Handbook of Knowledge Representation
Edited by F. van Harmelen, V. Lifschitz and B. Porter

DOI: 10.1016/S1574-6526(07)03002-7

Chapter 2

Satisfiability Solvers

Carla P. Gomes, Henry Kautz, Ashish Sabharwal, 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 [29–31, 228], automatic test pattern generation [138, 221], planning [129, 197], scheduling [103], and even challenging problems from algebra [238]. Annual SAT competitions have led to the development of dozens of clever implementations of such solvers (e.g., [13, 19, 71, 93, 109, 118, 150, 152, 161, 165, 170, 171, 173, 174, 184, 198, 211, 213, 236]), an exploration of many new techniques (e.g., [15, 102, 149, 170, 174]), and the creation of an extensive suite of real-world instances as well as challenging hand-crafted benchmark problems (cf. [115]). 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 [128, 129]. 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 [30]. Another example of a recent application of SAT solvers is in computing stable models used in the answer set programming paradigm, a powerful knowledge representation and reasoning approach [81]. In these applications—planning, verification, and answer set programming—the translation into a propositional representation (the "SAT encoding") is done automatically and is 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 [36–38, 145, 146]. For a review of the state of the art in this area, 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 [168, 213]. In the first [168], 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 [213] 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 [233].) Given these developments, it may be worthwhile to reconsider

[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 [141–143, 215]. 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.

the study of complexity tradeoffs in knowledge representation languages by not insisting on worst-case polynomial time reasoning but allowing 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.

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 [127].

2.1 Definitions and Notation

A propositional or Boolean formula is a logic expressions defined over variables (or atoms) that take value in the set {FALSE, 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 [51, 147]. 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 therefore 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, while 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 [225] 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 $(z \vee d)$.

2.2 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. A key feature of DPLL is 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.[3]

2.2.1 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 [61] came up with the basic idea behind this procedure. However, it was only a couple of years later that Davis, Logamann, and Loveland [60] 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 2.1, `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.

[3] Due to space limitation, we cannot do justice to a large amount of recent work on complete SAT solvers, which consists of hundreds of publications. The aim of this section is to give the reader an overview of several techniques commonly employed by these solvers.

Algorithm 2.1: 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, ρ)
begin
    while F contains no empty clause but has a unit clause x do
        F ← F|x
        ρ ← ρ ∪ {x}
    return (F, ρ)
end
```

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 2.1; 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, during 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.

2.2.2 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 propagation 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 next.

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., [160] 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 [121] or the BOHM heuristic (cf. [32]). One could select and fix the literal occurring most frequently in the yet unsatisfied clauses (the DLIS (Dynamic Largest Individual Sum) heuristic [161]), 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 [170]. Newer solvers like `BerkMin` [93], `Jerusat` [171], `MiniSat` [71], and `RSat` [184] 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 2.2.3, which will be devoted entirely 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. [170], 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 [236] 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 be 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 perform 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 [220], 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` [161] and `zChaff` [170]. It lets a solver 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 [171] in the 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 [71] 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. [102], 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` [170], employ 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 detail later in the chapter.

2.2.3 Clause Learning and Iterative DPLL

Algorithm 2.2 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 2.2: `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.[4] 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

[4] 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 truth values and no conflict is created by this assignment.

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 [220], Genesereth [82], Davis [62], Dechter [64], de Kleer and Williams [63], 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, Bayardo Jr. and Miranker [17], Marques-Silva and Sakallah [161], Bayardo Jr. and Schrag [19], Zhang [236], Moskewicz et al. [170], Zhang et al. [240], 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 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 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 2.1. *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 $C = (l_1 \vee \cdots l_k \vee l)$ 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.*

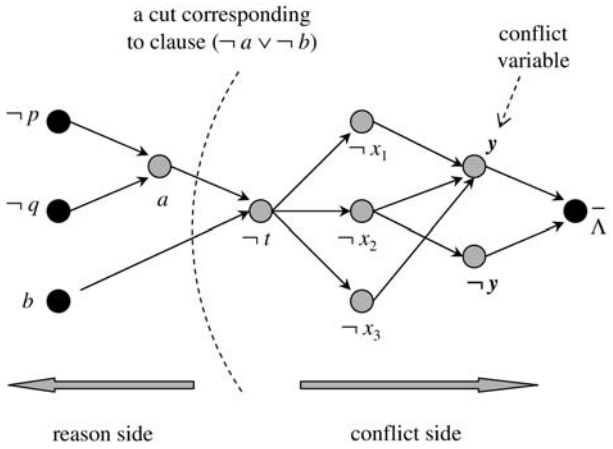

Figure 2.1: A conflict graph.

(ii) *Add edges* (l_i, l), $1 \leqslant i \leqslant k$, *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. Note that we are using $\overline{\Lambda}$ to denote the node of the implication graph representing a conflict, and Λ to denote the empty clause.

By definition, the 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 the implication graph, called the *conflict graph* (see Fig. 2.1), that captures only one among possibly many ways of reaching a conflict from the decision variables using unit propagation.

Definition 2.2. *A conflict graph* H *is any subgraph of the 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 $(l_1 \vee l_2 \vee \cdots \vee l_k \vee l)$ is a known clause.*

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 2.2, 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 Fig. 2.2). One may also define learning schemes based on cuts not involving conflict literals at all such as a scheme suggested by Zhang et al. [240], 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 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 [240], for example, uses the cut whose reason side comprises all decision variables. `Relsat` [19] 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 Fig. 2.2, 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.

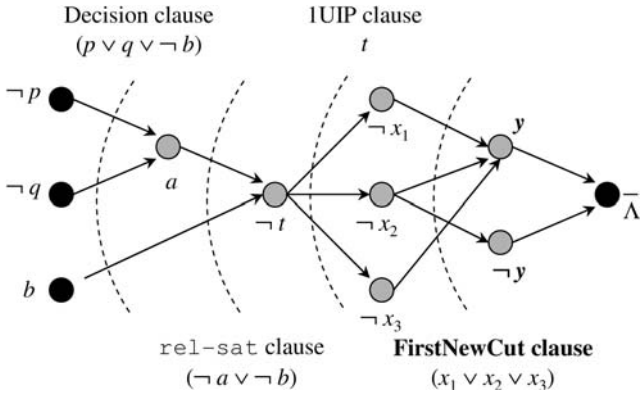

Figure 2.2: Learning schemes corresponding to different cuts in the conflict graph.

This nice flipping property holds in general for all *unique implication points* (UIPs) [161]. A UIP of an implication graph is a node at the current decision level d such that every path from the decision variable at level d to the conflict variable or 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` [161] and `zChaff` [170] 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. [240] 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 [22]. 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.

2.2.4 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 [52] 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