [212] in the solver `Sato`. The key idea behind the watched literals scheme, as the

name suggests, is to maintain and "watch" two special literals for each active (i.e., not yet satisfied) clause that are not FALSE under the current partial assignment; these literals could either be set to TRUE or as yet unassigned. Recall that empty clauses halt the DPLL process and unit clauses are immediately satisfied. Hence, one can always find such watched literals in all active clauses. Further, as long as a clause has two such literals, it cannot be involved in unit propagation. These literals are maintained as follows. Suppose a literal ℓ is set to FALSE. We preform two maintenance operations. First, for every clause C that had ℓ as a watched literal, we examine C and find, if possible, another literal to watch (one which is TRUE or still unassigned). Second, for every previously active clause C' that has now become satisfied because of this assignment of ℓ to FALSE, we make $\neg\ell$ a watched literal for C'. By performing this second step, positive literals are given priority over unassigned literals for being the watched literals.

With this setup, one can test a clause for satisfiability by simply checking whether at least one of its two watched literals is TRUE. Moreover, the relatively small amount of extra book-keeping involved in maintaining watched literals is well paid off when one unassigns a literal ℓ by backtracking – in fact, one needs to do absolutely nothing! The invariant about watched literals is maintained as such, saving a substantial amount of computation that would have been done otherwise. This technique has played a critical role in the success of SAT solvers, in particular those involving clause learning. Even when large numbers of very long learned clauses are constantly added to the clause database, this technique allows propagation to be very efficient – the long added clauses are not even looked at unless one assigns a value to one of the literals being watched and potentially causes unit propagation.

Conflict-directed backjumping, introduced by Stallman and Sussman [196], allows a solver to backtrack directly to a decision level d if variables at levels d or lower are the only ones involved in the conflicts in both branches at a point other than the branch variable itself. In this case, it is safe to assume that there is no solution extending the current branch at decision level d, and one may flip the corresponding variable at level d or backtrack further as appropriate. This process maintains the completeness of the procedure while significantly enhancing the efficiency in practice.

Fast backjumping is a slightly different technique, relevant mostly to the now-popular *FirstUIP* learning scheme used in SAT solvers `Grasp` [148] and `zChaff` [155]. It lets a solver to jump directly to a lower decision level d when even one branch leads to a conflict involving variables at levels d or lower only (in addition to the variable at the current branch). Of course, for completeness, the current branch at level d is *not* marked as unsatisfiable; one simply selects a new variable and value for level d and continues with a new conflict clause added to the database and potentially a new implied variable. This is experimentally observed to increase efficiency in many benchmark problems. Note, however, that while conflict-directed backjumping is always beneficial, fast backjumping may not be so. It discards intermediate decisions which may actually be relevant and in the worst case will be made again unchanged after fast backjumping.

Assignment stack shrinking based on conflict clauses is a relatively new technique introduced by Nadel [156] in their solver `Jerusat`, and is now used in other solvers as well. When a conflict occurs because a clause C' is violated and the resulting conflict clause C to be learned exceeds a certain threshold length, the solver backtracks to almost the highest decision level of the literals in C. It then starts assigning to FALSE the unassigned literals of the violated clause C' until a new conflict is encountered, which is expected to result in a smaller and more pertinent conflict clause to be learned.

Conflict Clause Minimization was introduced by Eén and Sörensson [65] in their solver `MiniSat`. The idea is to try to reduce the size of a learned conflict clause C by repeatedly identifying and removing any literals of C that are implied to be FALSE when the rest of the literals in C are set to FALSE. This is achieved using the subsumption resolution rule, which lets one derive a clause A from $(x \vee A)$ and $(\neg x \vee B)$ where $B \subseteq A$ (the derived clause A subsumes the antecedent $(x \vee A)$). This rule can be generalized, at the expense of extra computational cost that usually pays off, to a sequence of subsumption resolution derivations such that the final derived clause subsumes the first antecedent clause.

Randomized restarts, introduced by Gomes et al. [92] and further developed by Baptista and Marques-Silva [15], allow clause learning algorithms to arbitrarily stop the search and restart their branching process from decision level zero. All clauses learned so far are retained and now treated as additional initial clauses. Most of the current SAT solvers, starting with `zChaff` [155], employ very aggressive restart strategies, sometimes restarting after as few as 20 to 50 backtracks. This has been shown to help immensely in reducing the solution time. Theoretically, unlimited restarts, performed at the correct step, can provably make clause learning very powerful. We will discuss randomized restarts in more details later in the chapter.

1.2.3 Clause Learning and Iterative DPLL

Algorithm 1.2 gives the top-level structure of a `DPLL`-based SAT solver employing clause learning. Note that this algorithm is presented here in the *iterative* format (rather than recursive) in which it is most widely used in today's SAT solvers.

Algorithm 1.2: `DPLL-ClauseLearning-Iterative`

```
Input  : A CNF formula
Output : UNSAT, or SAT along with a satisfying assignment
begin
    while TRUE do
        DecideNextBranch
        while TRUE do
            status ← Deduce
            if status = CONFLICT then
                blevel ← AnalyzeConflict
                if blevel = 0 then return UNSAT
                Backtrack(blevel)
            else if status = SAT then
                Output current assignment stack
                return SAT
            else break
end
```

The procedure `DecideNextBranch` chooses the next variable to branch on (and the truth value to set it to) using either a static or a dynamic variable selection heuristic. The procedure `Deduce` applies unit propagation, keeping track of any clauses that may become empty, causing what is known as a conflict. If all clauses have been satisfied, it declares

the formula to be satisfiable.[3] The procedure `AnalyzeConflict` looks at the structure of implications and computes from it a "conflict clause" to learn. It also computes and returns the decision level that one needs to backtrack. Note that there is no explicit variable flip in the entire algorithm; one simply learns a conflict clause before backtracking, and this conflict clause often implicitly "flips" the value of a decision or implied variable by unit propagation. This will become clearer when we discuss the details of conflict clause learning and unique implication point.

In terms of notation, variables assigned values through the actual variable selection process (`DecideNextBranch`) are called *decision* variables and those assigned values as a result of unit propagation (`Deduce`) are called *implied* variables. *Decision* and *implied literals* are analogously defined. Upon backtracking, the last decision variable no longer remains a decision variable and might instead become an implied variable depending on the clauses learned so far. The *decision level of a decision variable x* is one more than the number of current decision variables at the time of branching on *x*. The *decision level of an implied variable y* is the maximum of the decision levels of decision variables used to imply *y*; if *y* is implied a value without using any decision variable at all, *y* has decision level zero. The *decision level* at any step of the underlying `DPLL` procedure is the maximum of the decision levels of all current decision variables, and zero if there is no decision variable yet. Thus, for instance, if the clause learning algorithm starts off by branching on *x*, the decision level of *x* is 1 and the algorithm at this stage is at decision level 1.

A clause learning algorithm stops and declares the given formula to be unsatisfiable whenever unit propagation leads to a conflict at decision level zero, i.e., when no variable is currently branched upon. This condition is sometimes referred to as a *conflict at decision level zero*.

Clause learning grew out of work in artificial intelligence seeking to improve the performance of backtrack search algorithms by generating explanations for failure (backtrack) points, and then adding the explanations as new constraints on the original problem. The results of Stallman and Sussman [196], Genesereth [77], Davis [56], Dechter [58], de Kleer and Williams [57], and others proved this approach to be quite promising. For general constraint satisfaction problems the explanations are called "conflicts" or "no-goods"; in the case of Boolean CNF satisfiability, the technique becomes clause learning – the reason for failure is learned in the form of a "conflict clause" which is added to the set of given clauses. Despite the initial success, the early work in this area was limited by the large numbers of no-goods generated during the search, which generally involved many variables and tended to slow the constraint solvers down. Clause learning owes a lot of its practical success to subsequent research exploiting efficient lazy data structures and constraint database management strategies. Through a series of papers and often accompanying solvers, Frost and Dechter [74], Bayardo Jr. and Miranker [16], Marques-Silva and Sakallah [148], Bayardo Jr. and Schrag [18], Zhang [212], Moskewicz et al. [155], Zhang et al. [216], and others showed that clause learning can be efficiently implemented and used to solve hard problems that cannot be approached by any other technique.

In general, the learning process hidden in `AnalyzeConflict` is expected to save us from redoing the same computation when we later have an assignment that causes conflict

[3] In some implementations involving lazy data structures, solvers do not keep track of the actual number of satisfied clauses. Instead, the formula is declared to be satisfiable when all variables have been assigned a truth value and no conflict is created by this assignment.

due in part to the same reason. Variations of such conflict-driven learning include different ways of choosing the clause to learn (different *learning schemes*) and possibly allowing multiple clauses to be learned from a single conflict. We next discuss formalize the graph-based framework used to define and compute conflict clauses.

Implication Graph and Conflicts

Unit propagation can be naturally associated with an *implication graph* that captures all possible ways of deriving all implied literals from decision literals. In what follows, we use the term *known clauses* to refer to the clauses of the input formula as well as to all clauses that have been learned by the clause learning process so far.

Definition 1. The *implication graph G* at a given stage of DPLL is a directed acyclic graph with edges labeled with sets of clauses. It is constructed as follows:

Step 1: Create a node for each decision literal, labeled with that literal. These will be the indegree zero source nodes of G.

Step 2: While there exists a known clause $C = (l_1 \vee \ldots l_k \vee l)$ such that $\neg l_1, \ldots, \neg l_k$ label nodes in G,

i. Add a node labeled l if not already present in G.

ii. Add edges $(l_i, l), 1 \leq i \leq k$, if not already present.

iii. Add C to the label set of these edges. These edges are thought of as grouped together and associated with clause C.

Step 3: Add to G a special "conflict" node $\overline{\Lambda}$. For any variable x that occurs both positively and negatively in G, add directed edges from x and $\neg x$ to $\overline{\Lambda}$.

Since all node labels in G are distinct, we identify nodes with the literals labeling them. Any variable x occurring both positively and negatively in G is a *conflict variable*, and x as well as $\neg x$ are *conflict literals*. G contains a *conflict* if it has at least one conflict variable. DPLL at a given stage has a *conflict* if the implication graph at that stage contains a conflict. A conflict can equivalently be thought of as occurring when the residual formula contains the empty clause Λ.

By definition, an implication graph may not contain a conflict at all, or it may contain many conflict variables and several ways of deriving any single literal. To better understand and analyze a conflict when it occurs, we work with a subgraph of an implication graph, called the *conflict graph* (see Figure 1.1), that captures only one among possibly many ways of reaching a conflict from the decision variables using unit propagation.

Definition 2. A *conflict graph H* is any subgraph of an implication graph with the following properties:

(a) H contains $\overline{\Lambda}$ and exactly one conflict variable.

(b) All nodes in H have a path to $\overline{\Lambda}$.

(c) Every node l in H other than $\overline{\Lambda}$ either corresponds to a decision literal or has precisely the nodes $\neg l_1, \neg l_2, \ldots, \neg l_k$ as predecessors where $(l_1 \vee l_2 \vee \ldots \vee l_k \vee l)$ is a known clause.

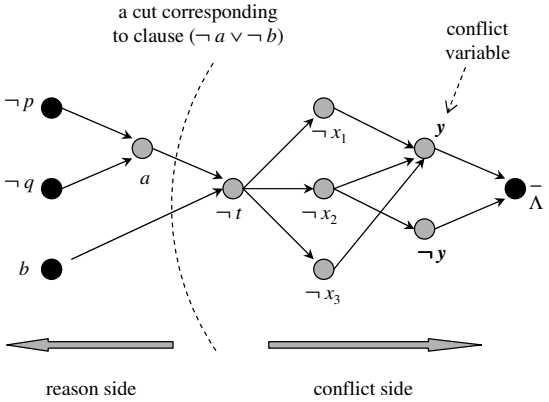


Figure 1.1: A conflict graph

While an implication graph may or may not contain conflicts, a conflict graph always contains exactly one. The choice of the conflict graph is part of the strategy of the solver. A typical strategy will maintain one subgraph of an implication graph that has properties (b) and (c) from Definition 2, but not property (a). This can be thought of as a *unique inference* subgraph of the implication graph. When a conflict is reached, this unique inference subgraph is extended to satisfy property (a) as well, resulting in a conflict graph, which is then used to analyze the conflict.

Conflict clauses

For a subset U of the vertices of a graph, the *edge-cut* (henceforth called a cut) corresponding to U is the set of all edges going from vertices in U to vertices not in U.

Consider the implication graph at a stage where there is a conflict and fix a conflict graph contained in that implication graph. Choose any cut in the conflict graph that has all decision variables on one side, called the *reason side*, and $\overline{\Lambda}$ as well as at least one conflict literal on the other side, called the *conflict side*. All nodes on the reason side that have at least one edge going to the conflict side form a *cause* of the conflict. The negations of the corresponding literals forms the *conflict clause* associated with this cut.

Learning Schemes

The essence of clause learning is captured by the *learning scheme* used to analyze and learn the "cause" of a failure. More concretely, different cuts in a conflict graph separating decision variables from a set of nodes containing $\overline{\Lambda}$ and a conflict literal correspond to different learning schemes (see Figure 1.2). One may also define learning schemes based on cuts not involving conflict literals at all such as a scheme suggested by Zhang et al. [216], but the effectiveness of such schemes is not clear. These will not be considered here.

It is insightful to think of the *nondeterministic* scheme as the most general learning scheme. Here we select the cut nondeterministically, choosing, whenever possible, one whose associated clause is not already known. Since we can repeatedly branch on the

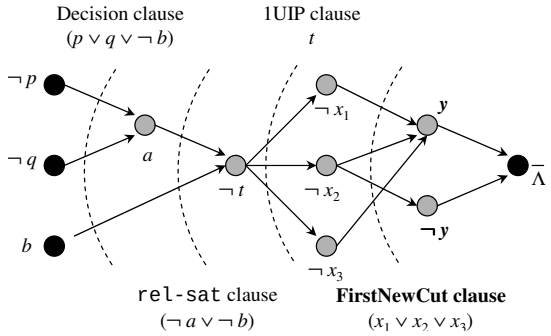


Figure 1.2: Learning schemes corresponding to different cuts in the conflict graph

same last variable, nondeterministic learning subsumes learning multiple clauses from a single conflict as long as the sets of nodes on the reason side of the corresponding cuts form a (set-wise) decreasing sequence. For simplicity, we will assume that only one clause is learned from any conflict.

In practice, however, we employ deterministic schemes. The *decision* scheme [216], for example, uses the cut whose reason side comprises all decision variables. `relsat` [18] uses the cut whose conflict side consists of all implied variables at the current decision level. This scheme allows the conflict clause to have exactly one variable from the current decision level, causing an automatic flip in its assignment upon backtracking. In the example depicted in Figure 1.2, the decision clause $(p \vee q \vee \neg b)$ has b as the only variable from the current decision level. After learning this conflict clause and backtracking by unassigning b, the truth values of p and q (both FALSE) immediately imply $\neg b$, flipping the value of b from TRUE to FALSE.

This nice flipping property holds in general for all *unique implication points* (UIPs) [148]. A UIP of an implication graph is a node at the current decision level d such that any path from the decision variable at level d to the conflict variable as well as its negation must go through it. Intuitively, it is a *single* reason at level d that causes the conflict. Whereas `relsat` uses the decision variable as the obvious UIP, `Grasp` [148] and `zChaff` [155] use *FirstUIP*, the one that is "closest" to the conflict variable. `Grasp` also learns multiple clauses when faced with a conflict. This makes it typically require fewer branching steps but possibly slower because of the time lost in learning and unit propagation.

The concept of UIP can be generalized to decision levels other than the current one. The *1UIP scheme* corresponds to learning the FirstUIP clause of the current decision level, the *2UIP scheme* to learning the FirstUIP clauses of both the current level and the one before, and so on. Zhang et al. [216] present a comparison of all these and other learning schemes and conclude that 1UIP is quite robust and outperforms all other schemes they consider on most of the benchmarks.

Another learning scheme, which underlies the proof of a theorem to be presented in the next section, is the *FirstNewCut* scheme [21]. This scheme starts with the cut that is closest to the conflict literals and iteratively moves it back toward the decision variables until a conflict clause that is not already known is found; hence the name FirstNewCut.

1.2.4 A Proof Complexity Perspective

Propositional proof complexity is the study of the structure of proofs of validity of mathematical statements expressed in a propositional or Boolean form. Cook and Reckhow [46] introduced the formal notion of a proof system in order to study mathematical proofs from a computational perspective. They defined a propositional proof system to be an efficient algorithm A that takes as input a propositional statement S and a purported proof π of its validity in a certain pre-specified format. The crucial property of A is that for all invalid statements S, it rejects the pair (S, π) for all π, and for all valid statements S, it accepts the pair (S, π) for some proof π. This notion of proof systems can be alternatively formulated in terms of unsatisfiable formulas — those that are FALSE for all assignments to the variables.

They further observed that if there is no propositional proof system that admits short (polynomial in size) proofs of validity of all tautologies, i.e., if there exist computationally hard tautologies for every propositional proof system, then the complexity classes NP and co-NP are different, and hence P $\neq$ NP. This observation makes finding tautological formulas (equivalently, unsatisfiable formulas) that are computationally difficult for various proof systems one of the central tasks of proof complexity research, with far reaching consequences to complexity theory and Computer Science in general. These hard formulas naturally yield a hierarchy of proof systems based on the sizes of proofs they admit. Tremendous amount of research has gone into understanding this hierarchical structure. Beame and Pitassi [22] summarize many of the results obtained in this area.

To understand current complete SAT solvers, we focus on the proof system called *resolution*, denoted henceforth as RES. It is a very simple system with only one rule which applies to disjunctions of propositional variables and their negations: $(a \text{ OR } b)$ and $((\text{NOT } a) \text{ OR } c)$ together imply $(b \text{ OR } c)$. Repeated application of this rule suffices to derive an empty disjunction if and only if the initial formula is unsatisfiable; such a derivation serves as a proof of unsatisfiability of the formula.

Despite its simplicity, unrestricted resolution as defined above (also called *general resolution*) is hard to implement efficiently due to the difficulty of finding good choices of clauses to resolve; natural choices typically yield huge storage requirements. Various restrictions on the structure of resolution proofs lead to less powerful but easier to implement refinements that have been studied extensively in proof complexity. Those of special interest to us are *tree-like resolution*, where every derived clause is used at most once in the refutation, and *regular resolution*, where every variable is resolved upon at most one in any "path" from the initial clauses to the empty clause. While these and other refinements are sound and complete as proof systems, they differ vastly in efficiency. For instance, in a series of results, Bonet et al. [32], Bonet and Galesi [33], and Buresh-Oppenheim and Pitassi [39] have shown that regular, ordered, linear, positive, negative, and semantic resolution are all exponentially stronger than tree-like resolution. On the other hand, Bonet et al. [32] and Alekhnovich et al. [6] have proved that tree-like, regular, and ordered resolution are exponentially weaker than RES.

Most of today's complete SAT solvers implement a subset of the resolution proof system. However, till recently, it wasn't clear where exactly do they fit in the proof system hierarchy and how do they compare to refinements of resolution such as regular resolution. Clause learning and random restarts can be considered to be two of the most important ideas that have lifted the scope of modern SAT solvers from experimental toy problems

to large instances taken from real world challenges. Despite overwhelming empirical evidence, for many years not much was known of the ultimate strengths and weaknesses of the two.

Beame, Kautz, and Sabharwal [21, 179] answered several of these questions in a formal proof complexity framework. They gave the first precise characterization of clause learning as a proof system called CL and began the task of understanding its power by relating it to resolution. In particular, they showed that with a new learning scheme called FirstNewCut, clause learning can provide exponentially shorter proofs than any proper refinement of general resolution satisfying a natural self-reduction property. These include regular and ordered resolution, which are already known to be much stronger than the ordinary DPLL procedure which captures most of the SAT solvers that do not incorporate clause learning. They also showed that a slight variant of clause learning with unlimited restarts is as powerful as general resolution itself.

From the basic proof complexity point of view, only families of unsatisfiable formulas are of interest because only proofs of unsatisfiability can be large; minimum proofs of satisfiability are linear in the number of variables of the formula. In practice, however, many interesting formulas are satisfiable. To justify the approach of using a proof system CL, we refer to the work of Achlioptas, Beame, and Molloy [2] who have shown how negative proof complexity results for unsatisfiable formulas can be used to derive time lower bounds for specific inference algorithms, especially DPLL, running on satisfiable formulas as well. The key observation in their work is that before hitting a satisfying assignment, an algorithm is very likely to explore a large unsatisfiable part of the search space that corresponds to the first bad variable assignment.

Proof complexity does not capture everything we intuitively mean by the power of a reasoning system because it says nothing about how difficult it is to *find* shortest proofs. However, it is a good notion with which to begin our analysis because the size of proofs provides a lower bound on the running time of any implementation of the system. In the systems we consider, a branching function, which determines which variable to split upon or which pair of clauses to resolve, guides the search. A negative proof complexity result for a system ("proofs must be large in this system") tells us that a family of formulas is intractable even with a perfect branching function; likewise, a positive result ("small proofs exist") gives us hope of finding a good branching function, i.e., a branching function that helps us uncover a small proof.

We begin with an easy to prove relationship between DPLL (without clause learning) and tree-like resolution (for a formal proof, see e.g. [179]).

Proposition 1. *For a CNF formula F, the size of the smallest DPLL refutation of F is equal to the size of the smallest tree-like resolution refutation of F.*

The interesting part is to understand what happens when clause learning is brought into the picture. It has been previously observed by Lynce and Marques-Silva [144] that clause learning can be viewed as adding resolvents to a tree-like resolution proof. The following results show further that clause learning, viewed as a propositional proof system CL, is exponentially stronger than tree-like resolution. This explains, formally, the performance gains observed empirically when clause learning is added to DPLL based solvers.

Clause Learning Proofs

The notion of clause learning proofs connects clause learning with resolution and provides the basis for the complexity bounds to follow. If a given formula F is unsatisfiable, the clause learning based DPLL process terminates with a conflict at decision level zero. Since all clauses used in this final conflict themselves follow directly or indirectly from F, this failure of clause learning in finding a satisfying assignment constitutes a logical proof of unsatisfiability of F. In an informal sense, we denote by CL the proof system consisting of all such proofs; this can be made precise using the notion of a branching sequence [21]. The results below compare the sizes of proofs in CL with the sizes of (possibly restricted) resolution proofs. Note that clause learning algorithms can use one of many learning schemes, resulting in different proofs.

We next define what it means for a refinement of a proof system to be natural and proper. Let $\mathscr{C}_S(F)$ denote the length of the short refutation of a formula F under a proof system S.

Definition 3 ([21, 179])**.** For proof systems S and T, and a function $f : \mathbb{N} \to [1,\infty)$,

- S is *natural* if for any formula F and restriction ρ on its variables, $\mathscr{C}_S(F|_\rho) \leq \mathscr{C}_S(F)$.
- S is a *refinement* of T if proofs in S are also (restricted) proofs in T.
- S is $f(n)$*-proper* as a refinement of T if there exists a witnessing family $\{F_n\}$ of formulas such that $\mathscr{C}_S(F_n) \geq f(n) \cdot \mathscr{C}_T(F_n)$. The refinement is *exponentially-proper* if $f(n) = 2^{n^{\Omega(1)}}$ and *super-polynomially-proper* if $f(n) = n^{\omega(1)}$.

Under this definition, tree-like, regular, linear, positive, negative, semantic, and ordered resolution are natural refinements of RES, and further, tree-like, regular, and ordered resolution are exponentially-proper [32, 6].

Now we are ready to state the somewhat technical theorem relating the clause learning process to resolution, whose corollaries are nonetheless easy to understand. The proof of this theorem is based on an explicit construction of so-called "proof-trace extension" formulas, which interestingly allow one to translate *any* known separation result between RES and a natural proper refinement S of RES into a separation between CL and S.

Theorem 1 ([21, 179])**.** *For any $f(n)$-proper natural refinement S of RES and for CL using the FirstNewCut scheme and no restarts, there exist formulas $\{F_n\}$ such that $\mathscr{C}_S(F_n) \geq f(n) \cdot \mathscr{C}_{\mathsf{CL}}(F_n)$.*

Corollary 1. *CL can provide exponentially shorter proofs than tree-like, regular, and ordered resolution.*

Corollary 2. *Either CL is not a natural proof system or it is equivalent in strength to RES.*

We remark that this leaves open the possibility that CL may not be able to simulate all regular resolution proofs. In this context, MacKenzie [145] has used arguments similar to those of Beame et al. [19] to prove that a natural variant of clause learning can indeed simulate all of regular resolution.

Finally, let CL-- denote the variant of CL where one is allowed to branch on a literal whose value is already set explicitly or because of unit propagation. Of course, such a

relaxation is useless in ordinary DPLL; there is no benefit in branching on a variable that doesn't even appear in the residual formula. However, with clause learning, such a branch can lead to an immediate conflict and allow one to learn a key conflict clause that would otherwise have not been learned. This property can be used to prove that RES can be efficiently simulated by CL-- with enough restarts. In this context, a clause learning scheme will be called *non-redundant* if on a conflict, it always learns a clause not already known. Most of the practical clause learning schemes are non-redundant.

Theorem 2 ([21, 179])**.** *CL-- with any non-redundant scheme and unlimited restarts is polynomially equivalent to RES.*

We note that by choosing the restart points in a smart way, CL together with restarts can be converted into a *complete* algorithm for satisfiability testing, i.e., for all unsatisfiable formulas given as input, it will halt and provide a proof of unsatisfiability [15, 92]. The theorem above makes a much stronger claim about a slight variant of CL, namely, with enough restarts, this variant can always find proofs of unsatisfiability that are as short as those of RES.

1.2.5 Symmetry Breaking

One aspect of many theoretical as well as real-world problems that merits attention is the presence of *symmetry* or *equivalence* amongst the underlying objects. Symmetry can be defined informally as a mapping of a constraint satisfaction problem (CSP) onto itself that preserves its structure as well as its solutions. The concept of symmetry in the context of SAT solvers and in terms of higher level problem objects is best explained through some examples of the many application areas where it naturally occurs. For instance, in FPGA (field programmable gate array) routing used in electronics design, all available wires or channels used for connecting two switch boxes are equivalent; in our design, it does not matter whether we use wire #1 between connector X and connector Y, or wire #2, or wire #3, or any other available wire. Similarly, in circuit modeling, all gates of the same "type" are interchangeable, and so are the inputs to a multiple fanin AND or OR gate (i.e., a gate with several inputs); in planning, all identical boxes that need to be moved from city A to city B are equivalent; in multi-processor scheduling, all available processors are equivalent; in cache coherency protocols in distributed computing, all available identical caches are equivalent. A key property of such objects is that when selecting k of them, we can choose, *without loss of generality*, any k. This without-loss-of-generality reasoning is what we would like to incorporate in an automatic fashion.

The question of symmetry exploitation that we are interested in addressing arises when instances from domains such as the ones mentioned above are translated into CNF formulas to be fed to a SAT solver. A CNF formula consists of constraints over different kinds of variables that typically represent tuples of these high level objects (e.g. wires, boxes, etc.) and their interaction with each other. For example, during the problem modeling phase, we could have a Boolean variable $z_{w,c}$ that is TRUE iff the first end of wire w is attached to connector c. When this formula is converted into DIMACS format for a SAT solver, the *semantic meaning* of the variables, that, say, variable 1324 is associated with wire #23 and connector #5, is discarded. Consequently, in this translation, the global notion of the obvious interchangeability of the set of wire objects is lost, and instead manifests itself indirectly as a symmetry between the (numbered) variables of the formula and therefore

also as a symmetry within the set of satisfying (or un-satisfying) variable assignments. These sets of symmetric satisfying and un-satisfying assignments artificially explode both the satisfiable and the unsatisfiable parts of the search space, the latter of which can be a challenging obstacle for a SAT solver searching for a satisfying assignment.

One of the most successful techniques for handling symmetry in both SAT and general CSPs originates from the work of Puget [169], who showed that symmetries can be *broken* by adding one lexicographic ordering constraint per symmetry. Crawford et al. [49] showed how this can be done adding a set of simple "lex-constraints" or *symmetry breaking predicates* (SBPs) to the input specification to weed out all but the lexically-first solutions. The idea is to identify the group of permutations of variables that keep the CNF formula unchanged. For each such permutation π, clauses are added so that for every satisfying assignment σ for the original problem, whose permutation $\pi(\sigma)$ is also a satisfying assignment, only the lexically-first of σ and $\pi(\sigma)$ satisfies the added clauses. In the context of CSPs, there has been a lot of work in the area of SBPs. Petrie and Smith [165] extended the idea to value symmetries, Puget [171] applied it to products of variable and value symmetries, and Walsh [207] generalized the concept to symmetries acting simultaneously on variables and values, on set variables, etc. Puget [170] has recently proposed a technique for creating dynamic lex-constraints, with the goal of minimizing adverse interaction with the variable ordering used in the search tree.

In the context of SAT, value symmetries for the high-level variables naturally manifest themselves as low-level variable symmetries, and work on SBPs has taken a different path. Tools such as `Shatter` by Aloul et al. [7] improve upon the basic SBP technique by using lex-constraints whose size is only linear in the number of variables rather than quadratic. Further, they use graph isomorphism detectors like `Saucy` by Darga et al. [50] to generate symmetry breaking predicates only for the generators of the algebraic groups of symmetry. This latter problem of computing graph isomorphism, however, is not known to have any polynomial time algorithms, and is conjectured to be strictly between the complexity classes P and NP [cf. 124]. Hence, one must resort to heuristic or approximate solutions. Further, while there are formulas for which few SBPs suffice, the number of SBPs one needs to add in order to break *all* symmetries can be exponential. This is typically handled in practice by discarding "large" symmetries, i.e., those involving too many variables with respect to a fixed threshold. This may, however, sometimes result in a much slower SAT solutions in domains such as clique coloring and logistics.

A very different and indirect approach for addressing symmetry is embodied in SAT solvers such as `PBS` by Aloul et al. [8], `pbChaff` by Dixon et al. [62], and `Galena` by Chai and Kuehlmann [41], which utilize non-CNF formulations known as pseudo-Boolean inequalities. Their logic reasoning are based on what is called the Cutting Planes proof system which, as shown by Cook et al. [47], is strictly stronger than resolution on which DPLL type CNF solvers are based. Since this more powerful proof system is difficult to implement in its full generality, pseudo-Boolean solvers often implement only a subset of it, typically learning only CNF clauses or restricted pseudo-Boolean constraints upon a conflict. Pseudo-Boolean solvers may lead to purely syntactic representational efficiency in cases where a single constraint such as $y_1 + y_2 + \ldots + y_k \leq 1$ is equivalent to $\binom{k}{2}$ binary clauses. More importantly, they are relevant to symmetry because they sometimes allow implicit encoding. For instance, the single constraint $x_1 + x_2 + \ldots + x_n \leq m$ over n variables captures the essence of the pigeonhole formula PHP_m^n over nm variables which is provably exponentially hard to solve using resolution-based methods without symmetry consider-

ations. This implicit representation, however, is not suitable in certain applications such as clique coloring and planning that we discuss. In fact, for unsatisfiable clique coloring instances, even pseudo-Boolean solvers provably require exponential time.

One could conceivably keep the CNF input unchanged but modify the solver to detect and handle symmetries during the search phase as they occur. Although this approach is quite natural, we are unaware of its implementation in a general purpose SAT solver besides `sEqSatz` by Li et al. [138], which has been shown to be effective on matrix multiplication and polynomial multiplication problems. Symmetry handling during search has been explored with mixed results in the CSP domain using frameworks like SBDD and SBDS [e.g. 66, 67, 79, 82]. Related work in SAT has been done in the specific areas of automatic test pattern generation by Marques-Silva and Sakallah [149] and SAT-based model checking by Shtrichman [192]. In both cases, the solver utilizes global information obtained at a stage to make subsequent stages faster. In other domain-specific work on symmetries in problems relevant to SAT, Fox and Long [68] propose a framework for handling symmetry in planning problems solved using the planning graph framework. They detect equivalence between various objects in the planning instance and use this information to reduce the search space explored by their planner. Unlike typical SAT-based planners, this approach does not guarantee plans of optimal length when multiple (non-conflicting) actions are allowed to be performed at each time step in parallel. Fortunately, this issue does not arise in the `SymChaff` approach for SAT to be mentioned shortly.

Dixon et al. [61] give a generic method of representing and dynamically maintaining symmetry in SAT solvers using algebraic techniques that guarantee polynomial size unsatisfiability proofs of many difficult formulas. The strength of their work lies in a strong group theoretic foundation and comprehensiveness in handling all possible symmetries. The computations involving group operations that underlie their current implementation are, however, often quite expensive.

When viewing complete SAT solvers as implementations of proof systems, the challenge with respect to symmetry exploitation is to push the underlying proof system up in the weak-to-strong proof complexity hierarchy without incurring the significant cost that typically comes from large search spaces associated with complex proof systems. While most of the current SAT solvers implement subsets of the resolution proof system, a different kind of solver called `SymChaff` [179, 180] brings it up closer to *symmetric resolution*, a proof system known to be exponentially stronger than resolution [202, 126]. More critically, it achieves this in a time- and space-efficient manner. Interestingly, while `SymChaff` involves adding structure to the problem description, it still stays within the realm of SAT solvers (as opposed to using a constraint programming (CP) approach), thereby exploiting the many benefits of the CNF form and the advances in state-of-the-art SAT solvers.

As a structure-aware solver, `SymChaff` incorporates several new ideas, including simple but effective symmetry representation, multiway branching based on variable classes and symmetry sets, and symmetric learning as an extension of clause learning to multiway branches. Two key places where it differs from earlier approaches are in using high level problem description to obtain symmetry information (instead of trying to recover it from the CNF formula) and in maintaining this information dynamically but without using a complex group theoretic machinery. This allows it to overcome many drawbacks of previously proposed solutions. It is shown, in particular, that straightforward annotation in the usual PDDL specification of planning problems is enough to automatically and quickly generate relevant symmetry information, which in turn makes the search for an op-

timal plan several orders of magnitude faster. Similar performance gains are seen in other domains as well.

1.3 SAT Solver Technology – Incomplete Methods

An *incomplete* method for solving the SAT problem is one that does not provide the guarantee that it will eventually either report a satisfying assignment or prove the given formula unsatisfiable. Such a method is typically run with a pre-set limit, after which it may or may not produce a solution. Unlike the systematic solvers based on an exhaustive branching and backtracking search, incomplete methods are based on *stochastic local search* or SLS. On many classes of problems, such incomplete methods for SAT significantly outperform DPLL-based methods. Since the early 1990's, there has been a tremendous amount of research on designing, understanding, and improving local search methods for SAT [e.g. 71, 94, 95, 99, 102, 103, 105, 139, 166, 186] as well as on hybrid approaches that attempt to combine DPLL and local search methods [e.g. 9, 96, 150, 175]. We begin this section by discussing two methods that played a key role in the success of local search in SAT, namely `GSAT` [191] and `Walksat` [189]. We will then explore the phase transition phenomenon in random SAT and a relatively new local search technique called Survey Propagation. We note that there are also solution techniques based on the discrete Lagrangian [205, 211] and on the interior point method [113], which we will not discuss.

The original impetus for trying a local search method on satisfiability problems was the successful application of such methods for finding solutions to large N-queens problems, first using a connectionist system by Adorf and Johnston [5], and then using greedy local search by Minton et al. [153]. It was originally assumed that this success simply indicated that N-queens was an *easy* problem, and researchers felt that such techniques would fail in practice for SAT. In particular, it was believed that local search methods would easily get stuck in local minima, with a few clauses remaining unsatisfied. The `GSAT` experiments showed, however, that certain local search strategies often do reach global minima, in many cases much faster than any systematic search strategies.

`GSAT` is based on a randomized local search technique [140, 162]. The basic `GSAT` procedure, described as Algorithm 1.3, starts with a randomly generated truth assignment. It then greedily changes ('flips') the assignment of the variable that leads to the greatest decrease in the total number of unsatisfied clauses. Such flips are repeated until either a satisfying assignment is found or a pre-set maximum number of flips (MAX-FLIPS) is reached. This process is repeated as needed, up to a maximum of MAX-TRIES times.

Selman et al. [191] showed that `GSAT` substantially outperformed even the best backtracking search procedures of the time on various classes of formulas, including randomly generated formulas and SAT encodings of graph coloring problems [112]. The search of `GSAT` typically begins with a rapid greedy descent towards a better assignment, followed by a long sequences of "sideways" moves. Each sequence of sideways moves is referred to as a *plateau*. Experiments indicate that in practice, `GSAT` spends most of its time moving from plateau to plateau, which motivates studying various modifications in order to speed up this process [187, 188]. One of the most successful strategies is to introduce noise into the search in the form of uphill moves, which forms the basis of the now well-known local search method for SAT called `Walksat` [189].

Algorithm 1.3: GSAT (F)

```
Input      : A CNF formula F
Parameters : Integers MAX-FLIPS, MAX-TRIES
Output     : A satisfying assignment for F, or FAIL
begin
    for i ← 1 to MAX-TRIES do
        σ ← a randomly generated truth assignment for F
        for j ← 1 to MAX-FLIPS do
            if σ satisfies F then return σ                    // success
            v ← a variable flipping which results in the greatest decrease
                  (possibly negative) in the number of unsatisfied clauses
            Flip v in σ
    return FAIL                          // no satisfying assignment found
end
```

`Walksat` interleaves the greedy moves of `GSAT` with random walk moves of a standard Metropolis search. It further focuses the search by always selecting the variable to flip from an (randomly chosen) unsatisfied clause C. If there is a variable in C flipping which does not turn any currently satisfied clauses to unsatisfied, it flips this variable (the "freebie" move). Otherwise, with a certain probability, it flips a random literal of C (the "random walk" move), and with the remaining probability, it flips a variable in C that minimizes the *break-count*, i.e., the number of currently satisfied clauses that become unsatisfied (the "greedy" move). `Walksat` in presented in detail as Algorithm 1.4. One of its parameters, in addition to the maximum number of tries and flips, is the *noise* $p \in [0,1]$, which controls how often are uphill moves considered during the stochastic search.

When one compares the biased random walk strategy of `Walksat` on hard random 3-CNF formulas against basic `GSAT`, the simulated annealing process of Kirkpatrick et al. [120], and a pure random walk strategy, the biased random walk process significantly outperforms the other methods [188]. In the years following the development of `Walksat`, many similar methods have been shown to be highly effective on not only random formulas but on many classes of structured instances, such as encodings of circuit design problems, Steiner tree problems, problems in finite algebra, and AI planning [cf. 105]. Various extensions of the basic process have also been explored, such as dynamic search policies like `adapt-novelty` [103], incorporating unit clause elimination as in the solver `UnitWalk` [99], and exploiting problem structure for increased efficiency [166]. Recently, it was shown that the performance of stochastic solvers on many structured problems can be further enhanced by using new SAT encodings that are designed to be effective for local search [168].

1.3.1 The Phase Transition Phenomenon in Random k-SAT

One of the key motivations in the early 1990's for studying incomplete, stochastic methods for solving SAT problems was the finding that `DPLL`-based systematic solvers perform quite poorly on certain randomly generated formulas. Consider a random k-CNF formula F on n variables generated by independently creating m clauses as follows: for each clause, select k distinct variables uniformly at random out of the n variables and negate each vari-

Algorithm 1.4: `Walksat` (F)

```
Input      : A CNF formula F
Parameters : Integers MAX-FLIPS, MAX-TRIES; noise parameter p ∈ [0,1]
Output     : A satisfying assignment for F, or FAIL
begin
    for i ← 1 to MAX-TRIES do
        σ ← a randomly generated truth assignment for F
        for j ← 1 to MAX-FLIPS do
            if σ satisfies F then return σ                          // success
            C ← an unsatisfied clause of F chosen at random
            if ∃ variable x ∈ C with break-count = 0 then
                v ← x                                          // the freebie move
            else
                With probability p:                       // the random walk move
                    v ← a variable in C chosen at random
                With probability 1 − p:                    // the greedy move
                    v ← a variable in C with the smallest break-count
            Flip v in σ
    return FAIL                                  // no satisfying assignment found
end
```

able with probability 0.5. When F is chosen from this distribution, Mitchell, Selman, and Levesque [154] observed that the median hardness of the problems is very nicely characterized by a key parameter: the *clause-to-variable ratio*, m/n, typically denoted by α. They observed that problem hardness peaks in a critically constrained region determined by α alone. The left pane of Figure 1.3 depicts the now well-known "easy-hard-easy" pattern of SAT and other combinatorial problems, as the key parameter (in this case α) is varied. For random 3-SAT, this region has been experimentally shown to be around $\alpha \approx 4.26$ (see [48, 121] for early results), and has provided challenging benchmarks as a test-bed for SAT solvers.

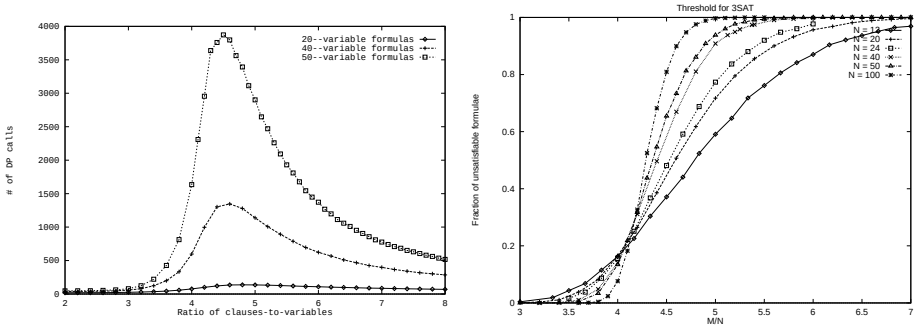


Figure 1.3: The phase transition phenomenon in random 3-SAT. Left: Computational hardness peaks at $\alpha \approx 4.26$. Right: Problems change from being mostly satisfiable to mostly unsatisfiable. The transitions sharpen as the number of variables grows.

This critically constrained region marks a stark transition not only in the computational hardness of random SAT instances but also in their satisfiability itself. The right pane of Figure 1.3 shows the fraction of random formulas that are unsatisfiable, as a function of α. We see that nearly all problems with α below the critical region (the under-constrained problems) are satisfiable. As α approaches and passes the critical region, there is a sudden change and nearly all problems in this over-constrained region are unsatisfiable. Further, as n grows, this phase transition phenomenon becomes sharper and sharper, and coincides with the region in which the computational hardness peaks. The relative hardness of the instances in the unsatisfiable region to the right of the phase transition is consistent with the formal result of Chvátal and Szemerédi [43] who, building upon the work of Haken [98], proved that large unsatisfiable random k-CNF formulas almost surely require exponential size resolution refutations, and thus exponential length runs of any DPLL-based algorithm proving unsatisfiability. This formal result was subsequently refined and strengthened by others [cf. 23, 20, 44].

Relating the phase transition phenomenon for 3-SAT to statistical physics, Kirkpatrick and Selman [121] showed that the threshold has characteristics typical of phase transitions in the statistical mechanics of disordered materials. Physicists have studied phase transition phenomena in great detail because of the many interesting changes in a system's macroscopic behavior that occur at phase boundaries. One useful tool for the analysis of phase transition phenomena is called *finite-size scaling* analysis. This approach is based on rescaling the horizontal axis by a factor that is a function of n. The function is such that the horizontal axis is stretched out for larger n. In effect, rescaling "slows down" the phase-transition for higher values of n, and thus gives us a better look inside the transition. From the resulting universal curve, applying the scaling function backwards, the actual transition curve for each value of n can be derived. This approach also localizes the 50%-satisfiable-point for any value of n, which allows one to generate the hardest possible random 3-SAT instances.

Interestingly, it is still not formally known whether there even exists a critical constant α_c such that as n grows, almost all 3-SAT formulas with $\alpha < \alpha_c$ are satisfiable and almost all 3-SAT formulas with $\alpha > \alpha_c$ are unsatisfiable. In this respect, Friedgut [72] provided the first positive result, showing that there exists a *function* $\alpha_c(n)$ depending on n such that the above threshold property holds. In a series of papers, researchers have narrowed down the gap between upper bounds on the threshold for 3-SAT [e.g. 70, 38, 122, 109, 63], the best so far being 4.596, and lower bounds [e.g. 69, 38, 73, 1, 4, 114, 97], the best so far being 3.52.

1.3.2 A New Technique for Random k-SAT: Survey Propagation

We end this section with a brief mention of Survey Propagation (SP), an exciting new algorithm for solving hard combinatorial problems. It was discovered in 2002 by Mezard, Parisi, and Zecchina [151], and is so far the only known method successful at solving random 3-SAT instances with one million variables and beyond in near-linear time in the most critically constrained region.

The SP method is quite radical in that it tries to approximate, using an iterative process of local "message" updates, certain marginal probabilities related to the set of satisfying assignments. It then assigns values to variables with the most extreme probabilities, simplifies the formula, and repeats the process. This strategy is referred to as SP-inspired dec-

imation. In effect, the algorithm behaves like the usual DPLL-based methods, which also assign variable values incrementally in an attempt to find a satisfying assignment. However, quite surprisingly, SP almost never has to backtrack. In other words, the "heuristic guidance" from SP is almost always correct. Note that, interestingly, computing marginals on satisfying assignments is strongly believed to be much harder than finding a single satisfying assignment (#P-complete vs. NP-complete). Nonetheless, SP is able to efficiently approximate certain marginals on random SAT instances and uses this information to successfully find a satisfying assignment.

SP was derived from rather complex statistical physics methods, specifically, the so-called *cavity method* developed for the study of spin glasses. The method is still far from well-understood, but in recent years, we are starting to see results that provide important insights into its workings [e.g. 152, 37, 11, 146, 3, 127]. Close connections to belief propagation (BP) methods [164] more familiar to computer scientists have been subsequently discovered. In particular, it was shown by Braunstein and Zecchina [37] (later extended by Maneva, Mossel, and Wainwright [146]) that SP equations are equivalent to BP equations for obtaining marginals over a special class of combinatorial objects, called covers. In this respect, SP is the first successful example of the use of a probabilistic reasoning technique to solve a purely combinatorial search problem. The recent work of Kroc et al. [127] empirically established that SP, despite the extremely loopy nature of random formulas which violate the standard tree-structure assumptions underlying the BP algorithm, is remarkably good at computing marginals over these covers objects on large random 3-SAT instances.

Unfortunately, the success of SP is currently limited to random SAT instances. It is an exciting research area to further understand SP and apply it successfully to more structured, real-world problem instances.

1.4 Runtime Variance and Problem Structure

The performance of backtrack-style search methods can vary dramatically depending on the way one selects the next variable to branch on (the "variable selection heuristic") and in what order the possible values are assigned to a variable (the "value selection heuristic"). The inherent exponential nature of the search process appears to magnify the unpredictability of search procedures. In fact, it is not uncommon to observe a backtrack search procedure "hang" on a given instance, whereas a different heuristic, or even just another randomized run, solves the instance quickly. A related phenomenon is observed in random problem distributions that exhibit an "easy-hard-easy" pattern in computational complexity, concerning so-called "exceptionally hard" instances: such instances seem to defy the "easy-hard-easy" pattern, they occur in the under-constrained area, but they seem to be considerably harder than other similar instances and even harder than instances from the critically constrained area. This phenomenon was first identified by Hogg and Willimans in graph coloring and by Gent and Walsh in satisfiability problems [78, 101]. An instance is considered to be exceptionally hard, for a particular search algorithm, when it occurs in the region where almost all problem instances are satisfiable (*i.e.*, the under constrained area), but, for a given algorithm, is considerably harder to solve than other similar instances, and even harder than most of the instances in the critically constrained area [78, 101, 194]. However, subsequent research showed that such instances are not inherently difficult; for example, by simply renaming the variables or by considering a different search heuristic

such instances can be easily solved [190, 195]. Therefore, the "hardness" of exceptionally hard instances does not reside in the instances *per se*, but rather in the combination of the instance with the details of the search method. This is the reason why researchers studying the hardness of computational problems use the median to characterize search difficulty, instead of the mean, since the behavior of the mean tends to be quite *erratic* [87].

1.4.1 Fat and Heavy Tailed behavior

The study of the full runtime distributions of search methods — instead of just the moments and median — has been shown to provide a better characterization of search methods and much useful information in the design of algorithms. In particular, researchers have shown that the runtime distributions of complete backtrack search methods reveal intriguing characteristics of such search methods: quite often complete backtrack search methods exhibit *fat* and *heavy-tailed* behavior [101, 87, 75].

The notion of *fat-tailedness* is based on the concept of *kurtosis*. The *kurtosis* is defined as μ_4/μ_2^2 (μ_4 is the fourth central moment about the mean and μ_2 is the second central moment about the mean, *i.e.*, the variance). If a distribution has a high central peak and long tails, than the kurtosis is in general large. The *kurtosis* of the standard normal distribution is 3. A distribution with a *kurtosis* larger than 3 is *fat-tailed* or *leptokurtic*. Examples of distributions that are characterized by *fat-tails* are the exponential distribution, the lognormal distribution, and the Weibull distribution.

Heavy-tailed distributions have "heavier" tails than fat-tailed distributions; in fact they have some infinite moments, *e.g.*, they can have infinite mean, or infinite variance, etc. More rigorously, a random variable X with probability distribution function $F(\cdot)$ is heavy-tailed if it has the so-called Pareto like decay of the tails, i.e.:

$$1 - F(x) = \Pr[X > x] \sim Cx^{-\alpha}, \quad x > 0,$$

where $\alpha > 0$ and $C > 0$ are constants. When $1 < \alpha < 2$, X has infinite variance, and infinite mean and variance when $0 < \alpha <= 1$. The log-log plot of $1 - F(x)$ of a Pareto-like distribution (i.e., the survival function) shows linear behavior with slope determined by α. Like *heavy*-tailed distributions, *fat*-tailed distributions have long tails, with a considerably mass of probability concentrated in the tails. Nevertheless, the tails of *fat*-tailed distributions are lighter than *heavy*-tailed distributions.

DPLL style complete backtrack search methods have been shown to exhibit heavy-tailed behavior, both in random instances and real-world instances. Examples domains are QCP [87], scheduling [89], planning[92], model checking, and graph coloring [206, 111]. Several formal models generating heavy-tailed behavior in search have been proposed [42, 209, 210, 111, 86]. If a runtime distribution of a backtrack search method is heavy-tailed, it will produce runs over several orders of magnitude, some extremely long but also some extremely short. Methods like randomization and restarts try to exploit this phenomenon.

1.4.2 Backdoors

Insight into heavy-tailed behavior comes from considering backdoor variables. These are variables which, when set, give us a polynomial subproblem. Intuitively, a small backdoor set explains how a backtrack search method can get "lucky" on certain runs, where backdoor variables are identified early on in the search and set the right way. Formally, the

definition of a backdoor depends on a particular algorithm, referred to as *sub-solver*, that solves a tractable sub-case of the general constraint satisfaction problem [209].

Definition 4. A *sub-solver A* given as input a CSP, *C*, satisfies the following:

i. Trichotomy: *A* either rejects the input *C*, or "determines" *C* correctly (as unsatisfiable or satisfiable, returning a solution if satisfiable),

ii. Efficiency: *A* runs in polynomial time,

iii. Trivial solvability: *A* can determine if *C* is trivially true (has no constraints) or trivially false (has a contradictory constraint),

iv. Self-reducibility: if *A* determines *C*, then for any variable *x* and value *v*, then *A* determines $C[v/x]$.[4]

For instance, *A* could be an algorithm that enforces arc consistency. Using the definition of sub-solver we can now formally define the concept of backdoor set. Let *A* be a sub-solver, and *C* be a CSP. A nonempty subset *S* of the variables is a *backdoor* in *C* for *A* if for some $a_S : S \to D$, *A* returns a satisfying assignment of $C[a_S]$. Intuitively, the backdoor corresponds to a set of variables, such that when set correctly, the sub-solver can solve the remaining problem. A stronger notion of the backdoor, considers both satisfiable and unsatisfiable (inconsistent) problem instances. A nonempty subset *S* of the variables is a *strong backdoor* in *C* for *A* if for all $a_S : S \to D$, *A* returns a satisfying assignment or concludes unsatisfiability of $C[a_S]$.

Szeider [199] considers the parameterized complexity of the problem of whether a SAT instance has a weak or strong backdoor set of size *k* or less for DPLL style sub-solvers, i.e., subsolvers based on unit propagation and/or pure literal elimination. He shows that detection of weak and strong backdoor sets is unlikely to be fixed-parameter tractable. Nishimura et al. [157] provide more positive results for detecting backdoor sets where the sub-solver solves Horn or 2-CNF formulas, both of which are linear time problems. They prove that the detection of such a strong backdoor set is fixed-parameter tractable, while the detection of a weak backdoor set is not. The explanation that they offer for such a discrepancy is quite interesting: for strong backdoor sets one only has to guarantee that the chosen set of variables gives a subproblem with the chosen syntactic class; for weak backdoor sets, one also has to guarantee satisfiability of the simplified formula, a property that cannot be described syntactically.

Dilkina et al. [60] study the tradeoff between the complexity of backdoor detection and the backdoor size. They prove that adding certain obvious inconsistency checks to the underlying class can make the complexity of backdoor detection jump from being within NP to being both NP-hard and coNP-hard. On the positive side, they show that this change can dramatically reduce the size of the resulting backdoors. They also explore the differences between so-called deletion backdoors and strong backdoors, in particular with respect to the class of renamable Horn formulas.

Concerning the size of backdoors, random formulas do not appear to have small backdoor sets. For example, for random 3-SAT problems, the backdoor set appears to be a constant fraction (roughly 30%) of the total number of variables [108]. This may explain why the current DPLL based solvers have not made significant progress on hard randomly

[4] We use $C[v/x]$ to denote the simplified CSP obtained by setting the value of variable *x* to *v* in *C*.

generated instances. Empirical results based on real-world instances suggest a more positive picture. Structured problem instances can have surprisingly small sets of backdoor variables, which may explain why current state of the art solvers are able to solve very large real-world instances. For example the logistics-d planning problem instance, (log.d) has a backdoor set of just 12 variables, compared to a total of nearly 7,000 variables in the formula, using the polynomial time propagation techniques of the SAT solver, Satz [135]. Hoffmann et al. [100] proved the existence of *strong* backdoor sets of size just $O(\log n)$ for certain families of logistics planning problems and blocks world problems.

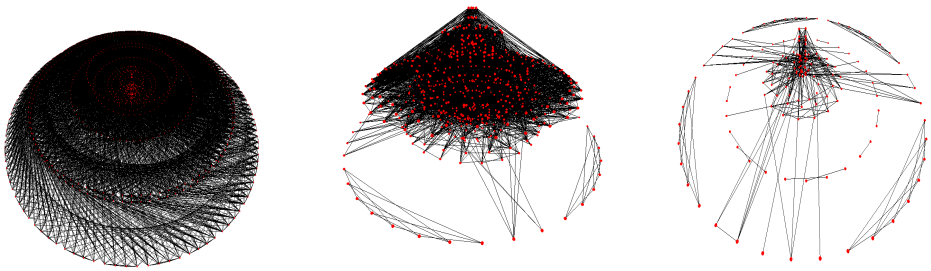


Figure 1.4: Constraint graph of a real-world instance from the logistics planning domain. The instance in the plot has 843 vars and 7,301 clauses. One backdoor set for this instance w.r.t. unit propagation has size 16 (not necessarily the minimum backdoor set). Left: Constraint graph of the original constraint graph of the instance. Center: Constraint graph after setting 5 variables and performing unit propagation on the graph. Right: Constraint graph after setting 14 variables and performing unit propagation on the graph.

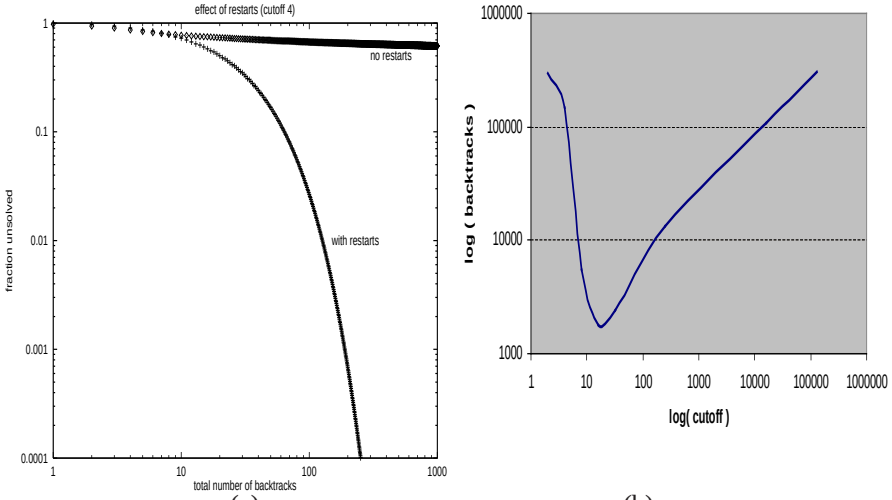
Even though computing backdoor sets is typically intractable [199], if we bound the size of the backdoor, heuristics and techniques like randomization and restarts may nevertheless be able to uncover a small backdoor in practice [119]. Dequen and Dubois introduced a heuristic for DPLL based solvers that exploits the notion of backbone that outperforms other heuristics on random 3-SAT problems [59, 64].

1.4.3 Restarts

One way to exploit heavy-tailed behavior is to add restarts to a backtracking procedure. A sequence of short runs instead of a single long run may be a more effective use of computational resources. Gomes et al. proposed a rapid randomization and restart (RRR) to take advantage of heavy-tailed behavior and boost the efficiency of complete backtrack search procedures [92]. In practice, one gradually increases the cutoff to maintain completeness ([92]). Gomes et al. have proved formally that a restart strategy with a fix cutoff eliminates heavy-tail behavior and therefore all the moments of a restart strategy are finite [88].

When the underlying runtime distribution of the randomized procedure is fully known, the optimal restart policy is a fixed cutoff [142]. When there is no *a priori* knowledge about the distribution, Luby *et al.* also provide a *universal strategy* which minimizes the expected cost. This consists of runs whose lengths are powers of two, and each time a pair of runs of a given length has been completed, a run of twice that length is immediately executed. The

universal strategy is of the form: $1,1,2,1,1,2,4,1,1,2,4,8,\cdots$. Although the universal strategy of Luby *et al.* is provably within a constant log factor of the optimal fixed cutoff, the schedule often converges too slowly in practice. Walsh introduced a restart strategy, inspired by Luby *et al.*'s analysis, in which the cutoff value increases geometrically [206]. The advantage of such a strategy is that it is less sensitive to the details of the underlying distribution. State-of-the-art SAT solvers now routinely use restarts. In practice, the solvers use a default cutoff value, which is increased, linearly, every given number of restarts, guaranteeing the completeness of the solver in the limit [155]. Another important feature is that they learn clauses across restarts.



(a) (b)

Figure 1.5: Restarts: (a) Tail $(1-F(x))$ as a function of the total number of backtracks for a QCP instance, log-log scale; the left curve is for a cutoff value of 4; and, the right curve is without restarts. (b) The effect of different cutoff values on solution cost for the logistics.d planning problem. Graph adapted from [87, 88].

In reality, we will be somewhere between full and no knowledge of the runtime distribution. Horvitz et al. [106] introduce a Bayesian framework for learning predictive models of randomized backtrack solvers based on this situation. Extending that work, Kautz et al. [115] considered restart policies that can factor in information based on real-time observations about a solver's behavior. In particular, they introduce an *optimal* policy for dynamic restarts that considers observations about solver behavior. They also consider the dependency between runs. They give a dynamic programming approach to generate the optimal restart strategy, and combine the resulting policy with real-time observations to boost performance of backtrack search methods.

Variants of restart strategies include randomized backtracking [143], and the random jump strategy [213] which has been used to solve a dozen previously open problems in finite algebra. Finally, one can also take advantage of the high variance of combinatorial search methods by combining several algorithms into a "portfolio," and running them in parallel or interleaving them on a single processor.

1.5 Beyond SAT: Quantified Boolean Formulas and Model Counting

We end this chapter with a brief overview of two important problems that extend beyond propositional satisfiability testing and will lie at the heart of the next generation automated reasoning systems: Quantified Boolean Formula (QBF) reasoning and counting the number of models (solutions) of a problem. These problems present fascinating challenges and pose new research questions. Efficient algorithms for these will have a significant impact on many application areas that are inherently beyond SAT, such as adversarial and contingency planning, unbounded model checking, and probabilistic reasoning.

These problems can be solved, in principle and to some extent in practice, by extending the two most successful frameworks for SAT algorithms, namely, DPLL and local search. However, there are some interesting issues and choices that arise when extending SAT-based techniques to these harder problems. In general, these problems require the solver to, in a sense, be cognizant of *all solutions* in the search space, thereby reducing the effectiveness and relevance of commonly used SAT heuristics designed for quickly zooming in on a single solution. The resulting scalability challenge has drawn many satisfiability researchers to these problems.

1.5.1 QBF Reasoning

A Quantified Boolean Formula (QBF) is a Boolean formula in which variables are quantified as existential ($\exists$) or universal ($\forall$). We will use the term QBF for *totally quantified (also known as closed) Boolean formulas in prenex form* beginning (for simplicity) with $\exists$:

$$F \;=\; \exists x_1^1 \ldots \exists x_1^{t(1)}\, \forall x_2^1 \ldots \forall x_2^{t(2)} \;\ldots\; Q x_k^1 \ldots Q x_k^{t(k)} \;\; M$$

where M is a Boolean formula referred to as the *matrix* of F, x_i^j above are distinct and include all variables appearing in M, and Q is $\exists$ if k is odd and $\forall$ if k is even. Defining $V_i = \left\{x_i^1, \ldots x_i^{t(i)}\right\}$ and using associativity within each level of quantification, we can simplify the notation to $F = \exists V_1 \forall V_2 \exists V_3 \ldots Q V_k \; M$. A QBF solver is an algorithm that determines the truth value of such formulas F, i.e., whether there exist values of variables in V_1 such that for every assignment of values to variables in V_2 there exist values of variables in V_3, and so on, such that M is satisfied (i.e., evaluates to TRUE).

QBF reasoning extends the scope of SAT to domains requiring adversarial analysis, like conditional planning [172], unbounded model checking [174, 25], and discrete games [81]. As a simple applied example, consider a two-player game where each player has a discrete set of actions. Here a winning strategy for a player is a partial game tree that, for every possible game play of the opponent, indicates how to proceed so as to guarantee a win. This kind of reasoning is more complex than the single-agent reasoning that SAT solvers offer, and requires modeling and analyzing adversarial actions of another agent with competing interests. Fortunately, such problems are easily and naturally modeled using QBF. The QBF approach thus supports a much richer setting than SAT. However, it also poses new and sometimes unforeseen challenges.

In terms of the worst-case complexity, deciding the truth of a QBF is PSPACE-complete [198] whereas SAT is "only" NP-complete.[5] Even with very few quantification levels, the

[5] Assuming P $\neq$ NP, PSPACE-complete problems are significantly harder than NP-complete problems; cf. [163].

explosion in the search space is tremendous in practice. Further, as the winning strategy example indicates, even a solution to a QBF may require exponential space to describe, causing practical difficulties [24].

Nonetheless, several tools for deciding the truth of a given QBF (QBF solvers) have been developed. These include DPLL-style search based solvers like `Quaffle` [217], `QuBE` [83], `Semprop` [131], `Evaluate` [40], `Decide` [173], and `QRSat` [160]; local search methods like `WalkQSAT` [80]; skolemization based solvers like `sKizzo` [25]; q-resolution [123] based solvers like `Quantor` [27]; and symbolic, BDD based tools like `QMRES` and `QBDD` [161]. Most of these solvers extend the concepts underlying SAT solvers. In particular, they inherit conjunctive normal form (CNF) as the input representation, which has been the standard for SAT solvers for over a decade. Internally, some solvers also employ disjunctive normal form (DNF) to cache partial solutions for efficiency [218].

We focus here on DPLL-based QBF solvers. The working of these solvers is not very different from that of DPLL-based SAT solvers. The essential difference is that when the DPLL process branches on an universal variable x by setting it to TRUE and finds that branch to be satisfiable, it must also verify that the branch $x = \text{FALSE}$ is also satisfiable. The need to be able to do this "universal reasoning" and explore both branches of universal variables has, as expected, a substantial impact on the efficiency of the solver.

In a series of papers, Zhang and Malik [217], Letz [131], and Giunchiglia et al. [84] described how the clause learning techniques from SAT can be extended to *solution learning* for QBF. The idea is to not only cache small certificates of unsatisfiability of sub-formulas (as learned CNF clauses), but also to cache small certificates of satisfiability of sub-formulas (as learned DNF "terms", also referred to as *cubes*). This can, in principle, be very useful because not only does a QBF solver need to detect unsatisfiability efficiently, it needs to also detect satisfiability efficiently and repeatedly.

Another interesting change, which is now part of most QBF solvers, is related to unit propagation. This stems from the observation that if the variables with the deepest quantification level in a clause are universal, they cannot help satisfy that clause. The clause can effectively ignore these universal variables. This also plays a role in determining which clauses are learned upon reaching a conflict, and also has a dual counterpart about existential variables in a DNF term.

While the performance of QBF solvers has been promising, translating a QBF into a (much larger) SAT specification and using a good SAT solver is often faster in practice — a fact well-recognized and occasionally exploited [27, 25, 182]. This motivates the need for further investigation into the design of QBF solvers and possible fundamental weaknesses in the modeling methods used.

It has been recently demonstrated by Samulowitz et al. that the efficiency of QBF solvers can be improved significantly – much more so than SAT solvers – by employing certain pre-processing techniques to the formula at the very beginning [184] or using inference techniques, such as those based on binary clauses, on the fly [183]. These methods typically involve adding a certain type of easy-to-compute resolvents as redundant constraints to the problem, with the hope of achieving faster propagation. Results show that this works very well in practice.

Any QBF reasoning task has a natural game playing interpretation at a high level. Using this fact, Ansotegui et al. [10] describe a general framework for modeling adversarial tasks as QBF instances. They view a problem P as a two-player game G with a bounded number of turns. This is different from the standard interpretation of a QBF as a game

[163]; in their approach, one must formulate the higher level problem *P* as a game *G before* modeling it as a QBF. The sets of "rules" to which the existential and universal players of *G* are bound may differ from one player to the other. Ansotegui et al. [10] observe that typical CNF-based encodings for QBF suffer from the "illegal search space issue" where the solver finds it artificially hard to detect certain illegal moves made by the universal player. They propose the use of special indicator variables that flag the occurrence of such illegal moves, which is then exploited by their solver to prune the search space.

Another recent proposal by Sabharwal et al. [181], implemented in the QBF solver `Duaffle` which extends `Quaffle`, is a new generic QBF modeling technique that uses a dual CNF-DNF representation. The dual representation considers the above game-theoretic view of the problem. The key idea is to exploit a dichotomy between the players: rules for the existential player are modeled as CNF clauses, (the negations of) rules for the universal player modeled as DNF terms, and game state information split equally into clauses and terms. This symmetric dual format places "equal responsibility" on the two players, in stark contrast with other QBF encodings which tend to leave most work for the existential player. This representation has several advantages over pure-CNF encodings for QBF. In particular, it allows unit propagation *across quantifiers* and avoids the illegal search space issue altogether.

An independent dual CNF-DNF approach of Zhang [215] converts a full CNF encoding into a logically equivalent full DNF encoding and provides both to the solver. In contrast, `Duaffle` exploits the representational power of DNF to simplify the model and make it more compact, while addressing some issues associated with pure CNF representations. Both of these dual CNF-DNF approaches are different from fully non-clausal encodings, which also have promise but are unable to directly exploit rapid advances in CNF-based SAT solvers. Recently, Benedetti et al. [26] have proposed "restricted quantification" for pure-CNF encodings for QCSPs. This general technique addresses the illegal search space issue and is applicable also to QBF solvers other than those that are search based.

1.5.2 Model Counting

Propositional model counting is the problem of computing the number of models for a given propositional formula, i.e., the number of distinct variable assignments for which the formula evaluates to TRUE. This problem generalizes SAT and is known to be a #P-complete problem, which means that it is no easier than solving a QBF with an unbounded number of "there exist" and "forall" quantifiers in its variables [200]. For comparison, notice that SAT can be thought of as a QBF with exactly one level of "there exist" quantification.

Effective model counting procedures would open up a range of new applications. For example, various probabilistic inference problems, such as Bayesian net reasoning, can be effectively translated into model counting problems [cf. 176, 141, 52, 13]. Another application is in the study of hard combinatorial problems, such as combinatorial designs, where the number of solutions provides further insights into the problem. Even finding a single solution can be a challenge for such problems: counting the number of solutions is much harder yet. Not surprisingly, the largest formulas we can solve the model counting problem with state-of-the-art model counters are significantly smaller than the formulas we can solve with the best SAT solvers.

The earliest practical approach for counting models is based on an extension of systematic DPLL-based SAT solvers. The idea is to simply explore the complete search tree for an n-variable formula, associating 2^t solutions with a search tree branch if that branch leads to a solution at decision level $n-t$. By using appropriate multiplication factors and continuing the search after a single solution is found, `Relsat` [17] is able to provide incremental lower bounds on the model count as it proceeds, and finally computes the exact model count. Newer tools such as `Cachet` [185] often improve upon this by using techniques such as component caching [19].

Another approach to model counting is to convert the formula into a form from which the count can be deduced easily. The tool `c2d` [51] uses this knowledge compilation technique to convert the given CNF formula into decomposable negation normal form (DDNF) [53] and compute the model count.

All exact counting methods, especially those based on DPLL search, essentially attack a #P-complete problem "head on" — by searching the raw combinatorial search space. Consequently, these algorithms often have difficulty scaling up to larger problem sizes. We should point out that problems with a higher solution count are not necessarily harder to determine the model count of. In fact, `Relsat` can compute the true model count of highly under-constrained problems with many "don't care" variables and a lot of models by exploiting big clusters in the solution space. The model counting problem is instead much harder for more intricate combinatorial problems where the solutions are spread much more finely throughout the combinatorial space.

Wei and Selman [208] use Markov Chain Monte Carlo (MCMC) sampling to compute an approximation of the true model count. Their model counter, `ApproxCount`, is able to solve several instances quite accurately, while scaling much better than both `Relsat` and `Cachet` as problem size increases. The drawback of `ApproxCount` is that one is not able to provide any hard guarantees on the model count it computes. To output a number close to the true count, this counting strategy requires uniform sampling from the set of solutions, which is generally very difficult to achieve. Uniform sampling from the solution space is much harder than just generating a single solution. MCMC methods can provide theoretical convergence guarantees but only in the limit, generally after an exponential number of steps.

Interestingly, the inherent strength of most state-of-the-art SAT solvers comes actually from the ability to quickly narrow down to a certain portion of the search space the solver is designed to handle best. Such solvers therefore sample solutions in a highly non-uniform manner, making them seemingly ill-suited for model counting, unless one forces the solver to explore the full combinatorial space. An intriguing question is whether there is a way around this apparent limitation of the use of state-of-the-art SAT solvers for model counting.

`MBound` [90] is a new method for model counting, which interestingly uses any complete SAT solver "as is." It follows immediately that the more efficient the SAT solver used, the more powerful its counting strategy becomes. `MBound` is inspired by recent work on so-called "streamlining constraints" [91], in which additional, non-redundant constraints are added to the original problem to increase constraint propagation and to focus the search on a small part of the subspace, (hopefully) still containing solutions. This strategy was earlier shown to be successful in solving very hard combinatorial design problems, with carefully created, domain-specific streamlining constraints. In contrast, `MBound` uses a domain-independent streamlining technique.

The central idea of the approach is to use a special type of randomly chosen constrains as streamliners, namely XOR or parity constraints on the problem variables. Such constraints require that an odd number of the involved variables be set to TRUE. (This requirement can be translated into the usual CNF form by using additional variables [201].) MBound works by repeatedly adding a number s of such constraints to the formula and feeding the result to a state-of-the-art complete SAT solver. At a very high level, each random XOR constraint will cut the search space approximately in half. So, intuitively, if after the addition of s XOR's the formula is still satisfiable, the original formula must have at least of the order of 2^s models. More rigorously, it can be shown that if we perform t experiments of adding s random XOR constraints and our formula remains satisfiable in each case, then with probability at least $1-2^{-\alpha t}$, our original formula will have at least $2^{s-\alpha}$ satisfying assignments for any $\alpha \geq 1$. As a result, by repeated experiments or by weakening the claimed bound, one can arbitrarily boost the confidence in the lower bound count. Similar results can also be derived for the upper bound. A surprising feature of this approach is that it does not depend at all on the how the solutions are distributed throughout the search space. It relies on the very special properties of random parity constraints, which in effect provide a good hash function, randomly dividing the solutions into two near-equal sets. Such constraints were first used by Valiant and Vazirani [203] in a randomized reduction from SAT to the related problem Unique SAT.

Bibliography

[1] D. Achlioptas. Setting 2 variables at a time yields a new lower bound for random 3-SAT. In *32st STOC*, pages 28–37, Portland,OR, May 2000.

[2] D. Achlioptas, P. Beame, and M. Molloy. A sharp threshold in proof complexity. In *33rd STOC*, pages 337–346, Crete, Greece, July 2001.

[3] D. Achlioptas and F. Ricci-Tersenghi. On the solution-space geometry of random constraint satisfaction problems. In *38th STOC*, pages 130–139, Seattle, WA, May 2006.

[4] D. Achlioptas and G. Sorkin. Optimal myopic algorithms for random 3-SAT. In *41st FOCS*, pages 590–600, Redondo Beach, CA, Nov. 2000. IEEE.

[5] H. Adorf and M. Johnston. A discrete stochastic neural network algorithm for constraint satisfaction problems. In *Intl. Joint Conf. on Neural Networks*, pages 917–924, San Diego, CA, 1990.

[6] M. Alekhnovich, J. Johannsen, T. Pitassi, and A. Urquhart. An exponential separation between regular and general resolution. In *34th STOC*, pages 448–456, Montréal, Canada, May 2002.

[7] F. A. Aloul, I. L. Markov, and K. A. Sakallah. Shatter: Efficient symmetry-breaking for Boolean satisfiability. In *40th DAC*, pages 836–839, Anahein, CA, June 2003.

[8] F. A. Aloul, A. Ramani, I. L. Markov, and K. A. Sakallah. PBS: A backtrack-search pseudo-Boolean solver and optimizer. In *5th SAT*, pages 346–353, Cincinnati, OH, May 2002.

[9] Anbulagan, D. N. Pham, J. K. Slaney, and A. Sattar. Old resolution meets modern SLS. In *20th AAAI*, pages 354–359, Pittsburgh, PA, July 2005.

[10] C. Ansotegui, C. P. Gomes, and B. Selman. The Achilles' heel of QBF. In *20th AAAI*, pages 275–281, Pittsburgh, PA, July 2005.

[11] E. Aurell, U. Gordon, and S. Kirkpatrick. Comparing beliefs, surveys, and random walks. In *17th NIPS*, Vancouver, Canada, Dec. 2004.

[12] F. Bacchus. Enhancing Davis Putnam with extended binary clause reasoning. In *18th AAAI*, pages 613–619, Edmonton, Canada, July 2002.

[13] F. Bacchus, S. Dalmao, and T. Pitassi. Algorithms and complexity results for #SAT and Bayesian inference. In *44nd FOCS*, pages 340–351, Cambridge, MA, Oct. 2003.

[14] F. Bacchus and J. Winter. Effective preprocessing with hyper-resolution and equality reduction. In *6th SAT*, volume 2919 of *LNCS*, pages 341–355, Santa Margherita, Italy, May 2003.

[15] L. Baptista and J. P. Marques-Silva. Using randomization and learning to solve hard real-world instances of satisfiability. In *6th CP*, pages 489–494, Singapore, Sept. 2000.

[16] R. J. Bayardo Jr. and D. P. Miranker. A complexity analysis of space-bounded learning algorithms for the constraint satisfaction problem. In *13th AAAI*, pages 298–304, Portland, OR, Aug. 1996.

[17] R. J. Bayardo Jr. and J. D. Pehoushek. Counting models using connected components. In *17th AAAI*, pages 157–162, Austin, TX, July 2000.

[18] R. J. Bayardo Jr. and R. C. Schrag. Using CSP look-back techniques to solve real-world SAT instances. In *14th AAAI*, pages 203–208, Providence, RI, July 1997.

[19] P. Beame, R. Impagliazzo, T. Pitassi, and N. Segerlind. Memoization and DPLL: Formula caching proof systems. In *Proc., 18th Annual IEEE Conf. on Comput. Complexity*, pages 225–236, Aarhus, Denmark, July 2003.

[20] P. Beame, R. Karp, T. Pitassi, and M. Saks. On the Complexity of Unsatisfiability Proofs for Random *k*-CNF Formulas. In *30th STOC*, pages 561–571, Dallas, TX, May 1998.

[21] P. Beame, H. Kautz, and A. Sabharwal. Understanding and harnessing the potential of clause learning. *JAIR*, 22:319–351, Dec. 2004.

[22] P. Beame and T. Pitassi. Propositional Proof Complexity: Past, Present, Future. In *Current Trends in Theoretical Computer Science*, pages 42–70. World Scientific, 2001.

[23] P. W. Beame and T. Pitassi. Simplified and improved resolution lower bounds. In *37th FOCS*, pages 274–282, Burlington, VT, Oct. 1996. IEEE.

[24] M. Benedetti. Extracting certificates from quantified Boolean formulas. In *19th IJCAI*, pages 47–53, Edinburgh, Scotland, July 2005.

[25] M. Benedetti. sKizzo: a suite to evaluate and certify QBFs. In *20th CADE*, volume 3632 of *LNCS*, pages 369–376, Tallinn, Estonia, July 2005.

[26] M. Benedetti, A. Lallouet, and J. Vautard. QCSP made practical by virtue of restricted quantification. In *20th IJCAI*, pages 38–43, Hyderabad, India, Jan. 2007.

[27] A. Biere. Resolve and expand. In *7th SAT*, volume 3542 of *LNCS*, pages 59–70, Vancouver, BC, Canada, May 2004. Selected papers.

[28] A. Biere, A. Cimatti, E. M. Clarke, M. Fujita, and Y. Zhu. Symbolic model checking using SAT procedures instead of BDDs. In *36th DAC*, pages 317–320, New Orleans, LA, June 1999.

[29] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *5th TACAS*, pages 193–207, Amsterdam, the Netherlands, Mar. 1999.

[30] P. Bjesse, T. Leonard, and A. Mokkedem. Finding bugs in an alpha microprocessor using satisfiability solvers. In *Proc. 13th Int. Conf. on Computer Aided Verification*, 2001.

[31] M. Böhm and E. Speckenmeyer. A fast parallel SAT-solver – efficient workload balancing. *Annals of Math. and AI*, 17(3-4):381–400, 1996.

[32] M. L. Bonet, J. L. Esteban, N. Galesi, and J. Johansen. On the relative complexity of resolution refinements and cutting planes proof systems. *SIAM J. Comput.*, 30(5):1462–1484, 2000.

[33] M. L. Bonet and N. Galesi. Optimality of size-width tradeoffs for resolution. *Comput. Complx.*, 10(4):261–276, 2001.

[34] R. J. Brachman and H. J. Levesque. The tractability of subsumption in frame based description languages. In *AAAI'84*, pages 34–37, 1984.

[35] R. J. Brachman and H. J. Levesque, editors. *Readings in Knowledge Representation*. Morgan Kaufmann, 1985.

[36] R. J. Brachman and J. Schmolze. An overview of the KL-ONE knowledge representation system. *Cognitive Science*, 9(2):171–216, 1985.

[37] A. Braunstein and R. Zecchina. Survey propagation as local equilibrium equations. *J. Stat. Mech.*, P06007, 2004. URL http://lanl.arXiv.org/cond-mat/0312483.

[38] A. Broder, A. Frieze, and E. Upfal. On the satisfiability and maximum satisfiability of random 3-CNF formulas. In *Proc., 4th SODA*, Jan. 1993.

[39] J. Buresh-Oppenheim and T. Pitassi. The complexity of resolution refinements. In *18th Annual IEEE Symp. on Logic in Comput. Sci.*, pages 138–147, Ottawa, Canada, June 2003.

[40] M. Cadoli, M. Schaerf, A. Giovanardi, and M. Giovanardi. An algorithm to evaluate quantified Boolean formulae and its experimental evaluation. *J. Auto. Reas.*, 28(2):101–142, 2002.

[41] D. Chai and A. Kuehlmann. A fast pseudo-Boolean constraint solver. In *40th DAC*, pages 830–835, Anahein, CA, June 2003.

[42] H. Chen, C. Gomes, and B. Selman. Formal models of heavy-tailed behavior in combinatorial search. In *7th CP*, 2001.

[43] V. Chvátal and E. Szemerédi. Many hard examples for resolution. *J. Assoc. Comput. Mach.*, 35(4):759–768, 1988.

[44] M. Clegg, J. Edmonds, and R. Impagliazzo. Using the Gröbner basis algorithm to find proofs of unsatisfiability. In *28th STOC*, pages 174–183, Philadelphia, PA, May 1996.

[45] S. A. Cook. The complexity of theorem proving procedures. In *Conf. Record of 3rd STOC*,

pages 151–158, Shaker Heights, OH, May 1971.

[46] S. A. Cook and R. A. Reckhow. The relative efficiency of propositional proof systems. *J. Symb. Logic*, 44(1):36–50, 1977.

[47] W. Cook, C. R. Coullard, and G. Turan. On the complexity of cutting plane proofs. *Discr. Applied Mathematics*, 18:25–38, 1987.

[48] J. Crawford and L. Auton. Experimental results on the cross-over point in satisfiability problems. In *Proc. AAAI-93*, pages 21–27, Washington, DC, 1993.

[49] J. M. Crawford, M. L. Ginsberg, E. M. Luks, and A. Roy. Symmetry-breaking predicates for search problems. In *5th KR*, pages 148–159, Cambridge, MA, Nov. 1996.

[50] P. T. Darga, M. H. Liffiton, K. A. Sakallah, and I. L. Markov. Exploiting structure in symmetry detection for CNF. In *41st DAC*, pages 518–522, San Diego, CA, June 2004.

[51] A. Darwiche. New advances in compiling CNF into decomposable negation normal form. In *Proc., 16th Euro. Conf. on AI*, pages 328–332, Valencia, Spain, Aug. 2004.

[52] A. Darwiche. The quest for efficient probabilistic inference, July 2005. Invited Talk, IJCAI-05.

[53] A. Darwiche and P. Marquis. A knowledge compilation map. *JAIR*, 17:229–264, 2002.

[54] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem proving. *CACM*, 5:394–397, 1962.

[55] M. Davis and H. Putnam. A computing procedure for quantification theory. *CACM*, 7:201–215, 1960.

[56] R. Davis. Diagnostic reasoning based on structure and behavior. *J. AI*, 24(1-3):347–410, 1984.

[57] J. de Kleer and B. C. Williams. Diagnosing multiple faults. *J. AI*, 32(1):97–130, 1987.

[58] R. Dechter. Learning while searching in constraint-satisfaction-problems. In *5th AAAI*, pages 178–185, Philadelphia, PA, Aug. 1986.

[59] G. Dequen and O. Dubois. Kcnfs: An efficient solver for random k-SAT formulae. In *6th SAT*, 2003.

[60] B. Dilkina, C. P. Gomes, and A. Sabharwal. Tradeoffs in the complexity of backdoor detection. In *13th CP*, Providence, RI, Sept. 2007.

[61] H. E. Dixon, M. L. Ginsberg, E. M. Luks, and A. J. Parkes. Generalizing Boolean satisfiability II: Theory. *JAIR*, 22:481–534, 2004.

[62] H. E. Dixon, M. L. Ginsberg, and A. J. Parkes. Generalizing Boolean satisfiability I: Background and survey of existing work. *JAIR*, 21:193–243, 2004.

[63] O. Dubois, Y. Boufkhad, and J. Mandler. Typical random 3-SAT formulae and the satisfiability threshold. In *Proc., 11th SODA*, pages 126–127, San Francisco, CA, Jan. 2000.

[64] O. Dubois and G. Dequen. A backbone search heuristic for efficient solving of hard 3-SAT formulae. In *18th IJCAI*, 2003.

[65] N. Eén and N. Sörensson. MiniSat: A SAT solver with conflict-clause minimization. In *8th SAT*, St. Andrews, U.K., June 2005.

[66] T. Fahle, S. Schamberger, and M. Sellmann. Symmetry breaking. In *7th CP*, volume 2239 of *LNCS*, pages 93–107, Paphos, Cyprus, Nov. 2001.

[67] F. Focacci and M. Milano. Global cut framework for removing symmetries. In *7th CP*, volume 2239 of *LNCS*, pages 77–92, Paphos, Cyprus, Nov. 2001.

[68] M. Fox and D. Long. The detection and exploitation of symmetry in planning problems. In *16th IJCAI*, pages 956–961, July 1999.

[69] J. Franco. Probabilistic analysis of the pure literal heuristic for the satisfiability problem. *Annals of Operations Research*, 1:273–289, 1983.

[70] J. Franco and M. Paull. Probabilistic analysis of the Davis-Putnam procedure for solving the satisfiability problem. *Discr. Applied Mathematics*, 5:77–87, 1983.

[71] J. Frank, P. Cheeseman, and J. Stutz. Where gravity fails: Local search topology. *JAIR*, 7: 249–281, 1997.

[72] E. Friedgut. Sharp thresholds of graph properties, and the *k*-sat problem. *Journal of the American Mathematical Society*, 12:1017–1054, 1999.

[73] A. Frieze and S. Suen. Analysis of two simple heuristics on a random instance of k- SAT. *J. Alg.*, 20(2):312–355, 1996.

[74] D. Frost and R. Dechter. Dead-end driven learning. In *12th AAAI*, pages 294–300, Seattle, WA, Aug. 1994.

[75] D. Frost, I. Rish, and L. Vila. Summarizing CSP hardness with continuous probability distributions. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence (AAAI-97)*, pages 327–334, New Providence, RI, 1997. AAAI Press.

[76] M. Gelfond. Answer sets. In F. van Harmelen, V. Lifschitz, and B. Porter, editors, *The Handbook of Knowledge Representation*. Elsevier, Oxford, 2006.

[77] M. R. Genesereth. The use of design descriptions in automated diagnosis. *J. AI*, 24(1-3): 411–436, 1984.

[78] I. Gent and T. Walsh. Easy problems are sometimes hard. *J. AI*, 70:335–345, 1994.

[79] I. P. Gent, W. Harvey, T. Kelsey, and S. Linton. Generic sbdd using computational group theory. In *8th CP*, volume 2833 of *LNCS*, pages 333–347, Kinsale, Ireland, Sept. 2003.

[80] I. P. Gent, H. H. Hoos, A. G. D. Rowley, and K. Smyth. Using stochastic local search to solver quantified Boolean formulae. In *8th CP*, volume 2833 of *LNCS*, pages 348–362, Kinsale, Ireland, Sept. 2003.

[81] I. P. Gent and A. G. Rowley. Encoding Connect-4 using quantified Boolean formulae. In *2nd Intl. Work. Modelling and Reform. CSP*, pages 78–93, Kinsale, Ireland, Sept. 2003.

[82] I. P. Gent and B. M. Smith. Symmetry breaking in constraint programming. In *Proc., 14th Euro. Conf. on AI*, pages 599–603, Berlin, Germany, Aug. 2000.

[83] E. Giunchiglia, M. Narizzano, and A. Tacchella. QUBE: A system for deciding quantified Boolean formulas satisfiability. In *1st IJCAR*, volume 2083 of *LNCS*, pages 364–369, Siena, Italy, June 2001.

[84] E. Giunchiglia, M. Narizzano, and A. Tacchella. Learning for quantified Boolean logic satisfiability. In *18th AAAI*, pages 649–654, Edmonton, Canada, July 2002.

[85] E. Goldberg and Y. Novikov. BerkMin: A fast and robust sat-solver. In *DATE*, pages 142–149, Paris, France, Mar. 2002.

[86] C. Gomes, C. Fernandez, B. Selman, and C. Bessiere. Statistical regimes across constrainedness regions. In *10th CP*, 2004.

[87] C. Gomes, B. Selman, and N. Crato. Heavy-tailed distributions in combinatorial search. In *3rd CP*, pages 121–135, 1997.

[88] C. Gomes, B. Selman, N. Crato, and H. Kautz. Heavy-tailed phenomena in satisfiability and constraint satisfaction problems. *J. Auto. Reas.*, 24(1/2):67–100, 2000.

[89] C. Gomes, B. Selman, K. McAloon, and C. Tretkoff. Randomization in backtrack search: Exploiting heavy-tailed profiles for solving hard scheduling problems. In *4th Int. Conf. Art. Intel. Planning Syst.*, 1998.

[90] C. P. Gomes, A. Sabharwal, and B. Selman. Model counting: A new strategy for obtaining good bounds. In *21th AAAI*, pages 54–61, Boston, MA, July 2006.

[91] C. P. Gomes and M. Sellmann. Streamlined constraint reasoning. In *10th CP*, volume 3258 of *LNCS*, pages 274–289, Toronto, Canada, Oct. 2004.

[92] C. P. Gomes, B. Selman, and H. Kautz. Boosting combinatorial search through randomization. In *15th AAAI*, pages 431–437, Madison, WI, July 1998.

[93] C. P. Gomes, B. Selman, K. McAloon, and C. Tretkoff. Randomization in backtrack search: Exploiting heavy-tailed profiles for solving hard scheduling problems. In *4th Int. Conf. Art. Intel. Planning Syst.*, pages 208–213, Pittsburgh, PA, June 1998.

[94] J. Gu. Efficient local search for very large-scale satisfiability problems. *Sigart Bulletin*, 3(1): 8–12, 1992.

[95] J. Gu, P. W. Purdom, J. Franco, and B. J. Wah. Algorithms for the Satisfiability (SAT) Problem:

A Survey. In *Satisfiability (SAT) Problem*, DIMACS, pages 19–151. American Mathematical Society, 1997.

[96] D. Habet, C. M. Li, L. Devendeville, and M. Vasquez. A hybrid approach for SAT. In *8th CP*, volume 2470 of *LNCS*, pages 172–184, Ithaca, NY, Sept. 2002.

[97] M. Hajiaghayi and G. Sorkin. The satisfiability threshold for random 3-SAT is at least 3.52, 2003. URL http://arxiv.org/abs/math/0310193.

[98] A. Haken. The intractability of resolution. *Theoretical Comput. Sci.*, 39:297–305, 1985.

[99] E. A. Hirsch and A. Kojevnikov. UnitWalk: A new SAT solver that uses local search guided by unit clause elimination. *Annals Math. and AI*, 43(1):91–111, 2005.

[100] J. Hoffmann, C. Gomes, and B. Selman. Structure and problem hardness: Asymmetry and DPLL proofs in SAT-based planning. In *11th CP*, 2005.

[101] T. Hogg and C. Williams. Expected gains from parallelizing constraint solving for hard problems. In *Proceedings of the Twelfth National Conference on Artificial Intelligence (AAAI-94)*, pages 1310–1315, Seattle, WA, 1994. AAAI Press.

[102] H. Hoos. On the run-time behaviour of stochastic local search algorithms for SAT. In *Proceedings of AAAI-99*, pages 661–666. AAAI Press, 1999.

[103] H. H. Hoos. An adaptive noise mechanism for WalkSAT. In *18th AAAI*, pages 655–660, Edmonton, Canada, July 2002.

[104] H. H. Hoos and T. Stützle. SATLIB: An online resource for research on SAT. In I. P. Gent, H. van Maaren, and T. Walsh, editors, *SAT2000*, pages 283–292. IOS Press, 2000. URL http://www.satlib.org.

[105] H. H. Hoos and T. Stützle. *Stochastic Local Search: Foundations and Applications*. Morgan Kaufmann, San Francisco, CA, USA, 2004.

[106] E. Horvitz, Y. Ruan, C. Gomes, H. Kautz, B. Selman, and D. Chickering. A bayesian approach to tackling hard computational problems. In *17th UAI*, 2001.

[107] M. Huele, J. van Zwieten, M. Dufour, and H. van Maaren. March-eq: Implementing additional reasoning into an efficient lookahead SAT solver. In *7th SAT*, volume 3542 of *LNCS*, pages 345–359, Vancouver, BC, Canada, May 2004. Springer.

[108] Y. Interian. Backdoor sets for random 3-SAT. In *6th SAT*, 2003.

[109] S. Janson, Y. C. Stamatiou, and M. Vamvakari. Bounding the unsatisfiability threshold of random 3-SAT. *Random Struct.s and Alg.*, 17(2):103–116, 2000.

[110] R. G. Jeroslow and J. Wang. Solving propositional satisfiability problems. *Annals of Math. and AI*, 1(1-4):167–187, 1990.

[111] H. Jia and C. Moore. How much backtracking does it take to color random graphs? rigorous results on heavy tails. In *10th CP*, 2004.

[112] D. Johnson, C. Aragon, L. McGeoch, and C. Schevon. Optimization by simulated annealing: an experimental evaluation; part ii. *Operations Research*, 39, 1991.

[113] A. Kamath, N. Karmarkar, K. Ramakrishnan, and M. Resende. Computational experience with an interior point algorithm on the satisfiability problem. In *Proceedings of Integer Programming and Combinatorial Optimization*, pages 333–349, Waterloo, Canada, 1990. Mathematical Programming Society.

[114] A. C. Kaporis, L. M. Kirousis, and E. G. Lalas. The probabilistic analysis of a greedy satisfiability algorithm. *Random Structures and Algorithms*, 28(4):444–480, 2006.

[115] H. Kautz, E. Horvitz, Y. Ruan, C. Gomes, and B. Selman. Dynamic restart policies. In *18th AAAI*, 2002.

[116] H. Kautz and B. Selman. The state of SAT. *Discrete Applied Mathematics*, 155(12):1514–1524, 2007.

[117] H. A. Kautz and B. Selman. Planning as satisfiability. In *Proc., 10th Euro. Conf. on AI*, pages 359–363, Vienna, Austria, Aug. 1992.

[118] H. A. Kautz and B. Selman. Pushing the envelope: Planning, propositional logic, and stochastic search. In *13th AAAI*, pages 1194–1201, Portland, OR, Aug. 1996.

[119] P. Kilby, J. Slaney, S. Thiebaux, and T. Walsh. Backbones and backdoors in satisfiability. In *20th AAAI*, 2005.

[120] S. Kirkpatrick, D. Gelatt Jr., and M. Vecchi. Optimization by simuleated annealing. *Science*, 220(4598):671–680, 1983.

[121] S. Kirkpatrick and B. Selman. Critical behavior in the satisfiability of random boolean expressions. *Science*, 264:1297–1301, May 1994.

[122] L. M. Kirousis, E. Kranakis, and D. Krizanc. Approximating the unsatisfiability threshold of random formulas. In *Proceedings of the Fourth Annual European Symposium on Algorithms*, pages 27–38, Barcelona, Spain, Sept. 1996.

[123] H. Kleine-Büning, M. Karpinski, and A. Flögel. Resolution for quantified Boolean formulas. *Information and Computation*, 117(1):12–18, 1995.

[124] J. Köbler, U. Schöning, and J. Torán. *The Graph Isomorphism Problem: its Structural Complexity*. Birkhauser Verlag, 1993. ISBN 0-8176-3680-3.

[125] H. Konuk and T. Larrabee. Explorations of sequential ATPG using Boolean satisfiability. In *11th VLSI Test Symposium*, pages 85–90, 1993.

[126] B. Krishnamurthy. Short proofs for tricky formulas. *Acta Inf.*, 22:253–274, 1985.

[127] L. Kroc, A. Sabharwal, and B. Selman. Survey propagation revisited. In *23rd UAI*, Vancouver, BC, July 2007. To appear.

[128] D. Le Berre, O. Roussel, and L. Simon (Organizers). SAT 2007 competition, May 2007. URL http://www.satcompetition.org/2007.

[129] D. Le Berre and L. Simon (Organizers). SAT 2004 competition, May 2004. URL http://www.satcompetition.org/2004.

[130] D. Le Berre and L. Simon (Organizers). SAT 2005 competition, June 2005. URL http://www.satcompetition.org/2005.

[131] R. Letz. Lemma and model caching in decision procedures for quantified Boolean formulas. In *Proc. of the TABLEAUX*, volume 2381 of *LNCS*, pages 160–175, Copenhagen, Denmark, July 2002.

[132] H. J. Levesque and R. J. Brachman. A fundamental tradeoff in knowledge representation and reasoning. In R. J. Brachman and H. J. Levesque, editors, *Readings in Knowledge Representation*, pages 41–70. Morgan Kaufmann, 1985.

[133] H. J. Levesque and R. J. Brachman. Expressiveness and tractability in knowledge representation and reasoning. *Computational Intelligence*, 3(2):78–93, 1987.

[134] L. Levin. Universal sequential search problems. *Problems of Information Transmission*, 9(3): 265–266, 1973. Originally in Russian.

[135] C. Li and Anbulagan. Heuristics based on unit propagation for satisfiability problems. In *15th IJCAI*, 1997.

[136] C. M. Li. Integrating equivalency reasoning into Davis-Putnam procedure. In *17th AAAI*, pages 291–296, Austin, TX, July 2000.

[137] C. M. Li and Anbulagan. Heuristics based on unit propagation for satisfiability problems. In *15th IJCAI*, pages 366–371, Nagoya, Japan, Aug. 1997.

[138] C. M. Li, B. Jurkowiak, and P. W. Purdom. Integrating symmetry breaking into a DLL procedure. In *SAT*, pages 149–155, Cincinnati, OH, May 2002.

[139] X. Y. Li, M. F. M. Stallmann, and F. Brglez. QingTing: A local search sat solver using an effective switching strategy and an efficient unit propagation. In *6th SAT*, pages 53–68, Santa Margherita, Italy, May 2003.

[140] S. Lin and B. Kernighan. An efficient heuristic algorithm for the traveling-salesman problem. *Operations Research*, 21:498–516, 1973.

[141] M. L. Littman, S. M. Majercik, and T. Pitassi. Stochastic Boolean satisfiability. *J. Auto. Reas.*, 27(3):251–296, 2001.

[142] M. Luby, A. Sinclair, and D. Zuckerman. Optimal speedup of Las Vegas algorithms. *Inf. Process. Lett.*, 47:173–180, 1993.

[143] I. Lynce, L. Baptista, and J. Marques-Silva. Stochastic systematic search algorithms for satisfiability. In *4th SAT*, 2001.

[144] I. Lynce and J. P. Marques-Silva. An overview of backtrack search satisfiability algorithms. *Annals Math. and AI*, 37(3):307–326, 2003.

[145] P. D. MacKenzie, July 2005. Private communication.

[146] E. N. Maneva, E. Mossel, and M. J. Wainwright. A new look at survey propagation and its generalizations. In *16th SODA*, pages 1089–1098, Vancouver, Canada, Jan. 2005.

[147] J. P. Marques-Silva. The impact of branching heuristics in propositional satisfiability algorithms. In *9th Portuguese Conf. on AI*, volume 1695 of *LNCS*, pages 62,74, Portugal, Sept. 1999.

[148] J. P. Marques-Silva and K. A. Sakallah. GRASP – a new search algorithm for satisfiability. In *ICCAD*, pages 220–227, San Jose, CA, Nov. 1996.

[149] J. P. Marques-Silva and K. A. Sakallah. Robust search algorithms for test pattern generation. In *27th FTCS*, pages 152–161, Seattle, WA, June 1997.

[150] B. Mazure, L. Sais, and E. Gregoire. Boosting complete techniques thanks to local search methods. In *Proc. Math and AI*, 1996.

[151] M. Mezard, G. Parisi, and R. Zecchina. Analytic and Algorithmic Solution of Random Satisfiability Problems. *Science*, 297(5582):812–815, 2002.

[152] M. Mézard and R. Zecchina. Random k-satisfiability problem: From an analytic solution to an efficient algorithm. *Phy. Rev. E*, 66:056126, Nov. 2002.

[153] S. Minton, M. Johnston, A. Philips, and P. Laird. Solving large-scale constraint satisfaction an scheduling problems using a heuristic repair method. In *Proceedings AAAI-90*, pages 17–24. AAAI Press, 1990.

[154] D. Mitchell, B. Selman, and H. Levesque. Hard and easy distributions of sat problems. In *Proc. AAAI-92*, pages 459–465, San Jose, CA, 1992.

[155] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *38th DAC*, pages 530–535, Las Vegas, NV, June 2001.

[156] A. Nadel. The Jerusat SAT solver. Master's thesis, Hebrew University of Jerusalem, 2002.

[157] N. Nishimura, P. Ragde, and S. Szeider. Detecting backdoor sets with respect to horn and binary clauses. In *7th SAT*, 2004.

[158] E. Nudelman, A. Devkar, Y. Shoham, K. Leyton-Brown, and H. H. Hoos. SATzilla: An algorithm portfolio for SAT, 2004. In conjunction with SAT-04.

[159] R. Ostrowski, E. Grégoire, B. Mazure, and L. Sais. Recovering and exploiting structural knowledge from cnf formulas. In *8th CP*, volume 2470 of *LNCS*, pages 185–199, Ithaca, NY, Sept. 2002. SV.

[160] C. Otwell, A. Remshagen, and K. Truemper. An effective QBF solver for planning problems. In *Proc. MSV/AMCS*, pages 311–316, Las Vegas, NV, June 2004.

[161] G. Pan and M. Y. Vardi. Symbolic decision procedures for QBF. In *10th CP*, number 3258 in LNCS, pages 453–467, Toronto, Canada, Sept. 2004.

[162] C. Papadimitriou and K. Steiglitz. *Combinatorial Optimization*. Prentice-Hall, Inc., 1982.

[163] C. H. Papadimitriou. *Computational Complexity*. Addison-Wesley, 1994.

[164] J. Pearl. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufmann, 1988.

[165] K. E. Petrie and B. M. Smith. Symmetry breaking in graceful graphs. In *8th CP*, volume 2833 of *LNCS*, pages 930–934, Kinsale, Ireland, Sept. 2003.

[166] D. N. Pham, J. Thornton, and A. Sattar. Building structure into local search for SAT. In *20th IJCAI*, pages 2359–2364, Hyderabad, India, Jan. 2007.

[167] K. Pipatsrisawat and A. Darwiche. RSat 1.03: SAT solver description. Technical Report D–152, Automated Reasoning Group, Computer Science Department, UCLA, 2006.

[168] S. D. Prestwich. Variable dependency in local search: Prevention is better than cure. In *10th SAT*, Lisbon, Portugal, May 2007.

[169] J.-F. Puget. On the satisfiability of symmetrical constrained satisfaction problems. In *Int. Symp. on Method. for Intel. Syst.*, volume 689 of *LNCS*, pages 350–361, Trondheim, Norway, June 1993.

[170] J.-F. Puget. Dynamic lex constraints. In *12th CP*, volume 4204 of *LNCS*, pages 453–467, Sept. 2006.

[171] J.-F. Puget. An efficient way of breaking value symmetries. In *21th AAAI*, Boston, MA, July 2006.

[172] J. Rintanen. Constructing conditional plans by a theorem prover. *JAIR*, 10:323–352, 1999.

[173] J. Rintanen. Improvements to the evaluation of quantified Boolean formulae. In *16th IJCAI*, pages 1192–1197, Stockholm, Sweden, July 1999.

[174] J. Rintanen. Partial implicit unfolding in the Davis-Putnam procedure for quantified Boolean formulae. In *8th Intl. Conf. Logic for Prog., AI, and Reason.*, volume 2250 of *LNCS*, pages 362–376, Havana, Cuba, Dec. 2001.

[175] I. Rish and R. Dechter. To guess or to think? hybrid algorithms for SAT. In *Proceedings of the Conference on Principles of Constraint Programming (CP-96)*, pages 555–556, 1996.

[176] D. Roth. On the hardness of approximate reasoning. *J. AI*, 82(1-2):273–302, 1996.

[177] S. J. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 2nd edition, 2002.

[178] L. Ryan. Efficient algorithms for clause-learning SAT solvers. Master's thesis, University of British Columbia, Vancouver, 2003.

[179] A. Sabharwal. *Algorithmic Applications of Propositional Proof Complexity*. PhD thesis, University of Washington, Seattle, 2005.

[180] A. Sabharwal. SymChaff: A structure-aware satisfiability solver. In *20th AAAI*, pages 467–474, Pittsburgh, PA, July 2005.

[181] A. Sabharwal, C. Ansotegui, C. P. Gomes, J. W. Hart, and B. Selman. QBF modeling: Exploiting player symmetry for simplicity and efficiency. In *9th SAT*, volume 4121 of *LNCS*, pages 353–367, Seattle, WA, Aug. 2006.

[182] H. Samulowitz and F. Bacchus. Using SAT in QBF. In *11th CP*, volume 3709 of *LNCS*, pages 578–592, Sitges, Spain, Oct. 2005.

[183] H. Samulowitz and F. Bacchus. Binary clause reasoning in QBF. In *9th SAT*, volume 4121 of *LNCS*, pages 353–367, Seattle, WA, Aug. 2006.

[184] H. Samulowitz, J. Davies, and F. Bacchus. Preprocessing QBF. In *12th CP*, volume 4204 of *LNCS*, pages 514–529, Nantes, France, Sept. 2006.

[185] T. Sang, F. Bacchus, P. Beame, H. A. Kautz, and T. Pitassi. Combining component caching and clause learning for effective model counting. In *7th SAT*, Vancouver, Canada, May 2004.

[186] D. Schuurmans and F. Southey. Local search characteristics of incomplete SAT procedures. In *Proc. of the 17th National Conf. on Articial Intelligence (AAAI-2000)*, pages 297–302, 2000.

[187] B. Selman and H. Kautz. Domain-independent extensions to GSAT: Solving large structured satisfiability problems. In *13th IJCAI*, pages 290–295, France, 1993.

[188] B. Selman, H. Kautz, and B. Cohen. Noise strategies for local search. In *Proc. AAAI-94*, pages 337–343, Seattle, WA, 1994.

[189] B. Selman, H. Kautz, and B. Cohen. Local search strategies for satisfiability testing. In D. S. Johnson and M. A. Trick, editors, *Cliques, Coloring, and Satisfiability: the Second DIMACS Implementation Challenge. DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, volume 26, pages 521–532. American Mathematical Society, 1996.

[190] B. Selman and S. Kirkpatrick. Finite-Size Scaling of the Computational Cost of Systematic Search. *Artificial Intelligence*, 81(1–2):273–295, 1996.

[191] B. Selman, H. J. Levesque, and D. G. Mitchell. A new method for solving hard satisfiability problems. In *10th AAAI*, pages 440–446, San Jose, CA, July 1992.

[192] O. Shtrichman. Accelerating bounded model checking of safety properties. *Form. Meth. in Sys. Des.*, 1:5–24, 2004.

[193] C. Sinz (Organizer). SAT-race 2006, Aug. 2006. URL http://fmv.jku.at/sat-race-2006.

[194] B. M. Smith and S. A. Grant. Sparse constraint graphs and exceptionally hard problems. In *14th IJCAI*, volume 1, pages 646–654, Montreal, Canada, Aug. 1995.

[195] B. M. Smith and S. A. Grant. Modelling exceptionally hard constraint satisfaction problems. In *3rd CP*, volume 1330 of *LNCS*, pages 182–195, Austria, Oct. 1997.

[196] R. M. Stallman and G. J. Sussman. Forward reasoning and dependency-directed backtracking in a system for computer-aided circuit analysis. *J. AI*, 9:135–196, 1977.

[197] P. R. Stephan, R. K. Brayton, and A. L. Sangiovanni-Vincentelli. Combinatorial test generation using satisfiability. *IEEE Trans. CAD and IC*, 15(9):1167–1176, 1996.

[198] L. J. Stockmeyer and A. R. Meyer. Word problems requiring exponential time. In *Conf. Record of 5th STOC*, pages 1–9, Austin, TX, Apr.-May 1973.

[199] S. Szeider. Backdoor sets for DLL solvers. *J. Auto. Reas.*, 2006. Special issue on SAT 2005. To appear.

[200] S. Toda. On the computational power of PP and $\oplus$P. In *30th FOCS*, pages 514–519, 1989.

[201] G. S. Tseitin. On the complexity of derivation in the propositional calculus. In A. O. Slisenko, editor, *Studies in Constructive Mathematics and Mathematical Logic, Part II*. 1968.

[202] A. Urquhart. The symmetry rule in propositional logic. *Discr. Applied Mathematics*, 96-97: 177–193, 1999.

[203] L. G. Valiant and V. V. Vazirani. NP is as easy as detecting unique solutions. *Theoretical Comput. Sci.*, 47(3):85–93, 1986.

[204] M. N. Velev and R. E. Bryant. Effective use of Boolean satisfiability procedures in the formal verification of superscalar and vliw microprocessors. *J. Symb. Comput.*, 35(2):73–106, 2003.

[205] B. W. Wah and Y. Shang. A discrete Lagrangian-based global-search method for solving satisfiability problems. *J. of Global Optimization*, 12(1):61–99, 1998.

[206] T. Walsh. Search in a small world. In *16th IJCAI*, 1999.

[207] T. Walsh. General symmetry breaking constraints. In *12th CP*, volume 4204 of *LNCS*, pages 650–664, Sept. 2006.

[208] W. Wei and B. Selman. A new approach to model counting. In *8th SAT*, volume 3569 of *LNCS*, pages 324–339, St. Andrews, U.K., June 2005.

[209] R. Williams, C. Gomes, and B. Selman. Backdoors to typical case complexity. In *18th IJCAI*, 2003.

[210] R. Williams, C. Gomes, and B. Selman. On the connections between backdoors, restarts, and heavy-tailedness in combinatorial search. In *6th SAT*, 2003.

[211] Z. Wu and B. W. Wah. Trap escaping strategies in discrete Lagrangian methods for solving hard satisfiability and maximum satisfiability problems. In *16th AAAI*, pages 673–678, Orlando, FL, July 1999.

[212] H. Zhang. SATO: An efficient propositional prover. In *14th CADE*, volume 1249 of *LNCS*, pages 272–275, Townsville, Australia, July 1997.

[213] H. Zhang. A random jump strategy for combinatorial search. In *International Symposium on AI and Math*, Fort Lauderdale, FL, 2002.

[214] H. Zhang and J. Hsiang. Solving open quasigroup problems by propositional reasoning. In *Proceedings of the International Computer Symp.*, Hsinchu, Taiwan, 1994.

[215] L. Zhang. Solving QBF by combining conjunctive and disjunctive normal forms. In *21th AAAI*, pages 143–149, Boston, MA, July 2006.

[216] L. Zhang, C. F. Madigan, M. H. Moskewicz, and S. Malik. Efficient conflict driven learning in a Boolean satisfiability solver. In *ICCAD*, pages 279–285, San Jose, CA, Nov. 2001.

[217] L. Zhang and S. Malik. Conflict driven learning in a quantified Boolean satisfiability solver. In *ICCAD*, pages 442–449, San Jose, CA, Nov. 2002.

[218] L. Zhang and S. Malik. Towards a symmetric treatment of satisfaction and conflicts in quantified Boolean formula evaluation. In *8th CP*, pages 200–215, Ithaca, NY, Sept. 2002.