2007

TR-2007008: The Satisfiability Problem—From the Theory of NP-Completeness to State-of-the-Art SAT Solvers

Rave Harpaz

The Satisfiability Problem- From The Theory of NP-completeness to State-Of-The-Art SAT Solvers

Rave Harpaz
Pattern Recognition Laboratory
Department of Computer Science
The Graduate Center
The City University of New York
365 Fifth Avenue
New York, N.Y. 10016

December 1, 2003

Abstract

Given a formula of the propositional calculus, determining whether there exists a variable assignment such that the formula evaluates to true under the usual rules of interpretation is called the Boolean Satisfiability Problem, commonly abbreviated as SAT. SAT has a central role in the theory of computational complexity as it was the first computational task shown to be NP-complete. Subsequent problems have been shown to belong to the same family by proving they are at least as hard as SAT. Roughly, a task is NPcomplete if a good algorithm for it would entail a good algorithm for SAT. Thus, despite its appealing simplicity, one can think of SAT as the "core" problem in this family of hard problems.

SAT is of special concern to AI because of its direct connection to reasoning and theorem-proving. Deductive reasoning is simply the complement of satisfiability, and researches first became interested in SAT precisely for this reason. Nonetheless, in recent years there has been an explosion of interest in SAT because more and more practical problems in AI that utilize other forms of reasoning such as *constraint-based* reasoning with applications in *planning*, *configuration*, *diagnosis*, *resource allocation*, *scheduling*, and *electronic design automation*, make direct appeal to satisfiability, i.e., can easily be represented as SAT problems.

The first SAT solver is traditionally attributed to Davis and Putnam in 1960. Since then a wealth of algorithms have been developed and several approaches have been proposed, including variations of backtrack search, local search, continuous formulations and algebraic manipulation. In this report we introduce SAT through the theory of NP-completeness, and follow it with a broad overview of the different techniques and algorithms used to solve the SAT problem, noting that given vast amount of algorithms that have been proposed only the classical and most prominent are discussed. Evaluating the performance of SAT algorithms has become a problem in itself, which has yielded interesting insights. We conclude this report with an overview of the different methods used to evaluate the performance of SAT algorithms.

Contents

Chapter 1

Introduction

Given a formula of the propositional calculus, determining whether there exists a variable assignment such that the formula evaluates to true under the usual rules of interpretation is called the Boolean Satisfiability Problem, commonly abbreviated as SAT. SAT has a central role in the theory of computational complexity as it was the first computational task shown to be NP-complete, by Stephen Cook in 1971 as part of a paper that defined the notion of NP-completeness[Coo71]. Subsequent problems have been shown to belong to the same family by proving they are at least as hard as SAT. Roughly, a task is NPcomplete if a good algorithm for it would entail a good algorithm for SAT. Thus, despite its appealing simplicity, one can think of SAT as the "core" problem in this family of hard problems. An overview of the theory of NP-completeness is given in chapter 2.

SAT Variants and Relaxations

Every NP-complete problem has special cases, often referred to *subproblems*, that are solvable in polynomial time. Besides their theoretical importance, subproblems have practical use for relaxation purposes. The idea is to transform a SAT instance into another instance that is a special case and is solvable in polynomial time. Then solving it can indicate whether the original instance is satisfiable. The two widely used methods for SAT relaxations are deleting clauses and deleting literals until the resulting formula is a special case. SAT variants and relaxations are discussed in chapter 3.

Motivation

NP-Completeness does not exclude the possibility of finding algorithms that are efficient enough for solving many interesting SAT instances which arise from many diverse areas in AI. SAT is of special concern to AI because of its direct connection to reasoning and theorem-proving. Deductive reasoning is simply the complement of satisfiability: given a collection of base facts Σ, then a sentence α should be deduced iff $\Sigma \cup \{\neg\alpha\}$ is not satisfiable. Researches first became interested in SAT precisely for this reason. This is substantiated by the facts that Cook's paper in which he showed that SAT is NP-complete was named "The complexity of theorem-proving procedures", and the most prominent SAT algorithm on which most effective algorithms are based on, was published in a paper titled "A machine program for theorem-proving" [DLL62]. Recently there has been an explosion of interest in SAT because more and more practical problems in AI that utilize other forms of reasoning such as *constraint-based* reasoning with applications in *planning*, *configuration*, *diagnosis*, *resource allocation*, *scheduling*, and *electronic design automation (EDA)*, make direct appeal to satisfiability. Traditionally these problems were represented as *constraint satisfaction problems* (CSP's). Informally, a CSP is defined by a set of variables, a domain of values for each variable, and a set of constraints on subsets of variables. A solution to a CSP is a consistent assignment of all variables to values in such a way that all the constraints are satisfied. A CSP can have zero solutions (insoluble), one or many solutions. SAT problems are a special case of CSP's, in which each variable can take only two values. SAT and CSP are competing representations, since most CSP's in particular those mentioned above can be represented as SAT problems. However, the encoding of CSP's as SAT problems results in formulas of very large size that early SAT solvers were not able to cope with. Recent advances in encoding schemes and the discovery of new SAT algorithms, has shown this approach to be competitive or even outperform some state-of-the-art CSP algorithms.

SAT Algorithms and Classification

The first SAT solver is traditionally attributed to Davis and Putnam, and referred to as the Davis-Putnam procedure, or simply DP [DP60]. DP is based on resolution and has the drawback of potentially generating exponentially many clauses, making it infeasible for large formulas due to memory limitations. Two years after the introduction of DP, Davis, Logemann and Loveland introduced an optimization to DP commonly referred to as DPLL [DLL62], which is the "barebone" of most current state-of-the-art SAT solvers. Since then a wealth of algorithms were developed and several approaches have been proposed, in-

cluding variations of backtrack search, local search, continuous formulations and algebraic manipulation. Each year the *International Conference on Theory and Applications of Satisfiability Testing* hosts a SAT competition that highlights the "world's fastest" SAT solvers. Of these, backtracking search is the most popular approach, and has proven to be the most effective.

SAT algorithms can generally be organized into two groups; *complete* and *incomplete*. In search context they are referred to as *systematic* and *non-systematic* respectively. A complete algorithm can always determine whether an input has a solution or does not have one, most of them also give the actual variable setting for the solution or can easily be modified to do so. Incomplete algorithms on the other hand are not guaranteed to find a solution if one exists, and cannot cannot report or prove that no solution exists if they do not find one. Thus, incomplete methods are inappropriate for applications in which the goal is to prove unsatisfiability such as with theorem-provers. Nonetheless incomplete algorithms make up for their incompleteness by outperforming complete methods dramatically on satisfiable instances.

Most of the complete methods are based on the paradigm of eliminating variables one at a time recursively until one or more primitive formulas have been generated and solved to determine satisfiability. This in turn is usually done by either making repeated use of resolution, as done in DP, or by a backtracking algorithm that assigns each possible truth value to each variable in the formula and generates a sub-formula for each value, as done in DPLL.

DPLL or backtracking based testers can be generally classified into two main categories [LA97a, Le 01], based on the two different approaches they utilize. In the first group we find algorithms such as C-SAT [DABC93], TABLEAU [CA93], POSIT [Fre95], and SATz [LA97a] utilizing *chronological* backtracking together with *look-ahead* techniques. These types of algorithms are commonly referred to as look-ahead algorithms, and were found better suited for solving randomly generated SAT problems. In the second group we find algorithms such as SATO [Zha97], rel-SAT [BS97], GRASP [MSS96], and Chaff [MMZ$^+$01] that employ *non-chronological* backtracking also known as *look-back* techniques and *intelligent backtracking*. These types of algorithms are commonly referred to as look-back algorithms, and were found more effective for solving real world problems. The main difference between the two techniques, is that lookahead techniques exploit information about the remaining search space, and to do that they must heavily rely on "clever" variable ordering heuristics. On the other hand, lookback techniques exploit information about search which has already taken place, and for that they must rely on efficient *clause learning* schemes, also referred to as *constraint recording*.

Variable Ordering Heuristics

Variable ordering heuristics are rules we believe will generate solutions without exhaustive search. These rules are often called branching rules. A key factor contributing to the overall performance of any SAT algorithm and in particular look-ahead algorithms consists of the application of "clever" branching rules. A clever branching rule in the case of a satisfiable instance is a selection of variables and corresponding assignments in a sequel such that the amount of traversed search space is as small as possible. In the case of an unsatisfiable instance the variables and assignments selected by the branching rule should lead as early as possible to contradictions, thus minimizing search cost. Branching heuristics can be classified according to their underlying "branching hypothesis" which is used to explain or motivate the branching rule. However, a lack of clear statistical evidence supporting one strategy over others has made it difficult to determine what makes a good decision strategy and what makes a bad one. Morweover, the problem of choosing an optimal sequence of variables and their assignments has been proven to be NP-hard as well as coNP-hard[Lib00]. Making good branching decisions is not the only consideration, the other aspect is the effort spent on acquiring the knowledge used to make this decision. A good branching rule is also one that is fairly simple to compute. The simplest possible strategy is to simply select the next decision randomly from among the unassigned variables, an approach commonly denoted as RAND. At the other extreme, one can employ a heuristic involving the maximization of some moderately complex function of the current variable state such as the *Maximum Occurrences in Clauses of Minimum size* (MOMs) [Fre95] heuristic, and the *Jeroslow-Wang* (JW-OS, JW-TS) [HV95] heuristics that try to estimate the likelihood of Satisfiability. In between there is the *dynamic largest individual sum* (DLIS) heuristic (also called MAXO) [MSS96], which selects the literal that appears most frequently in unresolved clauses, and the relatively new *Variable State Independent Decaying Sum* (VSIDS) heuristic [MMZ$^+$01], which is sate independent and thus faster to compute.

Look-back Techniques

Backtracking algorithms often explore regions of the search space that are clearly devoid of solutions, or rediscover the same contradictions in other regions of the search space, a behavior which is usually referred to as *thrashing*. The reason useless regions are explored is because of *futile* backtracking which occurs when the algorithm reaches a dead end in the search space and backtracks to a choice that was not in any way responsible for the failure. Non-chronological/intelligent backtracking, an idea borrowed from CSP procedures

and introduced in [BS97] for SAT purposes, aim at rectifying this problem by ensuring that the algorithm will backtrack to a variable that is somehow responsible for at least one of the derived contradictions. This is usually achieved by utilizing clause leaning schemes, where whenever the algorithm hits a dead end, a new clause is generated containing the variables responsible for the contradiction, and is added it to the original formula. This new clause is then used to back up the search tree to one of the causes of failure, and also to prevent the algorithm from rediscovering the same contradictions in other regions of the search space. Standard clause learning schemes however, suffer from a few critical drawbacks due to an unrestricted amount of learning. The addition of new clauses to the clause database (formula) without any control, can potentially cause an exponential growth (in the number of variables) of the formula, leading us back to the problem of the DP procedure. Another problem is that that the larger (more literals) the learned clauses are, the less useful they are for pruning purposes. Solutions to these problems such as *relevance-based learning* [BS97] and *k-bounded learning* [MSS96], are based on a selective choice of clauses that are to be added to the formula.

Stochastic Local Search Techniques

Systematic SAT algorithms traverse a search space systematically to ensure that no part of it goes unexplored. Alternative to systematic algorithms for SAT are the non-systematic local search algorithms which explore a search space randomly by making local perturbations to a working variable assignment without memory of where they have been. This feature causes them to be incomplete. Local search SAT procedures can be characterized by a cost function defined over truth assignments such that the global minima corresponds to a satisfying truth assignment. A typical local search procedure will start with a random truth assignment as a potential solution, and then try to improve on it incrementally by searching for a lower cost assignment within its local neighborhood. This is typically done by "flipping" the truth value of one of the variables according to some randomized greedy heuristic strategy with the aim of minimizing the number of unsatisfied clauses by the flip. Flips are repeated until either a satisfying assignment is found or a bound on the maximum number of flips is reached. In the this case the process is restarted and repeated up to a maximum number of tries. Unfortunately, local search procedures suffer from a common pathology. The possibility of getting stuck on local minima. A local minima is a valley in the state space landscape that is lower than each of its neighboring states but higher than the global minima. There are two standard ways to escape local minima. The first is by introducing random restarts [GSK98] and the second by introducing noise, usually both

are used together. Noise is a characteristic of a search strategy that causes it to make moves that are non-optimal in the sense that the moves increase or fail to decrease the objective function, even when such improving moves are available in the local neighborhood. Different local search procedures are realized by different noise strategies. GSAT [SLM92] was the first successful and the most widely studied SAT local search algorithm. It used a greedy strategy to choose next variable to flip, but did not employ an effective noise strategy. Simulated annealing a general technique that originates from the theory of statistical mechanics was one of the first effective mechanisms used to introduce noise into the search [KGV83]. However it was not until the introduction WalkSAT [SKC94], a greedy version of random walk, that these methods gained popularity, and demonstrated their inherent superiority over complete methods for satisfiable instances.

SAT algorithms and algorithmic approaches are reviewed in chapter 4. However, given the depth of the literature on this topic, and the vast amount of algorithms and approaches that have been proposed, only the classical and most prominent are mentioned and studied.

Performance Evaluation

Several methods to analyze and compare the performance of SAT algorithms are proposed in the literature. However, the very existence of all the different algorithms and approaches seems to suggest that there is no general useful theoretical way to evaluate or prefer one algorithm over the other. For the lack of satisfactory theoretical methods, algorithms are usually evaluated experimentally.

The analytical evaluation of SAT algorithms is primarily based on worst-case analysis, which is by its nature a very pessimistic analysis, that does often not reflect or explain the behavior of the algorithms in practice. Other forms of analysis are based on probabilistic measures which require a-priori knowledge of the input models, that are often unattainable. The two most widely used probabilistic measures of performance are *average-time/case* complexity and *probabilistic-time* complexity. Experimental studies, on the other hand, due to the space of likely formulas are forced to consider a relatively small number of input models, and are therefore inconclusive.

The two main classes of input models for evaluating the performance of SAT procedures are the *practical* problems, which are encodings of real world problems, and the *randomly generated* problems, which can be very useful benchmark problems, but have no known practical application. The incentive for using the random input model is that sufficiently hard instances are easy

to generate, and that the a-priori knowledge required for analytical studies is available. The advocates of random input models also argue that an efficient algorithm for hard random input models it most likely to perform on average at least as good as any specific algorithm tailored to solve a particular real-world problem, since these represent the "core" of hard problems [CA96]. Nonetheless some experiments strongly suggest that there is little correlation between the performance of a SAT procedure tested on random input models and the performance of the same algorithm tested on practical problems [GPFW97].

Several random input models have been proposed, prominent among them are the fixed-clause-length models (k-SAT). Instances of this model are generated by selecting uniformly a certain amount of clauses from the set of all possible clauses of a fixed length. By inspecting the relationship between ratios of clauses to variables, and the time required by DPLL to solve a random k-SAT instance, Mitchell et al. [MSL92] showed that random k-SAT instances exhibit a certain easy-hard-easy and satisfiable-to-unsatisfiable pattern, now called the *phase transition phenomena*. Their intriguing discovery was that the peak in difficulty occurs near the ratio where about 50% of the formulas are satisfiable. Moreover, the 50% point, known as the *crossover point*, seemed to occur at a fixed ratio of clauses to variables approximately equal to 4.3. Similar patterns of hardness were discovered by others, using substantially different algorithms, and for different clause sizes [ML96], which lead to the conjecture that this pattern will hold for all reasonable complete methods, and thus is a suitable model for evaluating SAT procedures. In recent years there has been a growing interest in finding the location of the crossover point, and the boundaries of the transition region. The range of ratios over which the phase transition has been observed, becomes smaller as the number of variables is increased. This coupled with the occurrence of transition phenomena in other random combinatorial problems lead to the *threshold conjecture*, which has been proven to be true for random 2-SAT when the ratio equals 1. For random 3-SAT, the provable bounds on the location of this threshold are 3.003 and 4.598 [CM97]. Performance evaluation of SAT procedures is discussed in chapter 5.

Chapter 2

An overview of the Theory of NP-completeness

Computational complexity is a branch of theoretical computer science that is concerned with classifying problems according to the computational resources, specifically time and memory, required to solve them. Since time requirements are often a dominant factor in determining whether a particular algorithm is efficient or not, most results in computational complexity focus on time complexity. This measure, in turn, depends on the algorithm's implementation as well as the computer on which the program is running. The theory of computational complexity and in particular the theory of *NP-completeness* provides us with a notion of complexity that is independent of implementation details and computer at hand.

2.1 Time Complexity

The time requirements of an algorithm are expressed in terms of the "size" of a problem's instance upon which the algorithm is applied. The size reflects the amount of data needed to describe the problem instance. After all, we would expect that the relative difficulty of solving a problem will increase with its input size, e.g. it takes more time to sort 100000 numbers rather than 10 numbers. Nonetheless, until we fix the encoding scheme, a procedure used to map problem instances into strings describing them, and the computer or model being used for determining execution time, the notion of "size" will remain ambiguous. However, we will see that these particular nuances will have no effect on the distinctions made in the theory of NP-completeness.

Since some algorithms perform better on particular instances of the same

problem, e.g. sorting an array that is already almost sorted, we consider the *worst-case* time complexity to minimize the dependency on the specific instance.

Definition 1 (Time complexity)**.** *Time complexity is a function* $T : \mathbb{N} \longrightarrow \mathbb{N}$*, where* $T(n)$ *is the* ***maximum*** *number (worse case) of time units that an algorithm requires in order to solve a problem instance of input size-*n*.*

This definition introduces another term that needs clarification-*time units*. A measure of time complexity should be based on a unit of time that is independent of the specific CPU's clock rate. Such time units are the number of elementary operations an algorithm executes, where elementary operations are considered to be simple operations such as adding or comparing two integers. Thus, measuring time units amounts to counting the number of elementary operations. This number in turn strongly depends on the algorithm's implementation details. Therefore rather than measuring the exact number $T(n)$ of elementary operations, we consider the asymptotic behavior of $T(n)$ as n gets very large. We say that $T(n)$ is of the order of $g(n)$ or that $g(n)$ is an **upper bound** for $T(n)$ and write $T(n) = \mathcal{O}(g(n))$ if there exists positive constants c and n_0 such that for all $n \geq n_0$, $T(n) \leq cg(n)$. Intuitively , $T(n) = \mathcal{O}(g(n))$ means that T is less than or equal to g if we discard differences up to a constant factor, i.e. $\mathcal{O}$ represents a suppressed constant.

2.2 Tractability and Polynomial Time Complexity

One way of discriminating problems is by classifying them as *tractable* or *intractable*. In computational complexity problems that are solvable by a **polynomial algorithm**, an algorithm that has time complexity $\mathcal{O}(n^k)$ for some constant k are called tractable, i.e. an algorithm that requires time bounded by a polynomial in the length of the input n. Problems with non-polynomial time complexity, such as $\mathcal{O}(k^n)$ i.e. exponential algorithms or $\mathcal{O}(n!)$, are referred to as intractable problems. In essence the notion of tractability is used to separate problems that in practice have feasible solutions from those that don't.

This system of classification may be challenged on several grounds. One can argue that based on this system a problem with time complexity $\mathcal{O}(n^{100})$ should be classified as tractable whereas it is much more reasonable to regrad it as intractable, $n = 10$ will already result a number that is comparable with the number of molecules in the universe or the number of nanoseconds since the "big bang". A similar argument can be raised for problems with time complexity $\mathcal{O}(n^{loglog\ n})$. A further argument is that this system classifies problems

based on a worst-case measure, such that a problem may have exponential time complexity but most of its instances require far less time, thus an average-case analysis (average time it takes to solve a problem over all possible instances) seems to be more appropriate and a better predictor of the practical utility of an algorithm.

Despite these reservations this method of classification has many appealing practical and theoretical advantages. Although it is reasonable to regard a problem that has time complexity $\mathcal{O}(n^{100})$ as intractable, or $\mathcal{O}(n^{loglog\ n})$ as tractable, there are far and few practical problems that have this time complexity, most practical problems tend to have polynomial time complexity with polynomial coefficients equal to 2 or 3. In support of the worst-case criterion it is argued that no methods are known for predicting in advance if an exponential time algorithm will run faster in practice, i.e. predicting the portion of instances that will require more time to be solved. Another contention is that utilizing average-case analysis requires us to determine which distributions of inputs to use for our analysis, a question with no absolute answer.

Among the advantages is what is commonly referred to as the *robustness* of P. The complexity class P has not been defined yet however the underlying idea is that a polynomial time complexity measure will be invariant under the change of model of computation we use. That is, if a problem is solved in polynomial time in one model it can be solved in polynomial time in another model, under the restriction that we are using a "reasonable" model, such as Turing machines, multi-tape Turing machines, random access machines (RAMs). Reasonable model in this context means that there is a polynomial bound on the amount of work that can be done in single unit of time.

Another advantage is that problems with polynomial time complexity are also invariant under different encoding schemes. Suppose an instance of a problem requires us to encode a graph, such an instance might be encoded by simply listing all vertices and edges, by an adjacency matrix or by an adjacency list, each producing different input length. However different encodings of the same problem producing different input length, will differ from one another by at most a polynomial amount. This again is true under the restriction that we are using a "reasonable" encoding scheme. In this context "reasonable" means that the encoding should be concise and not "padded" with unnecessary information (allowing us to convert a exponential time algorithm to a polynomial time one), and that we are using any base other than the expensive unary base to encode numbers. The reason we don't use unary is that unary encodings differ by an exponential factor from other base encodings, resulting unrealistically good bounds. Suppose for example we want to determine whether a number n is a prime number or not, a naive solution is to go through all numbers between 2 to $\sqrt{n}$ and check if any of them are factors of n. If the problem is encoded

in unary we will have to make $\mathcal{O}(\sqrt{n})$ comparisons since n is the input size, however if the problem is encoded in binary the input size is $k = \lceil log\ n \rceil$ but we still have to make $\mathcal{O}(\sqrt{n})$ comparisons, therefore in terms of the input size k we will have to make $\mathcal{O}(2^{k/2})$ comparisons,thus turning the problem into an intractable one.

In summary we see that polynomial time complexity is a useful measure in discriminating problems as tractable/intractable while maintaining a notion that is independent of implementation details and computer at hand.

2.3 Decision Problems

Recall that an ordered pair is a set of a pair of objects with an order associated with them, and a binary relation R from a set A to a set B is a subset of all possible ordered pairs $< a, b >$ such that $a \in A$, $b \in B$, i.e $R \subseteq A \times B$. A problem Π may be depicted abstractly as a binary relation R_Π on the set of problem instances I and the set of problem solutions S (assuming each instance has a solution). Solving problem Π corresponds to searching over the binary relation R_Π, where the input is some $i \in I$ and the task is to find some $s \in S$ such that $< i, s > \in R_\Pi$. Consider the *traveling salesman problem* (TSP), where a traveling salesman has a number of cities to visit, called a tour, where every city is visited only once except that he returns to the city from which he starts. The goal is to find a tour that minimizes the total distance the salesman has to travel among all possible tours. The set of problem instances for TSP are pairs each consisting of a set of cites to be visited and a distance matrix, the set of solutions are sequences of cities visited. The problem of finding shortest tour can be viewed as the relation that associates each pair (instance) with a sequence (solution), and formally may be written as $R_{TSP} = \{< i, s > | i \in I, s \in S,$ s is a shortest tour for i$\}$.

Problems generally come in three different flavors: the *search* problem, that is finding a feasible solution (most of which are optimization problems such as TSP, the *decision* problem, that is determining whether a feasible solution exists, and the *verification* problem, i.e. deciding whether a given solution is correct. Much of complexity theory however, and the theory of *NP-completeness* in particular deals with decision problems, which are problems whose solutions are either *yes* or *no*, i.e. the solution set is $\{1, 0\}$. In this case a decision problem can be viewed as a function mapping instances into the set $\{1, 0\}$. The reason decision problems are preferred is due to their simplicity and there very natural formal counterpart- a *language*. By investigating simpler problems whose solutions are recognizable without comparisons to all feasible solutions like with optimization problems such as TSP, we hope to learn more

about the barrier that separates tractable from intractable problems. Although most problems are not decision problems but rather optimization problems, they can be recast into decision versions by imposing a bound on the value to be optimized. As an example the problem TSP can be recast into TSP(D): given a graph, cost matrix and a bound B, does the graph contain a TSP tour of cost less than B. Although this process yields a different problem than the original optimization problem, we can still gain insight to the tractability of the optimization problem. If the optimization problem can be solved in polynomial time so can the decision version of it, simply by comparing the solution obtained with the bound. Taking the contrapositive which is more relevant to the theory of *NP-completeness*, if we can provide evidence that the decision version is hard to solve or intractable then we have also shown that its related optimization problem is also hard to solve. A question that naturally arises is whether the converse is true, i.e. does an efficient solution for a decision problem guarantee an efficient one for its optimization counterpart. The answer is not known to be true in general , but can be shown to be true for all *NP-complete* problems, via self reducible relations (beyond the scope of this survey).

2.4 Formal Languages, and Turing Machines

Computational complexity can be discussed informally in terms problems and algorithms for solving them, and formally in terms of languages and Turing machines, where the languages correspond to problems, and the Turing machines to algorithms. As mentioned earlier the main reason for focusing on decision problems is that they have a very natural, formal counterpart, a formal-language. By using tools from formal-languages we will be able to express the relation between a decision problem and the algorithm that solves it in a mathematically rigorous way. Lets first review some of the basic definitions of formal language theory.

An *alphabet* Σ is a finite set of symbols, e.g. $\Sigma = \{0, 1\}$. We denote by Σ^* the set of all strings of symbols (words) from Σ, e.g. $\Sigma^* = \{\epsilon, 0, 1, 00, 01, 10, 11, 000, ...\}$ where ϵ is the empty string. A *language* L over Σ is a subset of Σ^* and the *empty language* is denoted by $\emptyset$.

The correspondence between languages and decision problems is the following: an encoding scheme (assuming a "reasonable" one as discussed earlier) of a problem Π may be thought of as a way of describing each instance I of Π by a string w of symbols of some fixed alphabet Σ_Π. This in turn induces a partition of Σ_Π^* into three classes of strings; those that encode instances of Π for which the answer is "yes" Y_Π, those for which the answer is "no" N_Π, and

those that are not encodings of instances of Π. The language L_Π associated with decision problem Π is defined as Y_Π, i.e. $L_\Pi = Y_\Pi$, and the problem of determining whether the solution to an instance I of Π is "yes" or "no"', can be cast into the formal-language *membership* problem of determining the membership of w in L_Π, i.e does $w \in L_\Pi$?

This correspondence also allows us to define complexity classes as classes of languages instead of problems, where, informally each class is classified by the amount of computational resources needed to solve the membership problem of each of its members (languages).

In order to define the notion of a complexity measure precisely we also need to formalize the notion of an algorithm, initially by fixing a particular model of computation. The standard model in computability theory is the Turing machine, introduced by Alan Turing in 1936. A *deterministic one-tape Turing machine* (DTM) consists of a finite state control (i.e. a finite program) attached to read/write head moving on an infinite tape. The tape is divided into squares, each capable of storing one symbol from a finite alphabet Γ which includes the blank symbol b. Each DTM has a specified input alphabet Σ, which is a subset of Γ, not including the blank symbol 'b'. At each step in a computation the DTM is in some state q in a specified finite set Q of possible states. Initially a finite input string over Σ is written on adjacent squares of the tape, all other squares are blank (contain 'b'), the head scans the leftmost symbol of the input string, and the DTM is in the initial state q_0.

Formally a DTM is a tuple $\langle \Sigma, \Gamma, Q, \delta \rangle$ where Σ, Γ, Q are finite nonempty sets such that $\Sigma \subseteq \Gamma$, $b \in \Gamma - \Sigma$, and the state set Q contains three special states q_0, q_{accept}, q_{reject}. The transition function δ is

$$\delta : (Q - \{q_{accept}, q_{reject}\}) \times \Gamma \rightarrow Q \times \Gamma \times \{-1, 1\}$$

where $\delta(q, s) = (q', s', h)$ is to be interpreted as when the DTM is in state q scanning the symbol s then q' will be the new state, s' is the symbol printed, and the tape head moves right or left one square depending on whether h is 1 or -1.

At each step of the computation the DTM is in some state q and the head is scanning a tape square containing some tape symbol s, and the action performed depends on the pair (q, s) and is specified by the $DTM's$ transition function (or program) δ. The action consists of printing a symbol on the scanned square, moving the head left or right one square, and assuming a new state. If during the computation one of the states q_{accept} or q_{reject} are reached then the computation halts with either "yes" or "no" respectively. We say that a DTM program M with input alphabet Σ ***accepts*** $w \in \Sigma^*$ iff M halts in

state q_{accept}. We denote the language ***recognized*** (or accepted) by M as

$$L_M = \{w \in \Sigma^* | M \ accepts \ w\}$$

Note that if $w \in \Sigma^* - L_M$ then the computation may halt in state q_{reject} or never halt, i.e. loop forever. However, for a DTM program M to correspond to our notion of an algorithm, it must halt on all possible strings of its input alphabet. We say that a DTM program M ***decides*** $w \in \Sigma^*$ iff it either accepts or rejects (halts in state q_{reject}) w.

We can now state the correspondence between "recognizing" languages and solving decision problems. We say that a DTM program M solves the decision problem Π if M decides its input and $L_M = L_\Pi$.

2.5 Time Complexity Classes

2.5.1 Complexity Class $\mathcal{P}$

We are now ready for a formal definition of *time complexity* and the two most important time complexity classes, $\mathcal{P}$ and $\mathcal{NP}$. The time used in the computation of a DTM program M on input w, denoted by $t_M(w)$, is the number of steps occurring in that computation until a halt state is reached. The time complexity of M is a function $T_M : \mathbb{N} \to \mathbb{N}$ given by

$$T_M(|w|) = max\{t_m(w) | w \in \Sigma^{|w|}\}$$

where $|w|$ is the length of string w. A program M is called a **polynomial time** DTM program if there exists a constant k such that for any $n \in \mathbb{N}$, $T_M(n) = \mathcal{O}(n^k)$.

Definition 2 (Complexity Class $\mathcal{P}$)**.**

$\mathcal{P} = \{L | \ there \ exists \ a \ polynomial \ time \ DTM \ program \ M \ for \ which \ L = L_M\}$

We say that a decision problem Π belongs to $\mathcal{P}$, that is solvable in polynomial time, if $L_\Pi \in \mathcal{P}$.

2.5.2 Complexity Class $\mathcal{NP}$

Originally the class $\mathcal{NP}$ was defined in terms of polynomial time nondeterministic Turing machines ($NDTM$), which are similar to $DTMs$ except that the transition function δ now takes the form

$$\delta : (Q - \{q_{accept}, q_{reject}\}) \times \Gamma \to \Delta \subseteq (Q \times \Gamma \times \{-1, 1\})$$

that is, a machine that at any point in the computation has the choice to proceed according to more than one unique possibility for a given state and symbol scanned. One way of viewing the computation of a $NDTM$ is as a tree whose branches correspond to parallel processes, one for each of the possible elements of Δ, thus a nondeterministic computation can potentially perform an exponential number of computations in polynomial time. Another common view is that the nondeterministic computation consists of two separate stages, guessing and checking. Given an instance I the first stage merely "guesses" a solution, which together with I is then fed into the checking stage that checks in a normal deterministic manner whether or not the guessed solution is a solution to the problem. We say that a nondeterministic algorithm solves a decision problem Π if the following properties for all $I \in \Pi$ hold:

- if $I \in Y_{\Pi}$, then there exists at least one guess that will lead the checking stage to respond "yes", or equivalently at least one branch of the computation tree that leads to a "yes" answer.
- if $I \notin Y_{\Pi}$, then there is no guess that will cause the checking stage to respond "yes", or equivalently all branches of the computation tree lead to a "no" answer.

We say that nondeterministic algorithm solves decision problem Π in polynomial time if the checking stage returns an answer in time bounded by a polynomial, which in turn imposes a polynomial bound on the length of the guess, or equivalently the length of any branch in the computation tree is bounded by a polynomial. The class $\mathcal{NP}$ can now be defined informally as the class of all decision problems that can be solved by polynomial time nondeterministic algorithms.

Both these Traditional views come to show that the power of nondeterminism lies in the idea that it allows the exhaustive enumeration of an exponentially large number of candidate solutions in polynomial time, and if the evaluation of each candidate solution can be done in polynomial time then the total time for solving a problem is polynomial.

A more recent approach better suited for proving membership in $\mathcal{NP}$, and better captures the conceptual contents of the class, is the notion of polynomial time "verifiability" that the class $\mathcal{NP}$ intends to isolate. Informally, we can view the class $\mathcal{NP}$ as the class of languages that admit short *certificates* for membership in the language. Given this certificate, called *witness*, membership in the language can be verified in polynomial time. This certificate can be thought of as the string corresponding to the guess made by the $NDTM$ mentioned earlier, and as a result must be succinct, that is its length must be bounded by a polynomial, and checkable in polynomial time.

We formalize this notion using a *checking/verifying relation* which is simply a binary relation $R \subseteq \Sigma_x^* \times \Sigma_w^*$ for some finite alphabets Σ_x and Σ_w. The corresponding language associated with R and defined over $\Sigma_x \cup \Sigma_w \cup \{\#\}$ is

$$L_R = \{x\#w| < x, w > \in R\}$$

where # used to separate the input x from the witness w and therefore is not included in either Σ_x or Σ_w.

Definition 3 (Complexity Class $\mathcal{NP}$)**.** *The complexity class $\mathcal{NP}$ is the set of all languages L, each over some Σ_x, that satisfy the following conditions.*

1. *there exits a checking relation R for L where $L_R \in \mathcal{P}$*
2. $\forall\ x \in \Sigma_x^*\ , \exists k \in N\ (x \in L \Leftrightarrow \exists w(|w| \leq |x|^k\ and\ < x, w > \in R))$

The first condition states that the certificate must be checkable in polynomial time, while the second condition ensures that every "yes" instance of a problem must have a certificate, which furthermore is succinct, and for "no" instances there is no such certificate.

Using this definition of $\mathcal{NP}$ it can easily be shown that $\mathcal{P} \subseteq \mathcal{NP}$, by showing that for any language L defined over Σ, if $L \in \mathcal{P}$ then $L \in \mathcal{NP}$. Let the checking relation for L be $R = \{< x, y > |x \in L,\ y \in \Sigma^*,\ |y| \leq |x|^k\}$ (y is some arbitrary bounded string). $L \in \mathcal{P}$ entails that there exists a polynomial time DTM program M that recognizes L, using this M we can create a new program M' that simply ignores a portion of the input (the portion to the right of #), and proceeds to compute according to M. Thus we have showed that $L_R \in \mathcal{P}$, fulfilling the first requirement of the definition of $\mathcal{NP}$, where the second requirement holds by definition of R.

In this view of the complexity class $\mathcal{NP}$, the most famous and important question in computer science, $\mathcal{P} \stackrel{?}{=} \mathcal{NP}$ can be formulated as the question whether the existence of a succinct witness verifiable in polynomial time (as implied by membership in $\mathcal{NP}$) necessarily brings about an efficient algorithm for finding it (as required for membership in $\mathcal{P}$), i.e. $\mathcal{NP} \subseteq \mathcal{P}$?

2.6 Reducibility and $\mathcal{NP}$-*completeness*

A Reduction is a mathematical tool with which the relative complexity of problems are compared. The idea behind reductions is the transformation of one problem Π_1 to another Π_2 in such a way that if Π_2 is known to be easy, so is Π_1, and vice versa, if Π_1 is known to be hard so is Π_2. In practical terms,

the notion of a reduction form Π_1 to Π_2 is the ability to use an algorithm for solving Π_2 as a subroutine for solving Π_1. We will see that utilizing this idea leads to the result that there are some intractable problems that are complexity wise equivalent to each other. These problems are the so called $\mathcal{NP}$-*complete* problems, which seem to embody the secret of intractability in a way that an efficient algorithm for solving any one of them will immediately imply the tractability of all problems in $\mathcal{NP}$.

There several types of reductions commonly known as *Turing, Cook, Karp, Levin,* and *log-space* reductions. The two most significant to the theory of $\mathcal{NP}$-*completeness* are the *Cook, Karp* reductions.

Definition 4 (Oracle Turing Machine)**.** *An Oracle Turing Machine (OTM) denoted M^A is a Turing machine with an extra tape called the oracle tape for language A , and three extra states $q_?$, q_Y, q_N. The machine can write a string w on the oracle tape, and enter the query state $q_?$, then in one step transfer control to state q_Y or q_N depending whether $w \in A$ or $w \notin A$.*

Informally, an oracle for language A can be viewed as a magical device that can answer membership problems for language A (decide A) in a single time unit (single step).

Definition 5 (Cook Reduction)**.** *A Cook reduction from problem Π_1 to problem Π_2 denoted $\Pi_1 \leq_T^p \Pi_2$ is an OTM that in polynomial time solves problem Π_1 on input x while getting oracle answers for problem Π_2.*

Definition 6 (Karp Reduction)**.** *A Karp reduction (also called many to one reduction) of language $L_1 \subseteq \Sigma_1^*$ to language $L_2 \subseteq \Sigma_2^*$ denoted $L_1 \leq_m^p L_2$, is a polynomial time computable function $f : \Sigma_1^* \to \Sigma_2^*$ such that $x \in L_1 \Leftrightarrow f(x) \in L_2$.[Kar72]*

The $\leq$ symbol is used to emphasize that one problem is at least has hard as the other. The A Karp reduction is also called "many to one reduction" because the transformation function f is a many to one function, mapping potentially different instances of one problem to the same instance of another.

A Karp reduction can be viewed as a more realistic, constraint and special case of a Cook reduction where the oracle may be queried only once, as opposed to many times like with the Cook reduction. Formally this can be written as $\leq_m^p \subseteq \leq_T^p$, which seems to imply that a Cook reduction has more computing power than a Karp reduction. It is an open question however whether that is the case. Let *co*-$\mathcal{NP} = \{L | \overline{L} \in \mathcal{NP}\}$ denote the class of problem who's complement is in $\mathcal{NP}$, another open problem is $\mathcal{NP} \stackrel{?}{=}$ *co*-$\mathcal{NP}$, it conjectured that they are not (discussion beyond the scope of this survey), however it is easy to see that if $\leq_m^p = \leq_T^p$ then $\mathcal{NP} =$ *co*-$\mathcal{NP}$.

Lemma 1. *if $L_1 \leq_m^p L_2$ and $L_2 \in \mathcal{P}$ then $L_1 \in \mathcal{P}$.*

The proof is trivial and the idea is to use the DTM for deciding L_2 as a subroutine for deciding L_1. Since deciding L_2 takes polynomial time and the transformation of a string from L_1 to a string from L_2 also takes polynomial time then the total time is also polynomial, thus deciding L_1 takes polynomial time. This result comes to show, as stated earlier that an easy solution for one problem can imply an easy solution for another.

Definition 7 (*$\mathcal{NP}$-complete*)**.** *A language L is $\mathcal{NP}$-complete if and only if:*

1. *$L \in \mathcal{NP}$*
2. *for every language $L' \in \mathcal{NP}$, $L' \leq_m^p L$*

These languages are the hardest problems in $\mathcal{NP}$, in the sense that if we knew how to solve an *$\mathcal{NP}$-complete* problem efficiently we can efficiently solve any problem in $\mathcal{NP}$, which is merely an application of lemma 1. The following result shows the converse, informally the idea is that if one problem is known to be hard, and it reduces to another then so is the other problem hard. This result also shows how to prove that a problem is *$\mathcal{NP}$-complete*, by finding another problem known to be *$\mathcal{NP}$-complete* and a polynomial transformation of the known problem to the new problem.

Lemma 2. *if L_1 is $\mathcal{NP}$-complete, and $L_2 \in \mathcal{NP}$, and $L_1 \leq_m^p L_2$ then L_2 is $\mathcal{NP}$-complete.*

This result follows by the transitivity of Karp reductions.

2.7 Cook's Theorem

The theory of *$\mathcal{NP}$-completeness* is primarily used to compare the relative hardness of problems, where lemma 1 and lemma 2 provide us with the mechanism of establishing such relations. However without identifying the first *$\mathcal{NP}$-complete* problem the whole theory of *$\mathcal{NP}$-completeness* is somewhat sterilized. Such a problem was provided by Cook's theorem.

Definition 8. *Let $X = x_1, x_2, ..., x_n$ be a finite set of **Boolean variables**, and let $\bar{X} = \bar{x_1}, \bar{x_2}, ..., \bar{x_n}$ stand for the negations of $x_1, x_2, ..., x_n$. we call the elements of $X \cup \bar{X}$ **literals**. We call a set of literals a **clause** C, and a set of clauses a **formula** ϕ. We say that a formula is in **conjunctive normal form** (CNF), if $\phi = \bigwedge_{i=1}^{n} C_i$, where $n \geq 1$, and each C_i is the disjunction of one or more literals.*

Definition 9. *A truth assignment T for ϕ is a mapping $T : X \to \{1, 0\}$. A literal u is true under T ($T \models u$) iff $T(u) = 1$. A truth assignment T for X satisfies a clause $C \in \phi$, where ϕ is a Boolean formula in CNF, iff at least one literal $u \in C$ is true under T. We say that T satisfies ϕ iff it satisfies every clause in ϕ.*

Definition 10 (SAT problem)**.**

***Instance**: A Boolean formula ϕ in conjunctive normal form.*
***Question**: Does there exist a satisfying truth assignment for ϕ ?*

Theorem 1 (Cook's Theorem)**.** *$SAT \in \mathcal{NP}$-complete [Coo71]*

Proof (outline): We first show that $SAT \in \mathcal{NP}$. A nondeterministic algorithm for it will guess a truth assignment and check to see whether that assignment satisfies the formula which was the input to the algorithm. It is easy to see that this can be accomplished in polynomial time and that the length of the guess is less than the size of the input.

Showing that every $\mathcal{NP}$ language reduces to the SAT language cannot be achieved by presenting a reduction for each of them, since there are infinitely many of them. However each one of them has a NDTM program that recognizes it, thus we construct a generic reduction that takes the input string and produces a Boolean formula that simulates the NDTM program that recognizes the $\mathcal{NP}$ language with that input. If the machine accepts the formula produced will have a satisfying assignment that correspond to the accepting computation, and if the machine does not accept, no assignment will satisfy the formula produced. A tableau for NDTM program M on input w is an $n^k \times n^k$ table whose rows are the configurations of a branch of the computation of M on w. A tableau is accepting if any of its rows is an accepting configuration. The problem of determining whether M accepts w is thus equivalent to the problem of determining whether an accepting tableau for M on w exist. Therefore we generate a formula that simulates M on input w by generating a formula such that a satisfying assignment for it corresponds to an accepting tableau for M on w, and no satisfying assignment for it corresponds to the fact that M does not accept w. This formula in turn, is a conjunction of four smaller formulas that guarantee that certain conditions hold for the correspondence between an assignment and a tableau. The first guarantees that any satisfying assignment specifies one and only one symbol for each cell of the tableau. The second formula ensures that the first row of the tableau is a starting configuration. The third ensures that an accepting configuration occurs in the tableau. Finally, the forth formula guarantees that each row of the tableau corresponds to a configuration that legally follows form M's transition function, by constructing a formula for every 2×3 window of cells, and taking the conjunction of all such

windows. Showing that the construction of the formula took polynomial time follows from the fact the the length of the formula is bounded by a polynomial in the length of w.

2.8 $\mathcal{NP}$-*hard* problems

Although the definition of $\mathcal{NP}$-*complete* seems to be fairly unified , the definition of $\mathcal{NP}$-*hard* is somewhat less so. One of the definitions states that $\mathcal{NP}$-*hard* is the set of languages that satisfy only (but not necessarily property 1) of definition 7 ($\mathcal{NP}$-*complete*), i.e. $\mathcal{NP}$-*hard* refers to the class of decision problems that contains all problems L such that for all decision problems L' in NP there is a polynomial-time many-one reduction (Karp reduction) to L. Informally this class can be described as containing the decision problems that are at least as hard as any problem in NP. It is easy to see that according to this definition all $\mathcal{NP}$-*complete* problems are also $\mathcal{NP}$-*hard*, but there are not many problems that according to this definition are $\mathcal{NP}$-*hard* but not $\mathcal{NP}$-*complete*. One problem that has this property is the famous *halting problem*, given a program and its input, will it come to a halt or run forever? to show that it is $\mathcal{NP}$-*hard* we reduce SAT to it by transforming a SAT instance to the description of a Turing machine that tries all truth assignments for the SAT instance and when it finds one it halts, otherwise it goes into an infinite loop. It is also easy to see that the halting problem is not in NP since all problems in NP are decidable and the halting problem is not.

An alternative definition of $\mathcal{NP}$-*hard* which seems to have wider acceptance is one which extends the previous definition to also include search problems and not only decision problems, making it generally more useful. We say that a problem (search or decision) is $\mathcal{NP}$-*hard* if solving it in polynomial time would make it possible to solve all problems in class $\mathcal{NP}$ in polynomial time. Since the definition does not require that such a polynomial time solution exist, it implicitly replaces Karp reductions with Cook/Turings reduction where the oracle can compute any function, not only functions mapping to $\{0, 1\}$. Note however that under this definition all problems in *co*-$\mathcal{NP}$ are also $\mathcal{NP}$-*hard*.

Yet another definition, very similar to the previous says that a problem is $\mathcal{NP}$-*hard* if its decision version is known to belong to $\mathcal{NP}$-*complete*.

2.9 Historical Note

The existence of NP-complete problems was proved independently by Stephen Cook in the United States and Leonid Levin in the Soviet Union. Cook, then

a graduate student at Harvard, first identified the languages which we now call P and NP, and showed that several natural problems, including Satisfiability , and subgraph isomorphism are NP-complete. Meanwhile, Levin, a student of Kolmogorov at Moscow State University, proved that a variant of the tiling problem is NP-complete. Cook used in his proofs a different notion of reducibility, generating a similar class of problems, albeit believed to be bigger than the class presently known as NP-complete. He used polynomial Turing reductions which he termed "P-reductions", and originally meant to show that all polynomial time nondeterministic computations can be Turing reducible to the problem of determining if a propositional formula given in disjunctive normal form is tautology, where satisfiability result result was an intermediate result. It is also for this reason that Cook and Turing reduction are often considered the same thing. It was Richard Karp in 1972, in a tremendously influential paper, that the theory of NP-completeness took on its present form. Karp introduced the terms P and NP, and showed that Cook's theorem would hold if Cook reductions are replaced by the simpler and more realistic many to one reductions, which we call Karp reductions. He also showed that eight central combinatorial problems, including clique, independent set, set cover, and the traveling salesman problems are NP-complete. The terminology used today, such NP-complete, and NP-hard are primarily due to Donald Knuth's efforts of standardizing these terms.[FH03]

Chapter 3

Tractable and Intractable Variants of SAT

Any $\mathcal{NP}$-*complete* problem has special cases, often referred to *subproblems*. We can view these subproblems as lying on different sides of an imaginary boundary between tractability and intractability. One of the goals of analyzing NP-complete problems is to find the dividing line, the "frontier" between subproblems we know to be tractable and those that we know to be NP-complete. Thus the frontier can be viewed as a region consisting of subproblems whose NP-completeness is still open at the "current state of Knowledge". SAT provides such an example which is pursued in this section.

In the context of SAT solvers, subproblems are of interest to us for *relaxation* purposes. These relaxations are based on the existence of special cases of SAT that are solvable in polynomial time. The idea is to transform a SAT instance into another instance that is a special case and is solvable in polynomial time, then solving it can indicate whether the original instance is satisfiable. The two widely used methods for SAT relaxations are deleting clauses and deleting literals until the resulting formula is a special case. The first method is useful when we expect the formula to be unsatisfiable. We partition the original formula ϕ into to sub-formulas ϕ' and ϕ'' where ϕ' is solvable in polynomial time. Then we solve ϕ', if it is unsatisfiable then so is ϕ, else we can try to extend its satisfying assignment to ϕ. The second method is useful when we suspect the formula to be satisfiable. We partition each clause $C \in \phi$ into two sub-clauses C' (non empty) and C'' such that $\phi' = \bigwedge_i C'_i$ is solvable in polynomial time. We then solve ϕ', if it is satisfiable then so is ϕ and we are done.

The restriction of SAT to instances where each clause has exactly k where $k \geq 1$ literals is denoted k-SAT. Of special interest are 3-SAT and 2-SAT, since

3 is smallest value of k for which k-SAT is NP-complete [Coo71] (Cook actually proved the "dual" with 3-DNF-TAUT), while 2-SAT is solvable in linear time. The normalization to clauses of fixed size simplifies transformations used in proving NP-completeness results. In particular the restricted structure of 3-SAT, due to its "smallness", is used to prove many NP-completeness results.

Theorem 2. *3-$SAT \in \mathcal{NP}$-complete.*

Proof:Showing that 3-SAT belongs to NP is the same as showing that SAT belongs to NP. To show that 3-SAT is also NP-complete we transform SAT to it, i.e. we show $SAT \leq_m^p$ 3-SAT. To transform a SAT instance into a 3-SAT instance we examine each clause C in the SAT instance ϕ and do the following:

1. If the clause is already in the correct form then we do nothing, leave it as is.

2. If C has less than three literals then we construct a new clause C' of three literals by duplicating one or two of the literals. e.g.

$$C = (l_1 \vee l_2) \rightarrow C' = (l_1 \vee l_2 \vee l_1)$$

.

3. If C has more than three literals say k, then we introduce a new variable, and use it to split the clause into one with three literals and another with $k-1$ literals as follows:

$$C = (l_1 \vee l_2 \vee l_3 \vee ... \vee l_k) \rightarrow C' = (l_1 \vee l_2 \vee x_1) \wedge (\bar{x_1} \vee l_3 \vee ... \vee l_{k-1} \vee l_k)$$

We continue this procedure, constructing new clauses if necessary, until all clauses have exactly 3 literals. At this point the collection of all clauses generated by this procedure is the 3-SAT instance ϕ'. Now we have to show that any truth assignment T that satisfies ϕ can be extended to a truth assignment T' that satisfies ϕ', and vice versa. This is the same as showing that any truth assignment T that satisfies some C can be extended to a truth assignment T' that satisfies C' using the above procedure, and vice versa. This is trivially true when C consists of at most three literals (the first two cases). When C contains more than three literals we extend a truth assignment T that satisfies it as follows: since T satisfies C there must be a literal l_i where $1 \leq i \leq k$ which has been assigned the value true, if $i \leq 2$ then the new variable x is assigned false, else it is assigned true. It is easy to see that such an extension satisfies the clauses in C', and conversely, an assignment that satisfies C' when restricted to the original literals, satisfies C. To show that this transformation can be carried out in polynomial time it suffices to observe that the size of the

new formula is bounded linearly by the size of the original formula. For clauses of size one we add two literals, for size two we add one literal, and for size bigger than three the clause will be to be transformed into $k-2$ new clause with the addition of $2(k-3)$ new literals.

Theorem 3. *for all $k > 3$ k-$SAT \in \mathcal{NP}$-complete.*

Proof: using the fact that we know 3-SAT is NP-complete, it suffices to show a polynomial time reduction from k-SAT to $k+1$-SAT. This is trivial and can be done either by duplicating one of the literals or if clauses are depicted as sets where no duplicates are allowed, we transform each k-clause into two k+1-clauses as follows:

$$C = (l_1 \vee ... \vee l_k) \rightarrow C' = (l_1 \vee ... \vee l_k \vee x_1) \wedge (\bar{x_1} \vee l_1 \vee ... \vee l_k)$$

Theorem 4. 2-$SAT \in \mathcal{P}$.

Proof:There are alternative ways of showing that solving 2-SAT takes polynomial time. The first is by observing that the procedures suggested in [DP60, DLL62] will require no more than polynomial time to solve a 2-SAT instance, because the overall number of clauses/resolvents of size 2 is bounded by $O(n^2)$. However a Linear time algorithm for solving 2-SAT exists [APT79], by reducing a 2-SAT instance into an *implication graph*. A graph obtained by converting each clause into a direct edge, e.g. the clause $(\bar{A} \vee B) \equiv (A \rightarrow B) \equiv (\bar{B} \rightarrow \bar{A})$ will be converted to an edge from A to B and to an edge from $\bar{A}$ to $\bar{B}$, where $A, \bar{A}, B, \bar{B}$ are nodes in the graph. A path from a vertex x_i to a vertex $\bar{x_i}$ in the graph corresponds to a resolution derivation of $\bar{x_i}$ from the formula we started with[Sub]. Thus if we can find a path from x_i to $\bar{x_i}$ and a path from $\bar{x_i}$ to x_i then we have a derivation of both x_i and $\bar{x_i}$ which resolves to $\perp$, which in turn implies that the formula is unsatisfiable. the problem of finding a path between two vertices can be solved using BFS in $O(m)$ where m is the number of edges in the graph and clauses in the formula.

A clause (i.e. a disjunction of literals) is called a *Horn clause*, if it contains at most one positive literal. Such a clause can be written as an implication: $(x \vee \bar{y} \vee \bar{w} \vee \bar{z})$ is equivalent to $((y \wedge w \wedge z) \rightarrow x)$. HORNSAT is the problem of deciding whether a given Boolean expression that is a conjunction of Horn clauses is satisfiable.

Theorem 5. $HORNSAT \in \mathcal{P}$.

Proof:First we set all variables to false, if the formula does not contain clauses with exactly one positive literal then the formula is trivially (setting all variables to false) satisfied. We next go through the formula looking for false clauses and

fix them by setting their positive literal to true. We continue this process until a contradiction is reached (at purely negative clauses) or a satisfying truth assignment is found. This algorithm sets at most n variables, each time checking l literals. Thus, the algorithm requires $O(n \cdot l)$ steps. By using special data structures this algorithm can be improved to run in linear time.

The problem of determining whether a boolean formula in disjunctive normal form (DNF) is satisfiable is called DNF-SAT.

Theorem 6. *$DNF\text{-}SAT \in \mathcal{P}$.*

Proof: Since any DNF formula can be satisfied unless every clause contains both a literal and its complement Solving DNF-SAT is trivial, all we have to do is search for each clause for a literal and its negation which takes linear time .

Chapter 4

SAT Algorithms

Over the years several approaches have been proposed for solving SAT, including variations of backtrack search, local search, continuous formulations and algebraic manipulation. Of these, backtracking is the most popular method, and has proven to be the most effective for solving specific instances of SAT such as Electronic Design Automation (EDA), and for applications where the objective is to prove unsatisfiability, such as theorem provers. In [GPFW97] an attempt to organize all the different approaches for solving SAT into an *algorithm space* is presented. The algorithm space intends to unify a variety of search and optimization algorithms in terms of variable domain, constraint used and parallelism in the algorithms, which results a three dimensional algorithm space.

SAT algorithms can generally be classified as *complete* and an *incomplete* algorithms. In search context they are referred to as *systematic* and *non-systematic* respectively. A complete algorithm can always determine whether an input has a solution or does not have one, most of them also give the actual variable setting for the solution or can easily be modified to do so. Incomplete algorithms on the other hand do not always find a solution and cannot prove unsatisfiability. Most incomplete algorithms find a solution, but give up or do not terminate in other cases. In such cases one does not know whether an instance has no solution or the algorithm did not search hard enough. Nonetheless incomplete algorithms have been able to solve SAT instances for which complete algorithms were not able to.

4.1 Complete Algorithms

Most of the complete methods are based on the paradigm of eliminating variables one at a time recursively until one or more primitive formulas have been generated and solved to determine satisfiability. This in turn is usually done by either making repeated use of resolution, as was done in the original version of the Davis-Putnam (called DP) procedure [DP60], or by a backtracking algorithm that assigns each possible truth value to each variable in the formula and generates a sub-formula for each value, as was done by Davis, Logemann and Loveland's improvement to DP [DLL62]. This algorithm usually referred to as DPLL or DPL or DLL (we use DPLL) is the "barebone" of most current state-of-the-art SAT solvers. In this section we discuss these paradigms, initially by laying the logical framework used by them.

4.1.1 Logical Framework

Notation: To simplify the manipulation of CNF formulas, it is helpful to represent them as sets. In doing so we eliminate the order of clauses and order of literals in each clause. A clause $C = \bigvee_{j \leq m} l_j$ is represented as the set $C = \{l_j | j \leq m\}$, e.g. $(p \vee q \vee r)$ will be denoted as $\{pqr\}$. A formula $\phi = \bigwedge_{i \leq n} C_i$ is represented as the set $\phi = \{C_i | i \leq n\}$, e.g $(p \vee q) \wedge (\bar{p} \vee r)$ will be denoted as $\{\{pq\}, \{\bar{p}r\}\}$, or simply $\{pq, \bar{p}r\}$. We denote by $\Box$ the empty clause, and by $\emptyset$ the empty formula. A satisfiable formula is one which has an assignment that satisfies all clauses in the formula. Since the empty formula has no such clauses any assignment will satisfy it, thus the empty formula $\emptyset$ is satisfiable, moreover it is tautology. A clause is satisfiable if and only if there exists a true literal that belongs to it. Since $\Box$ is empty no such literal exists and therefore we regard the empty clause as unsatisfiable. This in turn implies that a formula ϕ such $\Box \in \phi$ is unsatisfiable.

Definition 11. *(Resolution rule) Let C_1, C_2 be clauses such that $l \in C_1$ and $\bar{l} \in C_2$. Then C the resolvent of C_1 and C_2 with respect to literal l is the clause:*

$$res_l(C_1, C_2) = (C_1 - \{l\}) \cup (C_2 - \{\bar{l}\})$$

Here are some examples:

$$Let\ C_1 = \{p\bar{q}\},\ C_2 = \{\bar{p}z\}\ then\ res_p(C_1, C_2) = \{\bar{q}z\}$$

$$Let\ C_1 = \{p\}, C_2 = \{\bar{p}\}\ then\ res_p(C_1, C_2) = \Box$$

$$Let\ C_1 = \{p\bar{q}\}, C_2 = \{p\bar{z}\}\ then\ res_p(C_1, C_2) = \emptyset$$

Note that $res_p(C_1, C_2) = \emptyset$ and $res_p(C_1, C_2) = \Box$ are not the same.

Theorem 7. *Let C_1, C_2 be clauses such that $l \in C_1$ and $\bar{l} \in C_2$, then $res_l(C_1, C_2)$ is satisfiable if and only if C_1 and C_2 are mutually satisfiable.*

here comes a proof

Definition 12. *let ϕ be a set of clauses. A sequence of clauses $C_1, C_2, ...C_n$ is called a **resolution derivation** of clause C_n from ϕ, denoted $\phi \vdash_{res} C_n$, if each C_i where $1 \leq i \leq n$, is either:*

1. *a member of ϕ, that is $C_i \in \phi$*
2. *there $j, k < i$ and literal l such that $C_i = res_l(C_j, C_k)$*

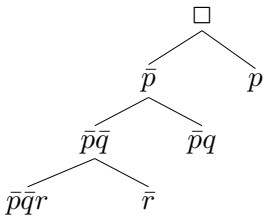
Definition 13. *A resolution derivation of $\square$ from ϕ, $\phi \vdash_{res} \square$, is called a **Resolution Refutation** of ϕ.*

Theorem 8. *(soundness of Resolution Refutation) If the unsatisfiable clause $\square$ is derived by resolution from the set ϕ of clauses, then ϕ is unsatisfiable.*

$$\phi \vdash_{res} \square \Rightarrow \phi \models \square$$

here comes a proof

Example, showing that $\{\bar{p}\bar{q}r, \bar{r}, \bar{p}q, p\}$ is unsatisfiable:



Theorem 9. *(completeness of Resolution Refutation) If a set of clauses ϕ is unsatisfiable then there is a resolution refutation of ϕ.*

$$\phi \models \square \Rightarrow \phi \vdash_{res} \square.$$

here comes a proof

4.1.2 The DP procedure

Theorem 8 suggests a way for proving that a formula is unsatisfiable, however it does not suffice for proving that a formula is satisfiable. The contrapositive

of theorem 9 says that if a resolution refutation for a formula ϕ does not exist then ϕ is satisfiable. Thus the combination of theorems 8 and 9 suggest a procedure for testing satisfiability. Choose a literal l, compute all possible resolvents of l and add them to the original set of clauses. If in this process a $\Box$ is derived then the original set of clauses is unsatisfiable, if not then choose another literal and repeat the same process. Since the number of literals is finite the process of choosing a different literal and generating all possible resolvents must terminate, and if a $\Box$ was not derived during this process then the original set of clauses must be satisfiable.

Resolution is attributed to Robinson (1965), however he did not invent it nor was he the first to apply it to automated deduction, and the proof is in the fact that it was used in the DP procedure (1960). Resolution refutation is the core of the DP procedure which is augmented by the following simplification optimization techniques, often referred to as the Davis-Putnam rules [DSW94].

1. **Unit rule**: If ϕ is a formula containing a one literal clause $\{l\}$ then ϕ is simplified and transformed into ϕ' by erasing all clauses that contain l and deleting all occurrences of $\bar{l}$ from the remaining clauses, e.g. $\{pq, p, r\bar{q}, r\bar{p}q\} \xrightarrow{p} \{r\bar{q}, rq\}$. ϕ is satisfiable if and only if ϕ' is satisfiable.

   here will come a proof

2. **Pure Literal Rule**: If a literal l occurs in ϕ only positively or only negatively then delete from ϕ all clauses which contain l, e.g. $\{pq, p, r\bar{q}, r\bar{p}q\} \xrightarrow{r} \{pq, p\}$. The resulting formula ϕ' is satisfiable if and only if ϕ.

   here will come a proof

3. **Elimination rule**: Let $res_l(\phi)$ be the set of all resolvents of ϕ with respect to literal l, let ϕ_l^0 be the set of all clauses not containing l or $\bar{l}$ (i.e. l-$free$ clauses), and let $\phi' = \phi_l^0 \cup res_l(\phi)$, e.g. $\{pq, p, r\bar{q}, r\bar{p}q\} \xrightarrow{p} \{r\bar{q}, rq\}$. Then ϕ is satisfiable if and only if ϕ' is satisfiable.

   here will come a proof

The efficiency of DP is highly correlated with the variable selection heuristic (*), in that "wiser" selections lead to faster solutions. Many heuristics were proposed for this problem, mainly aimed at the selection heuristic used in

```
procedure DP(CNF formula : φ)
    φ' := φ
    while loop forever do
        if there is a unit clause {l} in φ'
            then φ' := UnitRule(φ', l)
        else if there is a pure literal l in φ'
            then φ' := PureLiteralRule(φ', l)
        else
            v := select a variable mentioned in φ' according to some heuristic (*)
            φ' := EliminationRule(φ', v)
        end
        if □ ∈ φ' return unsatisfiable
        if φ' = ∅ return satisfiable
    end
end.
```

Figure 4.1: the Davis-Putnam procedure.

$\{w\bar{q}p, rp, p\bar{q}, \bar{p}s, q\bar{r}, q\bar{s}, t, tqp\}$	
$\{w\bar{q}p, rp, p\bar{q}, \bar{p}s, q\bar{r}, q\bar{s}\}$	unit t
$\{rp, p\bar{q}, \bar{p}s, q\bar{r}, q\bar{s}\}$	pure literal w
$\{q\bar{r}, q\bar{s}, \bar{q}s, rs\}$	elimination p
$\{rs, \bar{r}s, \bar{s}s\}$	elimination q
$\{s, \bar{s}s\}$	elimination r
$\emptyset$	unit s

Figure 4.2: Showing that a formula is satisfiable using DP

DPLL rather than DP. These heuristics can be as simple as choosing the first remaining variable, or may be quite sophisticated. In the DP procedure the selection rule proposed was to select the variable occurring in the first clause with minimal length, since this will potentially increase the ability to apply the unit rule, which in turn will create subproblems of smaller size much faster. A detailed discussion of variable selection heuristics is found in section 4.3.

4.1.3 The DPLL procedure

When the unit and pure literal rules cannot be applied, the DP procedure will, in each iteration, generate all possible resolvents with respect to some variable and delete all clauses mentioning that variable. This in turn reduces the problem to a subproblem with one less variable, but potentially quadratically more clauses in each iteration, and in total exponentially many clauses, making it infeasible for large formulas due to memory limitations.

Two years after the introduction of DP, Davis, Logemann and Loveland introduced an optimization to the DP procedure, by replacing the *elimination* rule by a new rule called the *splitting* rule.

4 **Splitting rule**: Let ϕ_l^+ be the set of all clauses in ϕ containing l, and ϕ_l^- be the set of all clauses in ϕ containing $\bar{l}$. Let

$$\phi'_{+l} = \phi_l^0 \cup \{C - \{l\} | C \in \phi_l^+\}$$
$$\phi'_{-l} = \phi_l^0 \cup \{C - \{\bar{l}\} | C \in \phi_l^-\}.$$

Then ϕ is satisfiable if and only if ϕ'_{+l} or ϕ'_{-l} is satisfiable.

here will come a proof

In this version a variable is selected and the problem is split into two smaller subproblems each of which assumes one of the two possible truth values of the selected variable. This procedure has the virtue of eliminating one literal at the expense of considering two formulas instead of one. The DPLL procedure is essentially a backtracking procedure augmented by simplification techniques, that searches implicitly in a depth-first manner the whole state space of the problem. It guarantees a solution, but on the other hand has the potential of requiring exponential time, due to the enumeration of all possible 2^n truth assignments in worse case. Nonetheless DPLL based implementations including "good" heuristics for choosing the splitting variable, and intelligent backtracking techniques perform quite well in practice, and are the best known and used SAT checking complete methods.

4.2 DPLL Based Backtracking Algorithms

Real world DPLL based testers can be generally classified into two main categories[LA97a, Le 01], based on the two different approaches they utilize. In the first group we find algorithms such as C-SAT, TABLEAU, POSIT, and SATz utilizing *chronological* backtracking together *look-ahead* techniques. This approach was found better suited for solving randomly generated SAT problems and is currently advocated by Chu Min Li et al. In the second group we find algorithms such as SATO, rel-SAT, GRASP, and Chaff that employ *non-chronological* backtracking also known as *look-back* techniques and *intelligent backtracking*, together with *clause learning* also known as *nogood*, *constraint recording*. This approach is more effective for solving real world problems, which are considered structured problems, and is currently advocated by João P. Marques et al.

```
procedure DPLL(CNF formula : φ)
    if □ ∈ φ return unsatisfiable
    else if φ = ∅ return satisfiable
    else if there is a pure literal l in φ then
        φ := PureLiteralRule(φ, l)
        return DPLL(φ)
    else if there is a unit clause {l} in φ then
        φ := UnitRule(φ, l)
        return DPLL(φ)
    else
        v := select a variable mentioned in φ according to some heuristic
        [φ'_{+l}, φ'_{−l}] := Split(φ, v)
        if DPLL(φ'_{+l}) =satisfiable then
            return satisfiable
        else
            return DPLL(φ'_{−l})
        end
    end.
```

Figure 4.3: The Davis, Logemann and Loveland procedure.

4.2.1 General Strategy and Tactics

DPLL based algorithms are based on splitting. During each iteration, the algorithm selects a variable according to some heuristic and generates two sub-formulas by assigning the two values, true and false, to the selected variable. A backtrack search algorithm for SAT is implemented by a search process that implicitly enumerates the space of 2^n possible binary assignments of the n problem variables. Starting from an empty truth assignment, a backtrack search algorithm enumerates the space of truth assignments implicitly and organizes the search for a satisfying assignment by maintaining a decision tree. For each node we have two branches, one for each possible assignment. We associate a branch of the tree with a sequence of variable assignments. The root of the tree corresponds to the initial formula. The leaf nodes correspond to sub-formulas that are either satisfiable or unsatisfiable. When searching for a solution, at each iteration/split the algorithm has to check whether any clause has become unsatisfiable, i.e. a □ has been generated, by the partial variable assignments made up to that point. In such cases a contradiction or conflict is said to be encountered and the algorithm must backtrack (hence *backtracking* algorithm) to an earlier stage and try a different assignment to one or more variables. If every node in the tree has been visited, which means backtrack is no more possible, the formula is declared to be unsatisfiable.

The two main types of SAT backtracking algorithms may be distinguished

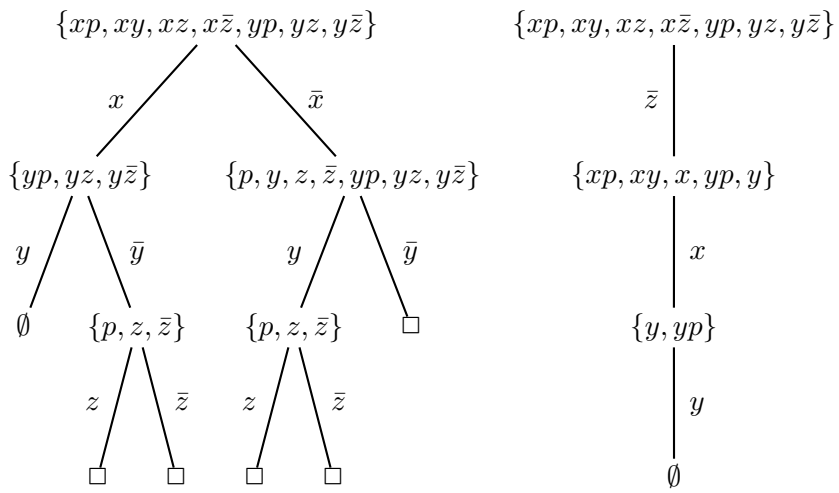


Figure 4.4: DPLL (using the splitting rule only) applied twice on same formula resulting two search trees, where the right tree corresponds to a better selection of splitting variables.

by the way the backtrack. Chronological backtracking algorithms backtrack to the most recent choice of a variable and its assignment, thus exploring one branch of the search tree completely before backtracking and exploring the other in a systematic chronological manner. Naive chronological backtracking algorithms often explore irrelevant branches, regions that are clearly devoid of solutions, or rediscover the same contradiction repeatedly. The general term for this sort of behavior is *thrashing*. One of the approaches to avoid this problem is to employ non-chronological backtracking methods. These methods enable the algorithm to backtrack more steps to any stage in the search process, and by that avoid the exploration of useless regions, which is why it is also referred to as "intelligent" backtracking.

In [LA97a] advocating chronological backtracking and look-ahead methods Chu Min LI et al. argue that the cause of exploring useless regions is a bad selection of splitting variables and that the purpose of intelligent backtracking is merely to correct these bad choices. However a better variable ordering heuristic, allows to avoid many useless backtrackings and explorations, thus eliminating the need for intelligent or non-chronological backtracking. João P. Marques et al. on the other hand argue [Lyn01, MS99, LaPMS02] that even good branching heuristics cannot eliminate mistakes, which non-chronological backtracking is able to correct. They further argue that experimental results obtained from a large number of benchmarks indicate that non-chronological techniques are very effective in solving a large number of classes of SAT prob-

lems.

In general it can be stated that the goals of an efficient SAT backtracking algorithm are in the case of a satisfiable instance, to reach a leaf node of the search tree that contains the empty $\emptyset$ sub-formula as early as possible and by exploring the least number of branches. In the case of an unsatisfiable instance, the goal is to encounter a contradiction $\square$ as soon as possible enabling earlier backtracking and the waste of exploring useless regions.

4.2.2 Generic Structure

Marques et al. in [LMS03, Lyn01, MS99] provide a generic template capturing the main ideas, organization and primitives that are (or usually) incorporated into any SAT backtracking based algorithm. The overall organization of a generic backtrack SAT search algorithm, is shown in figure 4.5 and composed out of three main "engines":

1. **Decide()**, is the engine in charge of selecting a variable and its truth value at a given level, possibly according to some heuristic. This engine is the means by which new regions in the search space are explored. Different SAT backtrack based algorithms differ by the different heuristics/branching rules they utilize for this engine.

2. **Deduce()**, identifies truth assignments of other variables that are deemed necessary, that are implied by the variable and its truth value selected by Decide(). Whenever a clause becomes unsatisfied, implying that the current assignment is not a satisfying one, we have a conflict/contradiction and the engine will return a conflict indication. Besides variations in the decision engine, SAT backtrack based algorithms also differ by the methods they use to deduce necessary assignments. These are obtained by using different forms of *value probing* which can be thought of as forms of domain reductions of variables or as mechanisms of identifying suitable resolution operations. The most popular method is Boolean Constraint Propagation (BCP), discussed in section 4.2.3.

3. **Diagnose()**, identifies the causes of a given conflicting partial assignment. Diagnose() returns a backtracking decision level, which corresponds to a level to which the search may provably backtrack from useless regions where satisfying assignments cannot be found. This engine is the core of non-chronological backtracking algorithms.

```
procedure GenericSAT(CNF formula : φ)
    d := 0
    while Decide(φ, d)=DECISION do
        if Deduce(φ, d)=CONFLICT then
            δ := Diagnose(φ, d)
            if δ = −1 then
                return UNSATISFIABLE
            else
                clear(φ, d, δ)
                d := δ
            end
        end
        d := d + 1
    end
    return SATISFIABLE
end.
```

Figure 4.5: Generic backtrack SAT algorithm.

Given a SAT problem, formulated as a CNF formula ϕ, the algorithm conducts a search through the space of the possible assignments to the problem variables. At each stage of the search, a variable is selected and its truth value is determined with the **Decide()** function. A decision level d is associated with each selection. When there are no more variables to be selected, i.e. all variables are already decided, **Decide()** will signal that by returning NO-DECISION. Whenever a clause becomes unsatisfied the **Deduce()** function returns CONFLICT, a conflict indication which is then analyzed using the **Diagnose()** function. The diagnosis of a given conflict returns a backtracking decision level δ, which denotes the decision level to which the search process is required to backtrack to. A decision level equal to -1 indicates that backtracking is no longer possible, that all paths were searched, and therefore the formula is unsatisfiable. The **clear()** function clears all assignments from the current decision level d through the backtrack decision level δ. Further, since the search process should resume at the backtrack level, the current decision level d becomes δ. Finally, the current decision level d is incremented to the next level.

Given this generic algorithm the distinct variations among the different SAT algorithms can be obtained by different configurations of the three main engines. For example DPLL can be captured by the generic algorithm as follows: the decision engine selects the variable occurring in the first clause with minimal length, and assigns it the value true. The pure literal rule can also be embedded into this engine. The deduction engine simplifies the formula

according to the selected variable and its value, then applies the unit rule repeatedly until no unit clauses remain, and checks for conflicts. The diagnosis engine implements chronological backtrack by first undoing the assignments implied by setting the selected variable to its given truth value. If the selected variable's value has not yet been toggled then it is assigned the opposite value, otherwise the search backtracks to level $d-1$, that is, to its chronological partial assignment predecessor.

4.2.3 Boolean Constraint Propagation

Boolean Constraint Propagation (BCP) also called *unit propagation* (UP), is an operation that is employed in most if not all backtrack based SAT algorithms, it is the core of look-ahead techniques, and the most commonly used procedure for identifying necessary assignments. BCP consists of the iterated application of the *unit rule* (figure 4.6) , inducing the derivation of implications and necessary assignments. BCP extends the current truth assignment to the variables contained in all unit clauses. BCP will usually be found in the deduction engine as the means by which implied assignments are induced. However the most competitive look-ahead based algorithms also include BCP as part of the decision engine, as one of the operations carried by a heuristic to determine the next variable to branch on. Since BCP is the most time consuming step and tends to degrade algorithm performance when applied inappropriately, the questions of when to apply it, and how to implement it have attracted considerable interest lately. Another problem with BCP is that it cannot identify all necessary assignments. Consider the clauses $(p \vee q) \wedge (\bar{p} \vee q)$ for example, BCP will not be able to detect that for any value of p, q must be set to $true$.

procedure $BCP(CNF\ formula : \phi)$
    while there exists a unit clause $\{l\} \in \phi$ **do**
        $\phi := UnitRule(\phi', l)$
        if $\square \in \phi'$ **then return** CONFLICT
    end
    return NOCONFLICT
end.

Figure 4.6: the BCP procedure.

4.3 Branching Heuristics

Backtrack algorithms differ not only in the way they backtrack (chronological vs. non-chronological) but also in the way they select which variable to set

at each iteration/split, i.e., selecting the branching variable. This is done using heuristic knowledge. The name heuristic in our context is used to refer to the rules we believe will generate solutions without exhaustive search. These rules are often called branching rules. A key factor contributing to the overall performance of any SAT algorithm consists of an application of "clever" branching rules. A clever branching rule in the case of a satisfiable instance is a selection of variables and corresponding assignments in a sequel such that the amount of traversed search space is as small as possible. In the case of an unsatisfiable instance the variables and assignments selected by the branching rule should lead as early as possible to contradictions, thus minimizing search cost. It should be noted however that the problem of choosing an optimal sequence of variables and their assignments has been proven to be NP-hard as well as coNP-hard[Lib00]. It should also be noted that making good branching decisions is not the only consideration, the other aspect is the effort spent on acquiring the knowledge used to make this decision. A good branching rule is also one that is fairly simple to compute.

Branching heuristics can be classified according to their underlying "branching hypothesis" which is used to explain or motivate the branching rule. The *satisfaction* and *simplification* hypotheses were formalized in [HV95] as an explanation for the Jeroslow-Wang heuristics. In [LA97a] the *constraint* hypothesis was introduced to explain the heuristics used in *Satz*. The *satisfaction* hypothesis assumes that a branching rule performs better when it creates subproblems that are more likely to be satisfiable. The *simplification* hypothesis on the other hand assumes that a branching rule performs better when it creates subproblems of smaller size. Whereas the *constraint* hypothesis assumes that a branching rule works better when it creates subproblems with more and stringent constraints so that a contradiction will be found earlier. The constraint hypothesis can be thought of as an extension of the simplification hypothesis. While the simplification hypothesis considers unit propagation for only one branch below the current branching point, the aim of the constraint hypothesis is the preparation of stronger constraints for unit propagation in deeper levels[Li99].

Branching rules can be as simple as choosing the first remaining variable in the formula, choosing randomly one of the unassigned variables (this heuristic is called **RAND**), or it may be quite sophisticated as the ones used by *look-ahead* techniques (section 4.4). The most effective heuristics take into account the dynamic information provided by the backtracking search. This information may include, the number of literals of each variable in *unresolved* clauses (clauses that are neither satisfiable, unsatisfiable, or unit), the size of unresolved clauses, etc. Below is a discussion of the main and most popular heuristics. In section 4.4 we show how these heuristics and variations of them

are embedded into more sophisticated branching rules.

4.3.1 MOMs Heuristics

Probably the most popular known and utilized heuristic, which involves branching on the variable having **M**aximum **O**ccurrences in Clauses of **M**inimum **s**ize (name given by Pretolani in his Ph.D. thesis, 1993). The intuition behind it is if a variable appears in many of the smallest clauses then it is likely to induce other implied assignments and maximize the effect of BCP. More formally, let $f^*(l)$ be the number of occurrences of literal l in the shortest *open* (neither satisfiable, or unsatisfiable) clauses. The goal is to select an open literal l such that: $f^*(l)$ and $f^*(\bar{l})$ are maximal, and that the two quantities should roughly be equal (eliminating the possibility of l being a very good choice but $\bar{l}$ a very bad one, or vice versa). It is argued that MOMs has several disadvantages: one argument is that is does not utilize the full power of unit propagation because it only considers clauses of minimum size, this problem was resolved by combining MOMs with forward consistency checking (discussed in the next section). Another is that its effectiveness depends considerably on the structure of the problem instance, specifically the number of binary clauses (clauses with two literals) in the given formula, which is not such a big problem for real-world SAT instances since they tend to contain a large proportion of binary clauses [MS99, LA97a, LA97b, Fre95].

MOMs comes in different flavors and has been incorporated into most SAT solvers in one form or another via priority functions. A priority function for a given heuristic uses information about some open literal l to compute a numerical priority for l in accordance with the heuristic's overall strategy. In section 4.4 some of these functions are analyzed.

4.3.2 Jeroslow-Wang Heuristics[HV95]

The version originally proposed by Jeroslow and Wang in 1990 is now called the one-sided J-W (JW-OS) heuristic. Their heuristic seems to adopt the satisfaction hypothesis since it tries to estimate the contribution each literal is likely to make to satisfying the formula. This is done by estimating the probability that a literal when set to true will result a simplified formula that is satisfiable. In essence it attempts to predict the result of a look-ahead (unit propagation) without actually carrying it out. Each literal is scored as follows:

$$J(l) = \sum_{l \in C} 2^{-|C|}$$

Their reasoning is that clause C rules out $2^{n-|C|}$ truth assignments so that all the clauses that contain l rule out $\sum_{l \in C} 2^{n-|C|} = 2^n \sum_{l \in C} 2^{-|C|} = 2^n J(l)$. By maximizing the number of valuations ruled out by the clauses that are deleted(by setting l to true), they maximize the number that are not ruled out by those remaining, presumably making the simplified formula more likely to be satisfiable.

In [HV95] Hooker and Vinay investigate the motivation behind JW-OS and argue that is does not provide any motivation for unsatisfiable instances. Moreover they argue that even for satisfiable instances JW-OS is problematic. They propose first and second order estimates of satisfaction probability, but show that the performance did not improve and in some cases even worsened. This leads them to desert the satisfaction hypothesis in favor of the simplification hypothesis. Their reasoning is that a branching rule that maximizes the probability of satisfaction may cause the algorithm to backtrack from fewer nodes before finding a solution, if one exists. But since the subtrees rooted at most nodes will contain no solution in any case, it makes sense to branch in a such a way that these subtrees are as small as possible, thus simplify the problem as much as possible. This in turn will happen when unit propagation will eliminate as many literals and clauses as possible. They therefore analyze unit propagation as a random process modeled by Markov chains. This leads them to the *reverse* JW rule, branch to a literal l that maximizes $J(\bar{l})$.

At last, they suggest to improve JW-OS and their simplification rule by considering both $J(l)$ and $J(\bar{l})$, that is a two-sided JS heuristic (JW-TS): branch on a variable v that maximizes

$$J(v) + J(\bar{v})$$

over all variables in the formula, and brunch first to v if

$$J(v) \leq J(\bar{v})$$

and otherwise first to $\bar{v}$.

They provide empirical evidence to show that JW-TS is superior to JW-OS on most problem instances. Further support to Hooker and Vinay's simplification hypothesis is the fact that one of the best current implementations of a DPLL backtrack based algorithm, Satz, uses a branching heuristic based on maximizing the amount of unit propagation, thus the effect of BCP intended to simplify the problem.

4.3.3 MAXO Heuristic

This heuristic selects the literal with the **MAX**imum number of **O**ccurrences in the formula, and has been used in GRASP [MSS96] to experiment on the

effectiveness of non-chronological backtracking. The idea is that splitting on such a choice will have a wide spread effect in the formula. Another way to view this heuristic is as one that tries to satisfy as many clauses as possible.

4.4 Look-ahead Techniques and Algorithms

The more effective and recent algorithms such as C-SAT, TABLEAU, POSIT, and Satz employ a combination of a variable ordering heuristic such as MOMs, with a forward consistency checking (a UP operation), known as look-ahead methods in CSP terms. Look-ahead SAT related techniques evolved out of the need for more efficient algorithms used to prove unsatisfiability, the goal of theorem provers. They aim at augmenting the chance of reaching a dead end in the search tree earlier by exploiting the power of unit propagation (an application of the unit rule/unit resolution).

The idea behind a UP based heuristic is to examine a variable v by respectively adding the unit clauses $\{v\}$ and $\{\bar{v}\}$ to the formula ψ and independently execute two unit propagations, one for v and one for $\bar{v}$. This allows to take all clauses into account, and not only the shortest onces, in order to evaluate or weigh a variable. Another advantage is that this will also uncover the *failed literals*, the ones that when set to true generate an empty clause, falsifying ψ, in a single unit propagation. However the disadvantage of UP based Heuristics is that they may significantly increase the effort spent on selecting a branching variable, since every open variable must be examined by two unit propagations. Aware of this fact, the more efficient algorithms involve tradeoffs. They restrict the number of variables examined by unit propagation at each node by first executing simpler heuristics such as MOMs in order to select which variables will be examined by unit propagation. This section summarizes, in an approximate chronological the most recent and efficient UP based algorithms.

4.4.1 TABLEAU

TABLEAU [CA93, CA96] evolved from an implementation of Smullyan's "tableau" based inference procedure, and as an extension to the DPPL procedure. Crawford and Auton observed that a simple variable selection heuristic can make several orders of magnitude difference in the average size of the search tree. Their first observation is that there is no need to branch on variables that occur in Horn clauses. Instead they focus on non-Horn clauses based on the idea that once an assignment that satisfies all the non-Horn clauses is found, the remaining Horn clauses can be satisfied by falsifying all the remaining unassigned variables. The first step of TABLEAU's branching rule is to construct a list V

of all variables that occur positively in non-Horn clauses with a minimal number of non-negated variables. They then impose a priority order on these variables using two more heuristics. The preference would have been to variables that cause the greatest amount of unit propagation, which is not cost-effective to actually compute. Instead they approximate the number of unit propagations by counting the number of binary clauses in which the variables appear. When a tie occurs another heuristic, that counts the number of unassigned *singleton neighbors*, is used. A singleton neighbor is a literal that appears in the same clause as the variable in consideration and only in that clause. Their experiments showed that on hard 3SAT problems TABLEAU grows at a rate of $2^{n/17}$, as opposed to $2^{n/5}$ for DPLL, thus allowing the algorithm to handle problems three times as large.

4.4.2 C-SAT

Dubois et al. [DABC93] first present A-SAT which is essentially the DPLL procedure augmented solely by a MOM's heuristic. They mention that the development of C-SAT is motivated by creating a more efficient algorithm for proving unsatisfiability, and state two criteria which in their opinion a branching heuristic should aspire to meet in order to achieve this goal. The first is that the heuristic should balance the search tree, and the second is to reduce the mean hight of the search tree. They demonstrate the development of C-SAT's heuristic by presenting intermediate heuristics and analyzing their performance. The first is:

$$B_1 = max_{x \in I}[f(x) + f(\bar{x}) + \alpha \; min(f(x), f(\bar{x}))]$$

where I is a set of unassigned variables at a current node, $f(x)$ and $f(\bar{x})$ are the number of positive and negative occurrences of x in the shortest clauses, and α a weighting constant determined empirically (1.5 in C-SAT), giving importance to the balancing of the signs of the occurrences with respect to the total number of occurrences. They next introduce weighting coefficients such that the heuristic will take into account not only the occurrences of variables in the shortest clauses but in all clauses weighted as a function of their lengths, similar to the J-W heuristics.

$$B_2 = max_{x \in I}[f(x) + f(\bar{x}) + \alpha \; min(f(x), f(\bar{x}))]$$

with

$$f(x) = \sum_r P_r |x|_r$$

where $|x|_r$ is the number of occurrences of x in clause of length r, and $P_r = -ln(1 - 1/(2^r - 1)^2)$. Next they demonstrate by an example that B_2 is not

enough, for the following formula:

$$(x \vee y), (x \vee t), (\bar{y} \vee \beta_1), (\bar{y} \vee \beta_2), (\bar{y} \vee \beta_3), (\bar{t} \vee \gamma_1), (\bar{t} \vee \gamma_1)$$

B_2 will select y where x allows more clauses to be reduced. They propose a week form of constraint propagation as a solution.

$$B_{C\text{-}SAT} = max_{x \in I}[F(x) + F(\bar{x}) + \alpha\ min(F(x), F(\bar{x}))]$$

with

$$F(x) = f(x) + \sum_{y \in x \vee y} f(\bar{y})$$

i.e. compute the weighted number of occurrences of literals of which the complemented ones are associated with the evaluated variable x in clauses of length 2.

Finally they discuss the inclusion of unit propagation in C-SAT, and refer to it as a *local processing* operation. Aware of the fact that UP is an expensive operation to carry at each node, they suggest that it should be carried only where it is likely to succeed. For nodes near the root it is not likely to succeed for hard instances, therefore they propose to apply it only on nodes where a certain percentage θ_1 of variables have been already assigned. Moreover, it may still be inefficient to apply UP on all the remaining unassigned variables, which is why they suggest to apply it only on a certain portion θ_2 of the unassigned variables listed in decreasing order according to the function $h(x) = f(x)+f(\bar{x})$, and where $\theta_1 = 0.5 \times n$ and $\theta_2 = 0.05 \times n$ are empirically set.

4.4.3 POSIT

POSIT (PrOpositional SatIsfiability Testbed)[Fre95] was developed and implemented by Freeman as part of his Ph.D. thesis. It was written in C and contained about 6700 lines of code, excluding comments and blank lines. Before POSIT begins the search a simplification preprocessing phase takes place by executing four operation in order: BCP, the pure literal rule, deleting (using the elimination rule) all variables p such that either p or $\bar{p}$ is a *singleton* (a literal that occurs only once in the formula), and deleting all variables p such that either p or $\bar{p}$ is a *doubleton* (a literal that occurs only twice in the formula). If the third or forth steps create unit clauses or pure literals then the first two steps must be repeated. The outcome of this preprocessing phase assuming a contradiction has not been encountered (the formula is unsatisfiable) is that every clause has at least two literals, no pure literals, and every variable occurs at least six times (three positively and three negatively).

Next we describe POSIT's branching rule, which Freeman refers to as the *Apply-Heuristic* function:

1. The first step is a simplification step where POSIT attempts to detect failed literals (literals that cause the formula to become false), and simplifies the formula accordingly. It is achieved by first calculating the number of positive and negative occurrences of each open (unassigned) variable v at the current stage in open (unassigned) binary clauses, denoted $n(v)$ and $n(\bar{v})$ respectively. Then for each of these variables, if the number of positive and negative occurrences is larger than a certain bound it will set the variable to true or false depending on whether $n(v) > n(\bar{v})$ and execute BCP. If BCP causes the formula to become false then it will toggle the variable's truth value and run BCP again. If again BCP causes then formula to become false then POSIT must backtrack. In the case that a failed literal is detected the formula is simplified, if not then the changes caused by the execution of BCP must be undone. It should be noted that POSIT forces a limit on the number of unsuccessful attempts to detect failed literals, thus the number of BCP runs is limited to a small number.

2. In this step POSIT runs a Mom's heuristic on all the open (unassigned) variables. This is done as in step 1 by, first calculating $n(v)$ and $n(\bar{v})$. Then depending on whether $n(v) > n(\bar{v})$, either v or $\bar{v}$ (in this case $\bar{v}$) will be added to an initial list of *candidate* branching literals with priority $\alpha(n(v), n(\bar{v}))$ calculated by:

$$\alpha(n(v), n(\bar{v})) = n(v) \cdot n(\bar{v}) \cdot 2^x + n(v) + n(\bar{v})$$

   where x is sufficiently large. The purpose of multiplying by 2^x is to enforce preference for propositions v such that both $n(v)$ and $n(\bar{v})$ are non-zero, and that $n(v)$, $n(\bar{v})$ are roughly balanced. The remaining terms in the sum are for discriminating among variables for which either $n(v)$ or $n(\bar{v})$ is zero.

3. This step is executed in case the initial *candidate* list generated in step 2 is empty, which implies that the formula is either satisfied or does not contain any open binary clauses. If the formula is satisfied then POSIT will return the truth assignment and terminate. Otherwise there are no open binary clauses and POSIT will use a modified version of Jeroslow and Wang's weighted occurrences heuristic to compute priorities for open variables. For each open variable POSIT will compute $weighted(v) = \sum_{v \in C} 2^{-|C|}$ and $weighted(\bar{v}) = \sum_{\bar{v} \in C} 2^{-|C|}$, if $weighted(v) > weighted(\bar{v})$ then $\bar{v}$ will be added to the initial *candidate* list with priority $\alpha(weighted(v), weighted(\bar{v}))$, otherwise v will be added with the same priority. POSIT now considers the initial candidate list as finalized and jumps directly to the last step.

4. In the case that the initial candidate list is not empty POSIT will run a BCP-based heuristic to compute a more accurate priority, for each literal in the candidate list and add the ones with the highest priority to a new final candidate list. This is done as follows: For each literal l in the initial list in decreasing order of their priority POSIT will

   (a) Set l to true, run BCP and compute two values: $\mathit{induced}(l)$-the number of necessary assignments induced by setting the l to true and running BCP, and $\mathit{bin}(l)$ the number of clauses that became binary. Then undo changes made by executing BCP.

   (b) If l is detected as a failed literal toggle its truth value, i.e. set $\bar{l}$ to true and run BCP. If $\bar{l}$ is also detected as a failed literal POSIT must backtrack, else continue with the next literal.

   (c) If l is not a failed literal run BCP on $\bar{l}$ and count $\mathit{induced}(\bar{l})$ and $\mathit{bin}(\bar{l})$ and then undo changes made by BCP. If $\bar{l}$ is a failed literal make l true, run BCP and continue with the next variable.

   (d) When neither l nor $\bar{l}$ is a failed literal calculate:

   $$\mathit{total\text{-}cost}(l) = \alpha(\mathit{induced}(l) + \mathit{bin}(l), \mathit{induced}(\bar{l}) + \mathit{bin}(\bar{l}))$$

   This value is the new priority of l, which is then compared with the largest value of *total-cost* computed so far. If the new *total-cost* is larger it replaces the old one and literal l is placed at the front of the finalized candidate list.

   (e) same as for the first step, there is also a bound in this step on the number of attempts to detect failed literals.

5. The last step just selects the next branching literal be the literal in the front of the finalized list. If there is no such one, which can happen due to lots of failed literal detection, POSIT will go back to the first step.

4.4.4 Satz

Satz developed by Chu Min Li and Anbulagan [LA97a, LA97b, LG03] is currently the best UP based algorithm for solving random SAT instances and also very competitive on structured problems for which look-back based algorithms are superior. Like the others TABLEAU, C-SAT, and POSIT it exploits the power of unit propagation to determine the next branching variable. However since examining variables by unit propagation is time consuming, the amount of variables examined should be restricted. The superiority of Satz over the other procedures is due to a vast amount of experiments to correctly choose the look-ahead search space. Based on experimental evaluations of different UP

restrictions they propose a heuristic that will dynamically restrict the number of unit propagations ensuring that at least T variables selected by a Mom's heuristic will be examined by unit propagation. Satz is somewhat an optimal combination of a Mom's heuristic with unit propagation.

Restricting the number of variables examined by UP at each node is done using ***PROP***, a predicate whose denotational semantics is the set of variables to be examined at a search tree node, i.e. variable v is to be examined if and only if $PROP(v)$ is true. By changing $PROP$ different restrictions on UP can be introduced, which in turn results in a different DPLL based procedures. In [LA97b] 12 different $PROP$ predicts were experimentally evaluated to potentially uncover the optimal one. More formally we can define a mapping

$$PROP : \{v | v \ is \ a \ free \ variable\} \longrightarrow \{true, false\}$$

and view $PROP$ as denoting the set of variables to be examined, free variable v is to be examined if and only if $PROP(v) = true$. Based on the intuition behind MOM's heuristic, we want to examine, focus on variables that occur in shortest, say binary, clauses, thus $PROP$ must be defined more realistically to accommodate this notion. Define $PROP_{ij}(v) = true$ if and only if variable v has at least i occurrences in binary clauses such that at least j occurrences are positive and j occurrences are negative. For example, $PROP_{31}$ gives the set of variables that have at least 3 occurrences in binary clauses and at least 1 occurrence is positive, and at least 1 occurrence is negative. Two special cases are defined: $PROP_0(v) = true$ for every free variable v, and $PROP_a(v) = false$ for every every free variable v, where a is some constant. The use of $PROP_0$ results a UP heuristic that examines every free variable at the current node, and $PROP_a$ results a UP heuristic that does no examine any free variables. $PROP_0$ and $PROP_a$ result two extremes, between then there are many other possible $PROP$ predicates. The investigation of these possible predicates lead to the $PROP_z$ predicate used in Satz.

The idea is that close to the root examining more variables by unit propagation is the best way to reduce the search space. However, in general it can be stated that close to the root, for random SAT (3SAT) formulas, which Satz focuses on, it is less likely to find binary clauses. Therefore any reasonable setting of $PROP_{ij}$ will not give enough variables to be examined. On the other hand close to the leaf levels, there will be many binary clauses, which will cause $PROP_{ij}$ to generate a large set of variables to be examined, an undesired property for a UP heuristic. Satz, dynamically restricts the number of free variables that will be examined at each node by the use of $PROP_z$, defined as follows:

$$PROP_z(v) = \begin{cases} PROP_{41}(v), & if \mid PROP_{41} \mid \geq T \\ PROP_{31}(v), & if \mid PROP_{31} \mid \geq T > \mid PROP_{41} \mid \\ PROP_0(v), & otherwise \end{cases}$$

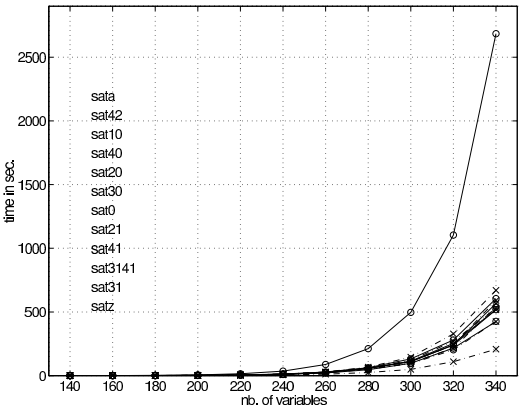


Figure 4.7: Mean run time for 12 different $PROP$ predicates resulting 12 different algorithms, as a function of n for hard random 3SAT problems, with $m/n = 4.25$, and with $PROP_z$ having the best performance.

Where $\mid PROP_{ij} \mid$ is the number of variables that satisfy $PROP_{ij}(v) = true$, and where T is a predefined constant, which was empirically set to 10. By using $PROP_z$, when there are few binary clauses such as when the search is close to the root $PROP_0$ will be used, and when the search is close to leaf nodes many binary clauses are likely to be found, and $PROP_{41}$ or $PROP_{31}$ will be used. Generally $PROP_{41}$ will be used more than $PROP_{31}$ as the search gets closer to leaf nodes. Let $diff(F_1, F_2)$ be a function that gives the difference in the number of binary clause between F_1 and F_2, and $H(v)$ a priority function used by Freeman in POSIT, then Satz's branching rule is given in figure 4.8.

The main differences between Satz and the other algorithms presented here is that Satz seems to be employing the *constraint hypothesis* whereas the others employ the simplification or satisfaction hypothesis. Moreover Satz does not impose an upper bound on the number of variables that will be examined by unit propagation like with TABLEAU and POSIT, instead it specifies a lower bound T. Experiments show that this feature causes Satz to examine more nodes than the others at each node.

Three constraint based improvements are proposed by Li in [Li99]. The underlying idea of these improvements is to reduce the width of a search tree, which based on experiments will contribute more to the speed up of a DPLL procedure than reducing the mean hight of the search tree. The first improvement is a preprocessing stage that adds resolvents of length ≤ 3 into the input

for each free variable v such that $PROP_z(v) = true$ **do**
let F' and F'' be two copies of F
begin
$F' := UnitPropagation(F' \cup \{v\})$
$F'' := UnitPropagation(F'' \cup \{\bar{v}\})$
if both F' and F'' contain an empty clause **then**
return "F is unsatisfiable"
if F' contains an empty clause **then** $v := 0,\ F := F''$
else if F'' contains an empty clause **then** $v := 1,\ F := F'$
if neither F' nor F'' contains an empty clause **then**
let $w(v)$ denote the weight of v
$w(v) := diff(F', F)$ and $w(\bar{v}) := diff(F'', F)$
end.

for each variable v **do** $H(v) := w(\bar{v}) \cdot w(v) \cdot 1024 + w(\bar{v}) + w(v)$

Branch on free variable v such that $H(v)$ is the greatest

Figure 4.8: The branching rule of Satz.

	300 vars 300 problems		350 vars 250 problems		400 vars 100 problems	
System	*time*	*t_size*	*time*	*t_size*	*time*	*t_size*
C-SAT	77	49567	512	275303	3818	1624869
Tableau	79	43041	558	253366	4544	1524551
POSIT	57	61797	474	400588	3592	2751611
Satz	34	32780	203	174337	1207	916569

Figure 4.9: Mean run time (in seconds) and mean search tree size for hard random 3SAT problems, with $m/n = 4.25$.

formula to increase the probability of finding unit or binary clauses close to the root. The second improvement is to distinguish binary clauses removing more solutions from other binary clause, since a stronger constraint is one that suppresses more solutions. The third improvement is to introduce unit propagations of second level to detect dead ends faster and as a remedy for the draw back of two-sided branching rules that prefer balanced trees.

In [LG03] Li and Sylvain suggest say that much of the research effort on DPLL based procedures concentrates on finding better heuristics to select the branching variable in order to minimize the search tree size. However, they claim that there has been a marked slow down in improvements of the DPLL methods on hard random 3SAT in recent years. This suggests that we may be close to the minimum size trees, and that the branching heuristics are probably

close to optimal. They support their claims by experiments.

4.5 Non-Chronological backtracking, Look-Back techniques

As stated in section 4.2.1 naive backtracking algorithms often explore regions of the search space that are clearly devoid of solutions, or rediscover the same contradictions in other regions of the search space. The general term used to describe such "pathologies" is *thrashing* [Fre95]. The reason useless regions are explored is because of *futile* backtracking. Futile backtracking occurs when the algorithm reaches a dead end in the search space and backtracks to a choice that was not in any way responsible for contradiction. A remedy for this problem is non-chronological/intelligent backtracking, a technique used to ensure that the algorithm will backtrack to a variable that is somehow responsible for at least one of the derived contradictions. This is usually achieved by utilizing another technique often referred to as: *clause learning*, *lateral pruning*, *nogood*, and *constraint recording*, in which whenever the algorithm hits a dead end, it generates a new clause/constraint containing the variables responsible for the contradiction, and adds it to original formula. This new clause is then used to back up the search tree to one of the causes of failure, and also to prevent the algorithm from rediscovering the same contradictions in other regions of the search space, i.e. making the same mistakes again [LMS03, Lyn01].

procedure *Diagnose*($CNF\ formula : \phi$, *decision level* : d)
    ω := Create_Conflict_Induced_Clause()
    AddTo_CNF_Formula(ω)
    δ :=Compute_Max_Decision_level(ω)
    return δ
end.

Figure 4.10: Outline of the diagnose engine in charge of non-chronological backtracking.

4.5.1 Implication Graphs

The implication of necessary assignments during a BCP process can be expressed as an **Implication Graph**. An implication graph G is a direct acyclic graph (DAG), where each vertex represents a variable and its assignment. Usually the decision level in which a variable was assigned will also be indicated, e.g. $x = 1@6$ or $+x(6)$ means variable x is assigned true at decision level 6. It is common to refer to the vertices that have directed edges to x as x's *antecedents*

vertices or assignments usually denoted $A(x)$. These correspond to the set of variables and their assignments that are directly responsible for implying the assignment of x due to some clause ω. The incident edges of x are usually labeled with the clauses (reasons) that lead to x's assignment. A vertex x with no antecedents, i.e. $A(x) = \emptyset$ is referred to as the *decision variable*, which corresponds to the splitting variable or the variable selected by the decision engine. The decision level of an implied variable x denoted $\delta(x)$ is the maximum decision level of any of its antecedent variables, i.e $\delta(x) = max\{\delta(y)|y \in A(x)\}$. A conflict arises during BCP when some clause becomes unsatisfiable or equivalently when a variable gets assigned both truth values. It is customary [MSS99] to add a special conflict vertex κ to indicate the occurrence of that conflict. An example of an implication graph is illustrated in figure 4.11. In actual implementations, the implication graph is maintained during the execution of BCP by associating each assigned (implied) variable with a pointer to its antecedent clause. The antecedent clause of an implied variable is the clause that was unit at the time when the implication happened. By following the antecedent pointers, the implication graph can be constructed when needed.

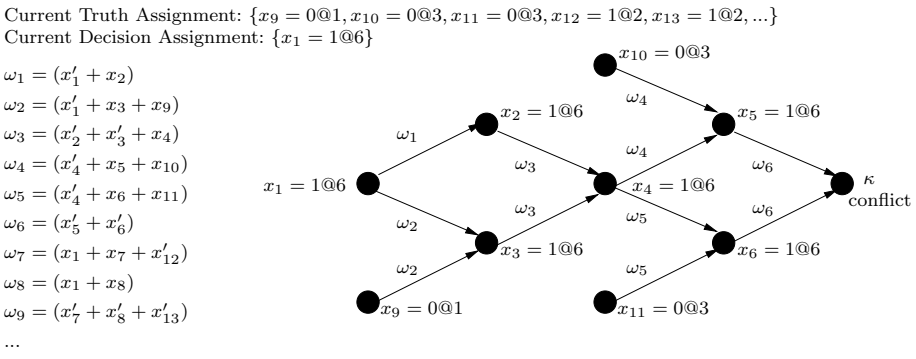


Figure 4.11: A typical implication graph (taken from[LMS03], where + stands for disjunction).

4.5.2 Conflict Analysis

Conflict analysis is a procedure that finds the reason for a conflict and tries to resolve it. It indicts that there is no solution for the problem in a certain search space, and guides the search to a new region to continue the search. The simplest conflict analysis procedure is found in the original DPLL and the look-ahead algorithms covered in section 4.4. These algorithms keep track of whether or not a decision variable has been flipped (both truth values were

tried). When a conflict occurs, the conflict analysis procedure looks for the decision variable with the highest decision level that has not been flipped, marks it flipped, undoes all the assignments between that decision level and the current one, and then tries the other truth value for the variable. This simple conflict analysis procedure works pretty well for randomly generated problems, because random problems don't have a structure, and learning from certain regions of the search space will generally not help the search in other regions. For structured problems more advanced conflict analysis techniques that rely on implication graph analysis have been shown to be profitable[ZMMM01].

4.5.3 Clause Learning

The more advanced conflict analysis techniques are based on the process of *learning* new clauses, called *conflict clauses*, from the implication graph. By inspecting the implication graph we can reveal the variables and their assignments that are directly responsible for the conflict. The conjunction of these variables assignments represent a sufficient condition for the conflict to arise. The negation of this conjunction yields a new clause, which did not exist in the original formula. This new clause provides a primary mechanism by which certain combinations of variable assignments are prevented from occurring again, thus pruning the search space, and by which non-chronological backtracking may take place. Figure 4.11 illustrates clause learning. The decision variable is x_1 and is is assigned true at decision level 6. The assignment of x_1 with the assignments of $x_9, x_{10}, x_{11}, x_{12}, x_{13}$ made at previous decision levels yields a conflict involving clause ω_6. By inspection of the implication graph, we can conclude that a sufficient condition for this conflict to be identified is:

$$(x_1 = 1) \wedge (x_9 = 0) \wedge (x_{10} = 0) \wedge (x_{11} = 0)$$

By negating this conjunction/condition we create a new clause

$$\omega_{10} = (\bar{x_1} \vee x_9 \vee x_{10} \vee x_{11})$$

that will prevent the same set of variable assignment from occurring again during the search.

Standard clause learning such as the one described above, suffers from three main drawbacks. The first is that clause learning introduces significant overhead which for some instances can lead to a large increase in run times. This drawback is inherent to the algorithm's frame work, however empirical evidence suggests that the over all performance gains outweighs the additional overhead for most of the structured SAT instances. The second drawback is that an unbounded amount of learning is impractical due to memory limitations. The

size of the formula when new learned clauses are added, grows with the number of backtracks. Such growth can be exponential in the number of variables in worst case, and thus must somehow be controlled. The third problem is that that the larger (more literals) the learned clauses are, the less useful they are for pruning purposes [MSS96]. Solutions to the last two drawbacks are based on a selective choice of clauses that are to be added to the formula. Among the more popular solutions are variations of two ideas. First, clauses are temporarily stored as long as they imply other assignments or they are unit, while being discarded as soon as the number of unassigned literals is greater than one. This in turn reduces the effectiveness of conflict analysis, so a variation of this idea called *relevance-based learning*, introduced in [BS97], consists of deleting recorded clauses only when the number of unassigned literals becomes larger than some m. The Second idea for solution, says that clauses with a size less than a threshold k are kept during subsequent search, whereas larger clauses are discarded as soon as the number of unassigned literals is greater than one. This technique is referred to as *k-bounded learning*, introduced in [MSS96].

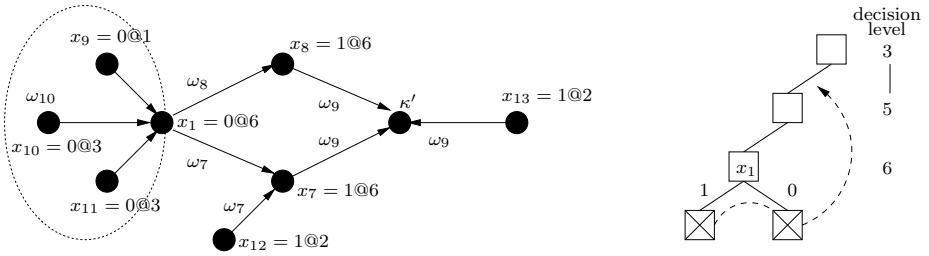
Further enhancements to the standard clause learning mechanism involve a more careful analysis of the implication graph ,to generate shorter and more useful clauses. For example, both the clauses $(\bar{x_1} \vee x_9 \vee x_4)$ and $(\bar{x_4} \vee x_{10} \vee x_{11})$ could have been learned. They are both stronger than the single clause ω_{10} learned earlier, and are potentially better for future applications of BCP in the presence of partial assignments.

Techniquely a clause is learned by a bipartition of the implication graph, where one partition is called the *reason* and the other called the *conflict*. All the vertices in the reason partition that have at least one edge connected to a vertex in the conflict side, comprise the reason for the conflict and can be used to construct a new conflict clause. This sort of bipartition is usually referred to as a *cut*, and different cuts correspond to different learning schemes. For example a cut depicted by a line going through edges ω_4 and ω_5 in figure 4.11 would enable us to learn the clause $(\bar{x_4} \vee x_{10} \vee x_{11})$. Different learning schemes will be discussed in more detail in conjunction with the discussion of the algorithms that utilize them.

4.5.4 Non-chronological Backtracking

Recorded clauses can be used to perform non-chronological backtracking, that takes place when all the literals in the newly created conflicting clause correspond to variables that were assigned at decision levels that are lower than the current decision level. In this case, the backtracking decision level is defined as the highest decision level of all variable assignments of the literals in the newly recorded clauses. To illustrate non-chronological backtracking, consider

the example of figure 4.12 that continues the example in figure 4.11 after learning the clause $\omega_{10} = (\bar{x_1} \vee x_9 \vee x_{10} \vee x_{11})$. At this stage BCP induces that $x_1 = true$ because clause ω_{10} becomes unit at decision level 6. By inspection of the implication graph (figure 4.12) we see that clause ω_7 and ω_8 cause a conflict involving clause ω_9. All the variables involved in this conflict are assigned at decision levels less than 6, the highest of which is 3. Therefore the algorithm can backtrack directly to decision level 3.



(a) Conflicting implication sequence

(b) Decision tree

Figure 4.12: computing a backtrack decision level (taken from[LMS03]).

4.6 Look-Back Algorithms

4.6.1 rel-SAT

rel-SAT [BS97] introduced *relevance-based learning*, hence its name rel-SAT. Is was also one of the first SAT solvers to incorporate learning and non-chronological backtracking. Despit the fact the learned clauses are created by analysis of the implication graph, they can also be viewed as the result of applying selective resolution. rel-SAT's diagnose engine generates learned clauses/conflict clauses by repeatedly resolving the clauses that cause the current conflict, i.e. resolving the conflicting clauses with its antecedents. In particular, when ever a conflict occurs a clause becomes unsatisfiable or equivalently a variable has both truth values excluded. The diagnose engine will construct a conflict clause C, the reason for this failure by resolving the respective clauses responsible for it, and then back up to the most recent assignment of a variable in C. Suppose $x = 1$ was the most recent assignment of variable x. If $x = 0$ is excluded by some clause D, the the engine will create a new conflict clause E by resolving C and D, and back up to the most recently assigned variable in E. Otherwise C becomes the reason for excluding the assignment $x = 1$, the current assignment of x will be set to $x = 0$ (flipping the variable's truth

value), and the algorithm will proceed with the search. This process will result a conflict clause involving only the decision variable of the current decision level and variables assigned at decision levels less than the current one. For example, the conflict clause ω_{10} that resulted from the analysis of the implication graph in figure 4.11, can be derived by resolution as follows: the conflicting clause is ω_6 and its antecedent clauses are ω_5 and ω_4. Resolving ω_6 with ω_5 and then with ω_4 results the clause $(\bar{x_4} \vee x_{10} \vee x_{11})$ which is then resolved with ω_3 for excluding both $x_4 = 1$ and $x_4 = 0$, to produce $(\bar{x_2} \vee \bar{x_3} \vee x_{10} \vee x_{11})$. This clause is then resolved with ω_1 for excluding both $x_2 = 1$ and $x_2 = 0$ and then ω_2 for excluding both $x_3 = 1$ and $x_3 = 0$ to produce $\omega_{10} = (\bar{x_1} \vee x_9 \vee x_{10} \vee x_{11})$.

In the implication graph representation, rel-SAT will put all the variables assigned at the current decision level, except for the decision variable in the conflict partition, and all the variables assigned prior to the current decision level together with the decision variable in the reason partition.

Aware of the problems created by unrestricted clause learning the authors, Bayardo and Schrag, of rel-SAT propose and experiment with two restricted learning schemes. The first which they call *size bounded* learning of order i, retains only the learned clauses containing at most i variables. The second which they call *relevance-bounded* learning of order i, maintains only the clauses that contain at most i variables whose assignments have changed since the clause was derived. They experiment with several variations of DPLL, including one version that does not employ any look-back enhancements, one that erases learned clauses right after they are used to determine a backtracking level, and two versions size-sat(i) and rel-sat(i) that use learn orders of 3 and 4. The reason learn orders 3 and 4 were chosen is that higher learn orders result in too high of an overhead to be useful, and lower learn orders had little effect. Their experiment was applied on real-world SAT instances and showed that the DPLL versions employing the relsat(i) learning scheme outperformed the others by orders of magnitude for most instances, where relsat(4) usually outperforms relsat(3). Based on their observations they suggest that size-bounded learning is effective when instances have relatively many short "nogoods" (partial assignments which cause the formula to be unsatisfied), which can be derived without deep inference. Relevance-bounded learning based on the other hand, is effective when many sub-problems corresponding to the current assignment also have this property. Real-world instances often contain subproblems with short easily derived nogoods, explaining the superiority of the rel-sat learning scheme, whereas hard random 3SAT instances have very short nogoods, but which require deep inference to derive, making look-back enhancements of little use.

4.6.2 GRASP

GRASP (**G**eneric sea**R**ch **A**lgorithm for the **S**atisfiability **P**roblem) [MSS99, MSS96] was developed by Marques and Sakallah in 1996. Their algorithm follows the standard template of SAT algorithms discussed in section 4.2.2. It employs the basic look-back and clause learning techniques discussed thus far, and is enhanced by two novel improvements. The first is the *k-bounded* restricted learning (see section 4.5.3), used to control the growth of the formula. The second is a more advanced conflict analysis procedure that generates stronger implicates by identifying and partitioning the implication graph according to unique implication points.

In an implication graph, vertex x is said to dominate vertex y if and only if any path from the decision variable of the decision level of x to y needs to go through x. A *unique implication point* (UIP) is a vertex at the current decision level that dominates the conflict vertex (or both vertices corresponding to the conflicting varaible), where the decision variable is always a UIP. For example, the set of UIPs in figure 4.11 is $\{(x_1 = 1), (x_4 = 1)\}$. Intuitively, a UIP can be viewed as a single reason for triggering an implication sequence that leads to a conflict at the current decision level. To illustrate this consider the implication sequence of figure 4.11 again, the UIP $x_4 = 1$ together with the earlier assignments $x_{10} = 0$ and $x_{11} = 0$ is a sufficient condition for triggering an implication sequence leading to the same conflict. Hence, the clause $(\bar{x_4} \vee x_{10} \vee x_{11})$ can be added as a newly learned clause to the formula. Moreover by analyzing the relationship between the two UIPs $x_1 = 1$ and $x_4 = 1$ the clause $(\bar{x_1} \vee x_9 \vee x_4)$ can also be derived. Using these two clauses GRASP backtracks first to the highest decision level of all variable assignments in the first clause, which causes x_4 to be flipped at the current decision level, which in turn causes x_1 to be flipped at the current decision level. Flipping x_1 causes another conflict depicted by figure 4.12. When it is found that flipping the decision variable still yields a conflict, GRASP will enter a *backtracking mode*. Besides the clauses added earlier, GRASP also adds when in the backtracking mode another clause, call a backtracking clause, which involves only variables assigned at previous decision levels.

The procedure for constructing conflict clauses and backtracking clauses can be described formally as follows: let x denote a variable and $v(x)$ its truth assignment at the current decision level, and let $\mathcal{K}$ be a conflict vertex. We partition the antecedents of x, $A(x)$ into variable assignments that were made at the current decision level $\Sigma(x)$, and those made before $\Lambda(x)$ by,

$$\Sigma(x) = \{(y, v(y)) \in A(x) | \delta(y) < \delta(x)\}$$

$$\Lambda(x) = \{(y, v(y)) \in A(x) | \delta(y) = \delta(x)\}$$

Determination of the conflicting assignments is computed by:

$$A_c(\mathcal{K}) = causes\_of(\mathcal{K})$$

where

$$causes\_of(x) = \begin{cases} (x, v(x)) & if\ A(x) = \emptyset \\ \Lambda(x) \cup [\ \bigcup_{(y,v(y))\in\Sigma(x)}\ causes\_of(y)] & otherwise \end{cases}$$

The conflict induced clause corresponding to $A_c(\mathcal{K})$ is computed by:

$$\omega_c(\mathcal{K}) = \bigvee_{(x,v(x))\in A_c(\mathcal{K})} x^{v(x)}$$

where $x^0 \equiv x$ and $x^1 \equiv \bar{x}$. For example, application of this procedure to the implication graph depicted in 4.12 can be used to construct a backtracking clause, where

$$A_c(\mathcal{K}') = \{x_9 = 0@1, x_{10} = 0@3, x_{11} = 0@3, x_{12} = 1@2, x13 = 1@2\}$$

$$\omega_c(\mathcal{K}') = (x_9 \vee x_{10} \vee x_{11} \vee \bar{x}_{12} \vee \bar{x}_{13})$$

The procedure for constructing stronger clauses based on a set of UIPs $U = \{(u_1, v(u_1)), ..., (u_k, v(u_k))\}$ is very similar to the previous and is based on reconvergence between UIPs.

$$causes\_of(x, u_i) = \begin{cases} (u_i, v(u_i)) & if\ x = u_i \\ \Lambda(x) \cup [\ \bigcup_{(y,v(y))\in\Sigma(x)}\ causes\_of(y, u_i)] & otherwise \end{cases}$$

where $(u_i, v(u_i)) \in U$ and $causes\_of(x, u_i)$ are to interpreted as the set of antecedent assignments of x due to UIP u_i. Conflict clauses are now created for every pair of UIPs as well as for the last UIP and the conflict vertex.

$$\omega_c(u_i, u_{i+1}) = [\bigvee_{(x,v(x))\in causes\_of(u_i,u_{i+1})} x^{v(x)}] \vee u_{i+1}^{\neg v(u_{i+1})}$$

$$\omega_c(\mathcal{K}, u_k) = \bigvee_{(x,v(x))\in causes\_of(\mathcal{K},u_k)} x^{v(x)}$$

For example, application of this procedure to the implication graph depicted in 4.11 can be used to construct the two stronger conflict clauses, where

$$\omega_c(x_1, x_4) = (\bar{x_1} \vee x_9 \vee x_4)$$

$$\omega_c(\mathcal{K}, x_4) = (\bar{x_4} \vee x_{10} \vee x_{11})$$

4.6.3 Chaff

Chaff [MMZ+01] is currently one of the best SAT solvers. Chaff has managed to achieve one to two orders of magnitude performance improvement on difficult SAT benchmarks in comparison with other look-back based solvers. One of its implementations called zChaff has won the SAT 2002 Competition in two benchmarking categories. The success of chaff is due to careful engineering of all aspects of the search, rather than to focus on complex ordering heuristics or learning mechanisms. The authors of chaff observed that for most SAT problems over 90% of the solvers' run time is spent on the BCP process. Thus, an efficient implementation of the BCP engine is key to any SAT solver. The performance of a BCP engine is determined by its ability to accurately and efficiently identify when clauses become satisfied unsatisfied and unit by the addition of a new variable assignment. Chaff uses a very efficient way to monitor the current set of unit clauses without searching the whole clause database. This mechanism also has the advantage of more efficient restoration (unassignment) of variables during backtrack.

The most simple and intuitive BCP implementation, is one which scans the entire database of clauses that contains a literal that the current assignment sets to false, and then checks if any clause became unit as a result of this assignment. This in effect could be achieved by keeping a counter for each clause of how many false valued literals are in the clause, and modify the counter every time a literal in the clause is set to false. When backtracking these counters and variable assignments must be restored. The counters scheme is attributed to Crawford and Auton in their design of the TABLEAU SAT solver [CA93], where according to [ZM02, LMS02] similar schemes are employed in more recent SAT solvers such as GRASP and Satz. These schemes however are not the most efficient. If a clause has n literals, there is no real reason to visit it when $1, 2, 3, ..., n-1$ literals are set to false. It is necessary to visit it only when the counter goes from $n-2$ to $n-1$, that is, only when the counter indicates that all literals but one are set to false, and that now the clause is potentially unit. Complexity wise, If the instance has m clauses and n variables, and on average each clause has l literals, then whenever a variable gets assigned, on the average $l \cdot m/n$ counters need to be updated. On backtracking from a conflict, we need to undo the counter assignments for the variables unassigned during the backtracking. Each undo for a variable assignment will also need to update $l \cdot m/n$ counters on average. Moreover, SAT solvers utilizing learning mechanisms such as the ones discussed in section 4.5.3 usually generate very large clauses which augment the inefficiency of counter-based BCP engines.

The authors of the SATO solver [Zha97] were the first to propose an improvement to the BCP engine using *lazy* data structures. Their solution in-

volved the maintenance of two pointers, called *head* and *tail*, for each clause. Their method relative to counter-based engines, improved on the complexity in one direction- when assigning truth values to variables. However it had the same complexity when undoing variable assignments. The authors of Chaff use a similar method called the *two literal scheme* or the *watched literals* scheme, which also improves the complexity of undoing variable assignments. To be aware of a transition from $n-2$ to $n-1$ false literals in a clause, two literals not assigned false are watched at any given time. Until one of these literals is assigned false, there cannot be more than $n-2$ literals assigned false. Therefore we would need to modify the status of only those clauses that have one of the watched variables assigned false. In particular every variable v has two lists $pos\_watched(v)$ and $neg\_watched(v)$ containing pointers to all the literals corresponding to it in either phase. When a variable v gets assigned only one of the lists will be accessed. For example if v is assigned true then only $neg\_watched(v)$ will need to be accessed since the only possibility of a clause becoming unit is when some literal becomes false. For each watched literal pointed to by the list of pointers, the clause it belongs to is inspected by searching for false literals, where one of four conditions must hold.

- The other watched literal is assigned true, then we do nothing, the clause is satisfied.
- All the literals in the clause are assigned false, then the clause is a conflicting clause.
- All the literals in the clause are assigned false except for the other watched literal which is free, then the clause is detected as unit, with the other watched literal being the unit literal, and the algorithm proceeds with the BCP process as normal.
- There are at least two literals that are not assigned false, including the other watched literal. This means that the clause cannot be unit, and that there is at least one non-watched literal that is not assigned false. Choose this literal to by the new watched literal, i.e. replace it with the one that was assigned false, and by that maintain the property of having two watched literals that are not assigned false. Let this operation be called "moving the watched literal".

Figure 4.13 illustrates the *watched literals* technique. It shows how the watched literals for a single clause change under a series of assignments and unassignments. The advantages of this technique is that each variable keeps a reduced set of clause references for which the variable can be used to declare the clause as satisfied unsatisfied or unit. Thus less clauses will be visited and less status change operations will be needed. The main advantage of this technique

is however, that at the time of backtracking there is no need to modify the watched literals, and according to the authors, unassigning a variable can be done in constant time. The authors also claim that reassigning a variable that was already been assigned or unassigned will tend to be faster because it is more likely to reside in cache than before.

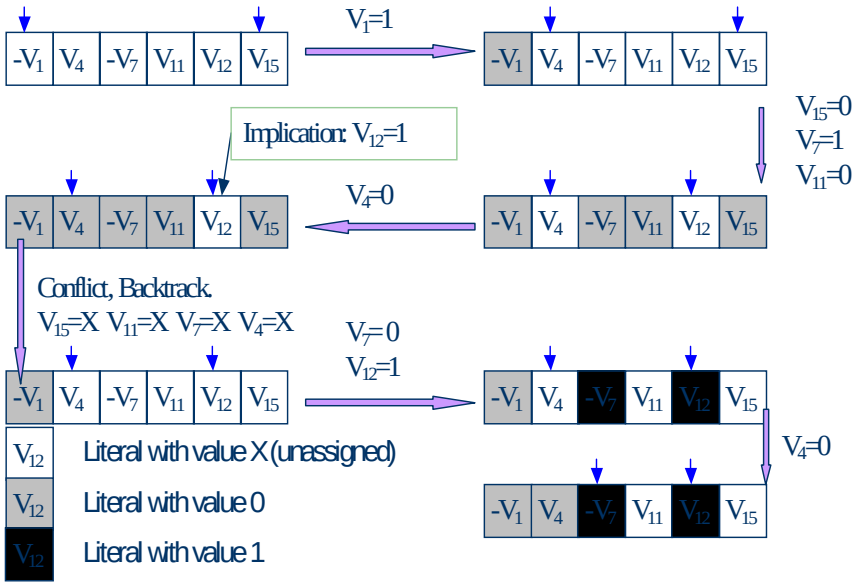


Figure 4.13: BCP using the two watched literal scheme (taken from[MMZ+01]).

As mentioned earlier rather than using or developing complex branching heuristics which require more run time, chaff uses a simple heuristic with low overhead, tailored for the domain of SAT instances on which chaff was tested, namely-EDA. Their heuristic termed *Variable State Independent Decaying Sum* (VSIDS) belongs to the family of *literal count heuristics* proposed in [MS99]. Literal count heuristics count the number of unresolved clauses in which a given variable appears in either phase. *Dynamic largest combined sum* (DLIS) also called MAXO, used in GRASP is an example. These heuristics however are state-dependent in the sense that different variable assignments yield different counts. This is because whether a clause is unresolved depends on the current variable assignment. Due to their dependence on variable state, each time a branching variable needs to be selected, the count for all free variables must be recalculated. VSIDS on the other hand is state-independent and therefore cheap to maintain. The literal counts are calculated as follows:

1. The counts for each variable and its polarity are initialized to 0.

2. When a new clause is learned and added to the clause database, the counter for each of its literals is incremented.

3. The literal with the highest count is chosen to be the next branching literal.

4. Ties are broken randomly.

5. Periodically, all the counters are divided by a constant.

Since the counts are updated only when there are conflicts and new clauses are learned, this heuristic has very low overhead. VSIDS can be viewed as adopting the "satisfaction hypothesis" strategy for choosing the next branching literal, restricted to attempts on satisfying the recently learned clauses. Step 5 ensures that the heuristic favors information generated by recently learned clauses.

The time required by complete search methods to solve similar instances, for example instances taken from the domain of EDA, can be surprisingly variable. Moreover, even for the same problem instance the run times may be considerably different. This could happen due to some embedded randomness, such as breaking ties randomly when two variables have the same heuristic measure, or due to different numberings of the same variables. This unpredictability in running times of complete algorithms undermines the main reason for preferring complete methods over incomplete ones, namely the desire to guarantee a solution. The unpredictability in running times can often be explained by a phenomenon called a "heavy-tailed cost distribution" [GSK98, GSCK00]. This means that if we set up an experiment measuring running times of an algorithm applied on a random sample of problem instances belonging to the same domain, EDA for example, then we will notice that at any given time during the experiment there will be a non-negligible probability of hitting a problem that requires exponentially more time to solve than any that had been encountered before, see figure 4.14. This phenomena causes the mean solution time to increase with the length of the experiment and be infinite in the limit. A general method that can provably eliminate such a phenomenon, is to introduce some sort of controlled randomization in the form of *restarts*.

Restarts, a technique imported from local search methods, usually consist of terminating the search process if it appears to be "stuck" exploring fruitless regions, and restarting the search at the root. It is usually implemented by introducing a *cutoff* parameter, that limits the search to a specified number of backtracks. When the cutoff is reached, the algorithm is restarted. Restarts will not effect the completeness of the procedure if some basic bookkeeping ensures that the algorithm will not revisit any previously explored regions of the search space. As implemented in Chaff, a restart consists of clearing the

state of all assigned variables and proceeding as normal. If the clauses learned prior to restarts, are never erased, Chaff will be able to maintain its completeness. However, to avoid memory explosion some of the learned clauses must be deleted. Chaff learns one new clause per conflict using the *first UIP* learning scheme, which partitions the implication graph such that all variables assigned at a current decision level after the first UIP are put in the conflict side. To control the size of the clause database and avoid memory explosion Chaff uses *scheduled lazy clause deletion* where each new clause is examined to determine at what point in the future, if any, the clause should be deleted. The metric used is relevance, such that when more than n (typically $100 \leq n \leq 200$) literals in the clause will first become unassigned, the clause will be deleted. To settle the trade off between bounded learning and maintaining completeness, Chaff slowly increases the relevance parameter n with time. Another strategy that Chaff uses to maintain completeness, which is also used in GRASP, is by extending the restart period with each restart.

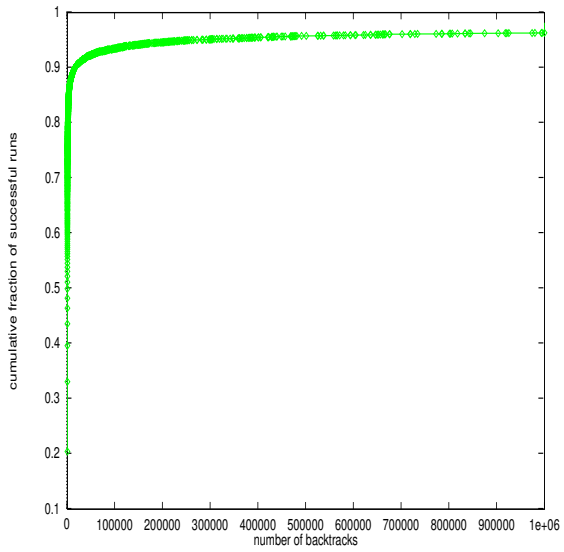


Figure 4.14: The cumulative fraction of successful runs as a function of the number of backtracks, by running a randomized sat procedure 10000 times on the same instance taken from the planning domain. In 80% of the runs a solution was found in 1000 or less backtracks. But 5% of the runs did not result in a solution even after 1000000 backtracks (taken from[GSCK00]).

4.7 Incomplete, Local Search Methods

Systematic, or global search SAT algorithms traverse a search space systematically to ensure that no part of it goes unexplored. They are complete in the sense that given enough running time, if a satisfying assignment exists they will find it; if no such assignment exists they are able to report or prove it. Alternative to systematic algorithms for SAT are stochastic local search algorithms, which are inspired by related techniques developed for constraint satisfaction problems, and were originally applied on the MAX-SAT problem, an optimization variant of SAT, where the problem is to find an assignment that satisfies as many clauses as possible.

Stochastic algorithms explore a search space randomly by making local perturbations to a working assignment without memory of where they have been. They are incomplete in the sense that they are not guaranteed to find a solution if one exists, and cannot cannot report or prove that no solution exists if they do not find one. In practice, if a formula is unsatisfiable they will always report that no solution exists, however may mistakenly report the no solution exist when in fact one does exist. Thus, local search methods are inappropriate for applications in which the goal is to prove unsatisfiability such as with theorem-provers. They are applied only when the goal is to find a satisfying assignments, which is why they are also referred to as *model-finders*. Nonetheless stochastic algorithms make up for their incompleteness by outperforming complete methods dramatically on satisfiable instances. In particular they are able to solve hard random 3SAT instances that complete methods are not able to solve, and are able to solve real-world instances of much larger size, both with a high success rate.

Local search SAT procedures can be characterized by a cost function defined over truth assignments such that the global minima corresponds to a satisfying truth assignment. A typical local search procedure will start with a random truth assignment as a potential solution, and then try to improve on it incrementally by searching for a lower cost assignment within a Hamming distance one neighborhood, that is by "flipping" the truth value of one of the variables. The flip is typically made according to some randomized greedy heuristic strategy with the aim of maximizing the number of satisfied clauses or minimizing the number of unsatisfied clauses, by the flip. Figure 4.15 gives the outline of a typical local search procedure. The procedure starts with a random truth assignment and changes the assignment by flipping the truth value of one of the variables, according to heuristic strategy captured by the *select_variable* function. Different SAT local search procedures are realized by different *select_variable* functions, that is, by the different strategies they employ to select the next variable to flip. Flips are repeated until either a sat-

isfying assignment is found or a maximum number of flips, set by the *MaxFlips* parameter, is reached. In the later case the process is restarted and repeated up to a maximum number of tries set by the *MaxTries* parameter.

```
procedure SAT-Local-Search(CNF formula : φ)
    for i := 1 to MaxTries do
        T := a randomly generated truth assignment
        for i := 1 to MaxFlips do
            if T satisfies φ then return T
            V := select_variable(φ, T)
            T := T with variable V flipped
        end
    end
    return "no satisfying assignment found"
end.
```

Figure 4.15: A generic local search procedure for SAT.

Unfortunately, local search procedures suffer from a common pathology. The possibility of getting stuck on local minima or plateaus (see figure 4.16). A local minima is a valley in the state space landscape that is lower than each of its neighboring states, i.e. has lower cost, but higher than the global minima. Hill-climbing local search procedures that reach the vicinity of a local minima will be drawn downwards towards the valley, but will then be stuck with nowhere else to go. A plateau is a flat area in the state space landscape, that is, an area where the cost function does not vary. Hill-climbing local search procedures may not be able to find their way of a plateaus. There are two standard ways to overcome these problems and enable local search procedures escape local minima and plateaus. The first is by introducing random restarts and the second by introducing noise, usually both of these techniques are used together. It should also be noted that NP-hard minimization problems typically have an exponential number of local minima to get stuck on [RN03].

4.7.1 Random Restarts

Random restarts, discussed previously in section 4.6.3 in the context of complete systematic procedures, adopt the well known adage, "if at first you don't succeed, try again", that is, if you get stuck start all over repeating the same process until the goal is found. Local search procedures utilizing restarts are complete with probability approaching 1 at infinity, for the trivial reason that they will eventually generate the goal state, in our case a satisfying truth assignment, as the initial state. Needless to say, executing a procedure infinitely many times is infeasible. In practice random restarts are controlled by two parameters: *MaxTries* and *MaxFlips*. *MaxTries* controls the total number of

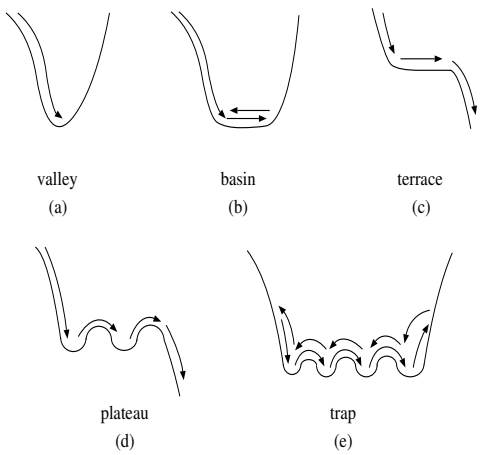


Figure 4.16: Different local minima structures, where traps are the hardest to deal with (taken from [GPFW97]).

restarts and can easily be calculated or optimized (expected number of restarts necessary to find the goal) if the probability of success, i.e., finding a goal state in each restart, is known, or can empirically be found. *MaxFlips* is a cutoff parameter that controls the frequency of restarts, that is, when is it that the procedure give up on local moves/flips and restarts with a new assignment. It is known that any search strategy can be improved, or at the very least not damaged by introducing random restarts and by choosing an appropriate cutoff value, *MaxFlips* in our case [GSK98]. Recently much attention was focused on the question of how to tune and optimize free parameters such as *MaxFlips* in local search procedures. For the general case research in this area has been largely empirical and it is still hard to predict the effects of a minor change in a procedure [SS01]. Nonetheless, estimating the optimal value of *MaxFlips* can be done efficiently by a probabilistic method [PW96].

Suppose we take a single instance φ and make N many runs of a local search procedure with $MaxFlips = M$. For each run we mark whether it was successful, i.e. a satisfying assignment was found, and the number of flips the run took to succeed. The goal is to make predictions for $MaxFlips = m < M$. Let S be the set of successful runs and S_m the set of runs that also took no more than m flips to succeed. An estimate for the probability that a try will succeed within m flips is:

$$p_m \approx |S_m|/N$$

Let $\overline{S_m}$ be the mean number of flips of the different runs in S_m. Therefore

we can say that with probability p_m that solution will be found on the first run using $\overline{S_m}$ flips. With probability $(1 - p_m)p_m$ the first run will fail but the second succeed, with $m + \overline{S_m}$ flips. Hence the expected number of flips for the given instance φ and bound m, which is what we are interested in, is:

$$E_{\varphi,m} = \sum_{k=0}^{\infty}(1 - p_m)^k p_m(km + \overline{S_m})$$

Which simplifies to:

$$E_{\varphi,m} = (1/p_m - 1)m + \overline{S_m}$$

By trying different values for m and calculating $E_{\varphi,m}$ we can find the optimal cutoff value for $MaxFlips$ by simply finding

$$m^* = arg_m \; min \; E_{\varphi,m}$$

For a collection D of instances from a certain domain we can extend these results and find the expected number of flips for D by:

$$E_m = \frac{1}{|D|}\sum_{\varphi \in D} E_{\varphi,m}$$

4.7.2 Noise

The characteristic of a search strategy that causes it to make moves that are non-optimal in the sense that the moves increase or fail to decrease the objective function, even when such improving moves are available in the local neighborhood, is called *noise*. As mentioned earlier, noise allows a local search procedures to escape from local minima and plateaus. It can also transform incomplete procedures to procedures that can a find a solution (if one exists) with probability approaching one as the run time approaches infinity [Hoo99]. Different procedures introduce noise in different forms, controlled by a noise parameter that determines the likelihood of escaping from local minima by making non-optimal moves. The performance of a stochastic local search procedure critically depends upon the setting of its noise parameter. Therefore finding the optimal noise parameter setting is crucial.

The optimal noise parameter setting depends both upon characteristics of the problem instances, and on the finegrained details of the search procedure, which may be influenced by other parameters. Finding it by trial and error methods requires considerable effort, which in most cases will be computationally prohibitive. In [MSK97] the authors suggest a general way of tunning local search procedures and quickly finding their optimal noise setting. In doing

so they discover two new search strategies that are discussed in section 4.8.5. Based on empirical evidence McAllester et al. observed that when the noise parameter is optimally tuned for a given search strategy, the *mean* value of the objective function (number of unsatisfied clauses) during a single run on a given instance is approximately the same across search strategies, see figure 4.17. They call this phenomena the *noise level invariant*, and show how it can be used to tune local search procedures: once the the mean value of the objective function giving the optimal performance of some single strategy over a given distribution of problems is determined, we can tune other strategies by simply finding the point at which their mean value of the objective function is the same as original strategy, in knowledge that this will give us close to optimal performance.

McAllester et al. also discovered an even more general principle than the *noise level invariant*. In order to use the noise level invariant, one needs to be able to gather statistics of at least one strategy across a sample of a given problem distribution. Some of the instances belonging to that distribution may be extremely hard to solve. Therefore, it is desirable to find a way of quickly predicting the setting of the noise parameter for a single problem instance, without actually having to solve it. They propose making many short runs of a given search procedure, at different noise settings, and recording the mean and variance of the objective function, without actually solving the problem. At low noise levels the mean is most likely small, that is, states with low numbers of unsatisfied clauses are reached. However, the variance is also very small, and the algorithm seldom reaches a state with zero unsatisfied clauses. When this occurs, the algorithm is stuck in a deep local minima. On the other hand, at high noise levels, the variance is large, but the average number of unsatisfied clauses is even larger. Once again, the algorithm is unlikely to reach a state with zero unsatisfied clauses. Therefore, a proper balance between the mean and variance needs to be found. Based on their observations they suggest that the optimal performance of a search strategy is obtained when the ratio of the mean to the variance is slightly above its minimum, and call this observation the *optimality invariant*.

4.8 Noise Strategies and Algorithms

4.8.1 GSAT

GSAT [SLM92] was the first successful SAT local search algorithm. GSAT uses a greedy strategy to choose next variable to flip, hence its name. It flips the variable that leads to the greatest decrease in the number of unsatisfied clause,

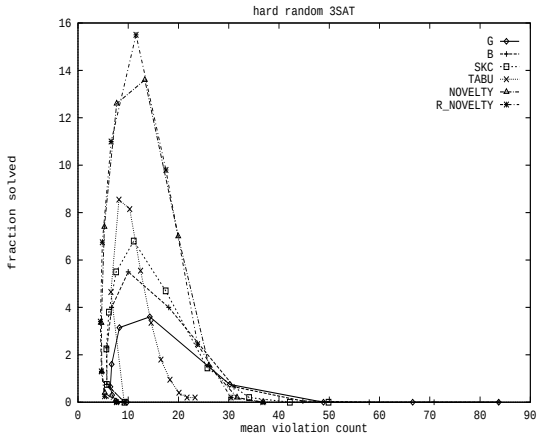


Figure 4.17: Strategy invariance of normalized noise level on random hard formulas (taken from [MSK97]).

and therefore the greatest increase the number of satisfied clauses, where ties are broken randomly. In departure from the standard approaches at the time used to avoid local minima and plateaus-terminating the search, GSAT continues to flip variables even when non-improving moves are available. These moves are called "sideways" moves. It was shown that if restricted to improving steps only, GSAT performs very poorly, and that in practice most of the search is dominated by sideways moves, except during an initial descent phase. Even using sideways moves GSAT may still get stuck on local minima and plateaus, for which the only way out is to make moves that increase the objective function, called "uphill" moves. This observation lead to the introduction of noise strategies, and "uphill" moves not performed by GSAT, into newer variable selection procedures. GSAT variable selection procedure is realized in figure 4.18.

4.8.2 Simulated Annealing

Simulated annealing is a general technique that originates from the theory of statistical mechanics used to solve hard combinatorial optimization problems based on an analogy with annealing of solids. In metallurgy, annealing is the process used to tamper or harden, metals and glass by heating them to a high temperature and then gradually cooling them, thus allowing the material to coalesce into a low energy crystalline state. Care must be taken however not to cool the solid too fast in order to avoid reaching an undesirable final state.

procedure *select_variable_GSAT*$(CNF\ formula : \phi, truth\ assignment : T)$
    for each variable v in ϕ **do**
        $T' :=$ truth assignment got by flipping v in T
        $E_1 :=$ number of unsatisfied clauses in ϕ given T
        $E_2 :=$ number of unsatisfied clauses in ϕ given T'
        $\Delta E(v) := E_2 - E_1$
    end
    return $min\ arg_v\ \{\Delta E(v) |\ \Delta E(v) \leq 0\}$ or nothing if no such v exists
end.

Figure 4.18: GSAT's variable selection strategy

Similarly, with combinatorial optimization problems we do not want to accept only configurations that lower the cost function or decrease the cost function too rapidly. At higher temperatures we should allow for more "bad" moves, developing the gross features of the eventual state, while at lower temperatures the fine details are developed where bad moves should be less likely.

The performance of a SAT local search procedure is determined by its ability to escape or to move between successive local minimas and plateaus. The search fails when the procedure can find no way off a plateau or local minima, either because such transitions are rare or nonexistent. When this occurs, the search is usually restarted with a new random assignment. As mentioned earlier the other common way of escaping from local minima, is based on occasionally making "bad" uphill moves, called noise. Simulated annealing [KGV83] is one of the first mechanisms used to introduce noise into the search. The amount of noise is controlled by a parameter called "temperature", that controls the probability of making uphill moves.

Simulated annealing comes in many forms and flavors, prominent among them is the one based on the *Metropolis* procedure (1953) from statistical mechanics which was adopted as a combinatorial optimization technique by Kirkpatrick in [KGV83]. Its implementation as a SAT local search procedure is as follows [SKC93]: Start with a random truth assignment. Repeatedly pick a random variable v, and compute $\Delta E(v)$, the change in the number of unsatisfied clauses, as in GSAT. If $\Delta E(v) \leq 0$, then flip the truth value of v since it is a downhill improving or a sideways move. Otherwise make an uphill "bad" move by flipping v with probability $e^{-\Delta E(v)/T}$. T is the noise parameter called *temperature*, that may either be held constant or slowly decreased from high to near zero according to a cooling schedule. The probability of making a "bad" move decreases exponentially with the "badness" of the move captured by $\Delta E(v)$, and as the temperature decreases. Thus, at higher temperatures bad moves are more likely then at lower temperatures. Simulated annealing variable selection procedure is realized in figure 4.19. Note that GSAT can be

realized by simulated annealing if the temperature is kept zero, that is, the probability of making a bad move approaches zero. The main problem with simulated annealing is that in general it may require much more flips than the other SAT local search procedures.

procedure *select_variable_SM*$(CNF\ formula : \phi, truth\ assignment : T)$
    $v :=$ randomly choose a variable
    $T' :=$ truth assignment got by flipping v in T
    $E_1 :=$ number of unsatisfied clauses in ϕ given T
    $E_2 :=$ number of unsatisfied clauses in ϕ given T'
    $\Delta E(v) := E_2 - E_1$
    if $\Delta E(v) \leq 0$ **then return** v
    else if $rnd(0,1) \leq e^{-\Delta E(v)/T}$ **then return** v
    else return nothing
end.

Figure 4.19: Simulated Annealing variable selection strategy

4.8.3 Random Walk, WSAT

A random walk is generally thought of as process consisting of a sequence of changes each of whose characteristics is determined haphazardly. In [SKC93] the authors introduce several extensions to the GSAT procedure by mixing a random walk strategy with a greedy strategy. In simulated annealing, the random uphill move depends on the probability $e^{-\Delta E(v)/T}$, where v could be any variable as long as $\Delta E(v) > 0$. With random walk, however, random moves are closely related to the variables that appear in some unsatisfied clause. Thus, by flipping the truth assignment of a single arbitrary variable in that clause, such a random walk will at least fix one clause. This random walk strategy is called GSAT-RW and proceeds as follows:

> *With probability p, pick a variable occurring in some unsatisfied clause and flip its truth assignment.*

> *With probability* $1-p$*, follow the standard GSAT scheme, i.e., make the best possible local move.*

Where p is the noise parameter, with optimal value equal to 0.5. The main problem with procedures that incorporate a greedy strategy, like GSAT and GSAT-RW is that they have to evaluate $\Delta E(v)$ for each variable v that appears in the formula, which is a time consuming operation. Another extension to GSAT proposed by the same authors in [SKC94] called WSAT or Walk-SAT,

addresses this problem by examining only a subset of the variables, exhibiting improved performance. WSAT employs a two step random process by first randomly picking an unsatisfied clause, and then picking, either at random (with probability p) or according to a greedy strategy, a variable within that clause to flip. Thus, GSAT-RW can be viewed as adding random walk to a greedy procedure, while WSAT can be viewed as adding a greedy strategy to a random walk.

procedure $select\_variable\_WSAT(CNF\ formula : \phi, truth\ assignment : T)$
    $C :=$ randomly select an unsatisfied clause
    for each variable v in C **do**
        $T' :=$ truth assignment got by flipping v in T
        $E_1 :=$ number of unsatisfied clauses in ϕ given T
        $E_2 :=$ number of unsatisfied clauses in ϕ given T'
        $\Delta E(v) := E_2 - E_1$
    end
    if $rnd(0,1) \leq p$ **then**
        return a randomly chosen variable in C
    else
        return $min\ arg_v\ \{\Delta E(v)\}$
end.

Figure 4.20: WSAT variable selection strategy

4.8.4 TSAT/Tabu-SAT

The basic concept of "Tabu Search" (Glover, 1986) is the existence of a meta-heuristic superimposed on another heuristic, to avoid entrapment of the search in cycles by forbidding or penalizing moves which take the solution, in the next iteration, to points in the search space previously visited (hence "tabu"). That is, prevent a previous move from being repeated and by that ensure that new regions of the search space will be explored. To avoid retracing the steps used, tabu search records recent moves in one or more *tabu lists*.

Tabu search can be incorporated into any of the existing SAT local search procedures as follows: fist, pick a candidate variable according to the original selection rule, such as a variable that minimizes the number of unsatisfied clauses when flipped. Before flipping it however, make sure that the variable has not been flipped in the last t steps. This is done by maintaining a length t FIFO list of flipped variables called the *tabu list*, that prevents any variables appearing the list from being flipped again during a given amount of time (t steps). If a candidate variable appears in the list it is dropped and another variable must be chosen. Each time a new variable is flipped the list is updated.

In addition to other parameters, t the length of the tabu list is also a noise parameter, and like other noise parameters needs to be optimized. In [MSG97] the authors found by extensive experimentations and by the use of regression, a linear model capturing the relationship between the the number of variables in a formula n, and the optimal length of the tabu list.

$$t = 0.01875n + 2.8125$$

4.8.5 Novelty/R_Novelty

As mentioned earlier Novelty and R_Novelty are two new heuristics resulting from the findings reported in [MSK97], that allows one to quickly find the optimal noise setting. They outperform other variations of WSAT, and proceeds as follows:

> **Novelty**: *Pick a random clause C. Flip variable v in C that would result in the smallest number of unsatisfied clauses, unless is it the most recently flipped variable in C. If it is the most recent, then flip it with probability $1 - p$, otherwise flip variable v' in C that results in the second smallest number of unsatisfied clauses.*
>
> **R_Novelty**: *Like Novelty, but in the case where the best variable is the most recently flipped one, the decision between the best and second-best variable probabilistically depends on their score difference n, where $n \geq 1$.*
>
> 1. *when $p < 0.5$ and $n > 1$, pick the best.*
> 2. *when $p < 0.5$ and $n = 1$, then with probability $2p$ pick the second-best, otherwise pick the best.*
> 3. *when $p \geq 0.5$ and $n = 1$, pick the second-best.*
> 4. *when $p \geq 0.5$ and $n > 1$, then with probability $2(p - 0.5)$ pick the second-best, otherwise pick the best.*
>
> *Additionally, every 100 flips, instead of using this heuristic, the variable to be flipped is randomly picked from the selected clause.*

The intuition behind Novelty is that one wants to avoid repeatedly flipping the same variable back and forth. The intuition behind R_Novelty is that the objective function should influence the choice between the best and second-best, where a large difference in the objective function favors the best.

Chapter 5

Performance Evaluation

The performance of SAT procedures can be determined *experimentally* or *analytically*. The analytical evaluation of SAT algorithms is primarily based on worst-case analysis (the theory of NP-completeness). This sort of analysis is by its nature a very pessimistic analysis, which often does not reflect or explain the behavior of the algorithms in practice. Since the typical performance of SAT algorithms is much better than any proven worst-case result, SAT algorithms are either evaluated experimentally or by a different form of analysis which is based on probabilistic measures.

Probabilistic studies require a-priori knowledge of the input models or probability distributions on the input formulas to the algorithm. The two most widely used probabilistic measures of performance are *average-time/case* complexity and *probabilistic-time* complexity. Average-time is a weighted average of the time, or some other measure such as the size of the search tree, needed in order to solve a given set of formulas sampled from a fixed known distribution. In probabilistic-time studies on the other hand, the algorithm is given a termination deadline, usually specified as a polynomial in the size of the input, and the performance is measured by the fraction of formulas that are solved within the deadline. Probabilistic-time studies are in most cases used to measure the performance of incomplete algorithms, since average-time cannot be defined for them. When the probability distributions of the input formulas cannot be determined in advance, which unfortunately is the case for the class of real-world problems (discussed in section 5.1.2), the only way to evaluate SAT algorithms is experimentally.

Experimental studies, however, due to the space of likely formulas are forced to consider a relatively small number of input models, thus are inconclusive. Analytical studies on the other hand, aim at broad range of input models, where each model typically represents a class of formulas of a particular size.

Nonetheless they too have a drawback, which is that only the simplest algorithms can be analyzed. To compensate for this, several features of a complex algorithm can be removed, leaving a more analyzable one. The analytical result on the simplified algorithm often provides enough insight on the behavior of the more complex algorithm. A side benefit to this approach is that the study can suggest which of the removed components should be kept and which discarded.

5.1 Input Models

There are two main classes of input models for evaluating the performance of SAT procedures: *practical* problems, which are encodings of real world problems; and *randomly generated* problems, which can be very useful benchmark problems, but have no known practical application. Random models are better suited for analytical studies and to some extent also for experimental studies, the reasons are outlined in section 5.1.1. Moreover, the advocates of random input models argue that if one has to solve real-world instances of varying types, without having any particular information on the origin of these instances, an efficient algorithm for random input models it most likely to perform on average at least as good as any specific algorithm tailored to solve a particular real-world problem. The reason is that the randomly generated problems represents a "core" of hard problems. Once the structure of a real-world instance is exploited and "squeezed out" by an algorithm tailored to solve and exploit the properties of that type of instance, the remaining of the problem will be a random problem[CA96]. Nonetheless experiments strongly suggest that there is little correlation between the performance of a SAT procedure tested on random input models and the performance of the same algorithm tested on practical problems [GPFW97].

5.1.1 Random Models

There are two main classes of random models that have been subject to significant analytical study: the *random-clause-length* and the *fixed-clause-length*.

Random-clause-length models, also called *fixed-density*, are based on the probabilistic model $M_1(n, m, p)$. Instances of this model are generated according to three parameters, where n is the number of variables, m the number of clauses, and a probability $0 < p \leq 0.5$. Additionally p, m may be functions of n. The m clauses are constructed independently as follows: in each clause each of the n variables occurs positively with probability p and negatively with probability p. Thus both positively and negatively with probability p^2, not at

all with probability $(1-p)^2$, and where the expected number of literals in a clause is $l = 2np$.

fixed-clause-length models are based on the probabilistic model $M_2(n, m, k)$, where m and n are the same as before, and k is the number of literals per clause. Instances of M_2 are generated by selecting uniformly m clauses at random from the set of all possible clauses of length k. Alternatively instances of M_2 can be viewed as generated as follows: each clause is produced by randomly choosing a set of k variables from the set of n variables, and negating each with probability 0.5, thus the set of all possible clauses contains $2^k \binom{n}{k}$ different clauses. Instances generated according to this model are also referred to as *random k-SAT* instances.

The main impetus for the average case analysis of SAT algorithms was given by Goldberg [Gol79]. He proposed a variant of the Random-clause-length model M_1, and suggested that SAT might not be that hard on average. He showed that for any value of p DPLL solves instances generated from that model in polynomial time on average. This result was put in perspective by Franco and Paull in 1983, who showed that Goldberg's result was a direct consequence of a favorable choice of distribution (from the space of all possible instances almost no hard ones were sampled), and thus was overly optimistic. They showed that the formulas he used were so easily satisfiable that an algorithm which simply tried randomly generated assignments would, with probability 1, find a satisfying assignment with a constant amount of trials [Fra86]. They also proposed the fixed-clause-length as an input model, and studied the performance of DPLL on instances generated by this model. They found that the fixed-clause-length instances took exponential time on average if DPLL were to find all satisfying solutions and not stop at the first one found. This result lead to the hypothesis, which was confirmed later, that the fixed-clause-length input model is more appropriate for evaluating SAT procedures. Although the majority of random k-SAT instances are easy to solve, an empirical study made by Mitchell et al. [MSL92] gave birth to a technique used to generate random k-SAT instances that are hard to solve on average.

By inspecting the relationship between ratios of clauses to variables $c = m/n$, and the time required by DPLL to solve a random k-SAT instance, Mitchell et al. showed that the space of random k-SAT instances can be divided into three regions, figure 5.1. For instances that are relatively short (small values of c) or instances that are relatively long (large values of c), DPLL finishes quickly, but for instances of medium relative length, DPLL takes much longer. The reason is that short instances with few clauses are *under-constrained* , that is, have many satisfying assignments which are likely to be found earlier in the search. Long instances with many clauses are over-constrained, and usually unsatisfiable, so contradictions are found earlier, and the whole search can be

completed quickly. Finally, instances in between are much harder because they have relatively few (if any) satisfying assignments, and contradictions are discovered only after assigning many variables, resulting a deep search tree. To further understand the hardness of random k-SAT instances, the relationship between the ratio c and the likelihood of satisfiability was examined. In the under-constrained region the probability of a satisfying instance was 1 or very close to 1. In the over-constrained region the probability of a satisfying instance was 0 or very close to 0. In between, a region now called the *transition region*, the probability smoothly shifted form near 1 to near 0. The intriguing discovery in this experiment was that the peak in difficulty occurs near the ratio where about 50% of the formulas are satisfiable, i.e., the probability of a formula being satisfiable is 0.5. Moreover the 50% point, known as the *crossover point*, seemed to occur at a fixed ratio of clauses to variables, when the number of clauses is about 4.3 times the number of variables. Similar patterns of hardness were discovered by others, using substantially different algorithms, and for different values of k [ML96], which lead to the conjecture that this pattern will hold for all reasonable complete methods, and thus is a suitable model for evaluating SAT procedures.

The easy-hard-easy pattern and satisfiable-to-unsatisfiable transition was also observed for random-clause-length instances, when p was made a function of n such that the expected clause length remains constant as n is increased [MSL92, ML96]. However the peak in difficulty at the transition region was much less dramatic, and instances belonging to this region were generally found much easier for complete algorithms to solve than the similar fixed-clause-length instances. Moreover, large instances generated by this model, were found to have quite frequently, several empty and unit clauses which resulted either trivially unsatisfiable instances or easily satisfiable instance. These findings lead a belief still held at present, that random k-SAT instances from the hard region are better for evaluating the performance SAT procedures.

It should also be noted that the hard-easy distributions of SAT instances depend not only on the properties of the input models, but also on the algorithms used to solve these instances. However, as mentioned earlier, random k-SAT instances from the hard region appear to be algorithm independent because they are a consequence of the expected number of satisfying truth assignments.

5.1.2 Practical Models

Practical input models are ultimately the most important for researches in the area of applications. Random models, as mentioned above are better suited for analytical studies since they possess properties that can be exploited for anal-

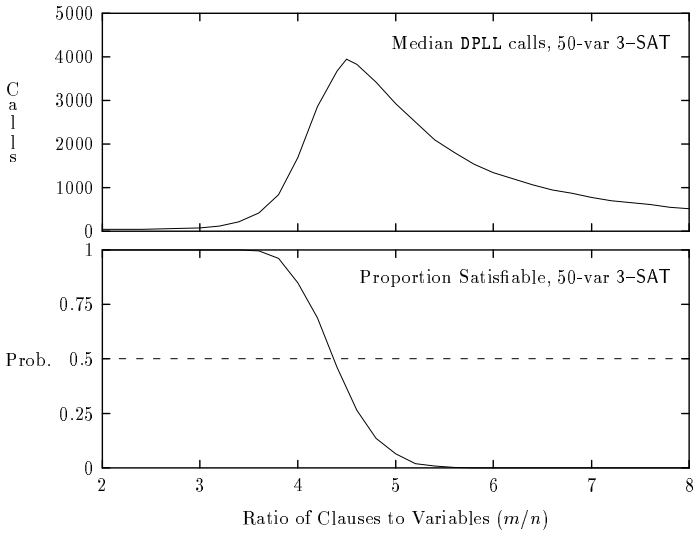


Figure 5.1: Phase transition in random 3-SAT: DPLL performance and probability of being satisfiable(taken from [CM97]).

ysis. However, they have very little resemblance to actual real-world instances. Practical instances on the other hand, also have structure, but structure which is much harder to base a general theory on, such as for the random input models. The phase transition phenomena, for example, has not yet been observed in practical input models.

Practical input models can be divided into two main classes. The first class contains SAT-encoded, randomly generated instances, such as graph coloring problems, n-queens problems, and pigeon-hole problems. The second class of problems is derived from real-world applications of SAT from various domains, such as problems from the planning and logistics domains, and problems from the Electronic Design Automation (EDA) domain. The advantage of problems from the first group is that they can easily be generated and their structure easily controlled, whereas problems from the other group are harder to obtain and analyze. Therefore more research is done on the first type of problems in hope of better understanding the performance of SAT procedures when applied to problems from the second group.

In recent years many problem instances from the EDA domain have been created. They are in general much larger, and often unsatisfiable, thus SAT procedures solving them are required to be complete. The hardest among them are instances of Bounded Model Checking, Combinatorial Equivalence Checking,

and Processor Verification, which most state-of-the-art SAT solvers are unable to solve. Moreover EDA and circuit design companies are known for having evaluated solvers with even harder instances. Although problem size is not indicative of hardness, their benchmarking instances exceed 300000 variables, 700000 clauses, and 1500000 literals, which current SAT solvers are unable to solve [MS00]. Given the interest of the EDA community, SAT solvers targeting real-world instances ought to prove effective for some of these problems.

5.2 The Crossover Point

In recent years there has been a growing interest in finding the location of the crossover point, and the boundaries of the transition region. The range of ratios (clauses to variables) over which this transition has been observed, becomes smaller as the number of variables is increased. This coupled with the occurrence of a transition region phenomena in other random combinatorial problems lead to the *threshold conjecture* that states: For each k, there is some c', such that for each fixed value of $c < c'$, random k-SAT with n variables and cn clauses is satisfiable with probability tending to 1 as $n \to \infty$, and when $c > c'$, unsatisfiable with probability tending to 1. For random 2-SAT this has been proved to be true when $c' = 1$. For random 3-SAT, many models were proposed as estimators of the crossover point, such as that the crossover point occurs when $m = 4.24n + 6.21$ [CA93], or when $m = 4.258n + 58.26n^{-2/3}$ [CA96]. The provable bounds on the location of the crossover point are $3.003 < c < 4.598$ [CM97], where both the upper and lower bound are gradually improved by various researches. The original 5.19 upper bound can easily be proved and is based on the independence assumption that all clauses are independent, i.e. have no variables in common. Let p be the probability that a random 3-SAT instance with n variables and m clauses is satisfiable by a random truth assignment v. Then, since each clause is a disjunction of three different variables, the probability that a clause is satisfied by v is 7/8. Since all clauses are independent $p = (\frac{7}{8})^m$, and since there are 2^n different truth assignments, the expected number of satisfying assignments is therefore $N = 2^n(\frac{7}{8})^m$. The change in phase occurs when the expected number of solutions goes from more than one to less than one, therefore substituting $N = 1$ we get $c = m/n = log_{\frac{8}{7}} 2 = 5.19$. Note however that if we employ the independence assumption, that is, all clauses are independent of each other, then every 3-SAT instance would be satisfiable.

Bibliography

[APT79] B. Aspvall, M.F. Plass, and R. Tarjan, *A linear-time algorithm for testing the truth of certain quantified boolean formulas*, Information Processing Letters (1979), 8(3):121–123.

[BAH+94] A. Beringer, G. Aschemann, H. Hoos, M. Metzger, and A. Weiss, *Gsat versus simulated annealing*, Proceedings of the European Conference on Artificial Intelligence (A.G. Cohn, ed.), John Wiley Sons, 1994, pp. 130–134.

[BS97] Roberto J. Jr. Bayardo and Robert C. Schrag, *Using CSP look-back techniques to solve real-world SAT instances*, Proceedings of the Fourteenth National Conference on Artificial Intelligence (AAAI'97) (Providence, Rhode Island), 1997, pp. 203–208.

[CA93] J. M. Crawford and L. D. Auton, *Experimental results on the crossover point in satisfiability problems*, Proceedings of the Eleventh National Conference on Artificial Intelligence (AAAI'93), 1993, pp. 21–27.

[CA96] James M. Crawford and Larry D. Auton, *Experimental results on the crossover point in random 3-SAT*, Artificial Intelligence **81** (1996), no. 1-2, 31–57.

[CLR90] T. Corman, C. Leiserson, and R. Rivest, *Introduction to Algorithms*, McGraw Hill, New York, 1990.

[CM97] Stephen A. Cook and David G Mitchell, *Finding hard instances of the satisfiability problem: A survey*, Satisfiability Problem: Theory and Applications (Du, Gu, and Pardalos, eds.), vol. 35, American Mathematical Society, DIMACS, 1997, pp. 1–17.

[Coo71] Stephen A. Cook, *The complexity of theorem-proving procedures*, In Conference Record of Third Annual ACM Symposium on Theory of Computing (1971), pages 151–158.

[Coo00] Stephen Cook, *The P versus NP problem*, Manuscript prepared for the Clay Mathematics Institute for the Millennium Prize Problems, April 2000.

[DABC93] Olivier Dubois, Pascal André, Yacine Boufkhad, and Jacques Carlier, *SAT versus UNSAT*, in Johnson and Trick [JT93], pp. 415–436.

[DLL62] M. Davis, G. Logemann, and D. Loveland, *A machine program for theorem-proving*, Communications of ACM (1962), 5:394–397.

[DP60] M. Davis and H. Putnam, *A computing procedure for quantification theory*, Journal of ACM Symposium on Theory of Computing (1960), 7:201–215.

[DR94] R. Dechter and I. Rish, *Directional resolution:the davis-putnam procedure, revisited*, In Proc KR-94 (1994), 134–145.

[DSW94] Martin D. Davis, Ron Sigal, and Elaine J. Weyuker, *Computability, complexity, and languages: Fundamentals of theoretical computer science*, second ed., Academic Press, San Diego, 1994.

[EIS76] S. Even, A. Itai, and A. Shamir, *On the complexity of timetable and multi-commodity flow problems*, SIAM Journal of Computing (1976), 5(4).

[FH03] L. Fortnow and S. Homer, *A short history of computational complexity*, The History of Mathematical Logic (D. van Dalen, J. Dawson, and A. Kanamori, eds.), North-Holland, Amsterdam, 2003, To appear.

[Fra86] John V. Franco, *On the probabilistic performance of algorithms for the satisfiability problem*, Information Processing Letters **23** (1986), no. 2, 103–106.

[Fre95] Jon W. Freeman, *Improvements to propositional satisfiability search algorithms*, Ph.D. thesis, Department of Computer Science, University of Pennsylvania, Philadelphia, 1995.

[GJ79] M.R. Garey and D.S. Johnson, *Computers and intractability, a guide to the theory of np-completeness*, W.H.Freeman and Co., San Francisco, 1979.

[Gol79] A Goldberg, *On the complexity of the satisfiability problem*, Courant Computer Science Report No. 16, 1979.

[GPFW97] J. Gu, P. Purdom, J. Franco, and B. Wah, *Algorithms for the satisfiability (SAT) problem: A survey*, Satisfiability Problem: Theory and Applications (Du, Gu, and Pardalos, eds.), vol. 35, American Mathematical Society, DIMACS, 1997, pp. 19–151.

[GSCK00] Carla P. Gomes, Bart Selman, Nuno Crato, and Henry A. Kautz, *Heavy-tailed phenomena in satisfiability and constraint satisfaction problems*, Journal of Automated Reasoning **24** (2000), no. 1/2, 67–100.

[GSK98] Carla P. Gomes, Bart Selman, and Henry Kautz, *Boosting combinatorial search through randomization*, Proceedings of the Fifteenth National Conference on Artificial Intelligence (AAAI'98) (Madison, Wisconsin), 1998, pp. 431–437.

[Hay98] B. Hayes, *Can't Get No Satisfaction*, American Scientist **85** (1998), no. 2, 108–112.

[HE80] R. M. Haralick and G. L. Elliot, *Increasing tree search efficiency for constraint satisfaction problems*, Artificial Intelligence (1980), no. 14, 263–313.

[Hoo99] Holger H. Hoos, *On the run-time behaviour of stochastic local search algorithms for SAT*, Proceedings of the Sixteenth National Conference on Artificial Intelligence (AAAI'99) (Orlando, Florida), 1999, pp. 661–666.

[HS] Holger H. Hoos and Thomas Stützle, *SATLIB: An Online Resource for Research on SAT*, pp. 283–292.

[HV95] John N. Hooker and V. Vinay, *Branching rules for satisfiability*, Journal of Automated Reasoning **15** (1995), no. 3, 359–383.

[HW87] Robert M. Haralick and Sheau-Huei Wu, *An approximate linear time propagate and divide theorem prover for propositional logic*, International Journal of Pattern Recognition and Artificial Intelligence **1** (1987), no. 1, 141–155.

[JT93] D. S. Johnson and M. A. Trick (eds.), *Second DIMACS implementation challenge : cliques, coloring and satisfiability*, DIMACS Series in Discrete Mathematics and Theoretical Computer Science, vol. 26, American Mathematical Society, 1993.

[Kar72] Ricard M. Karp, *Reducibility among combinatorial problems*, Complexity of Computer Computations (K.E. Miller and J.W. Thatcher, eds.), Plenum Press, New York, 1972, pp. 85–103.

[KGV83] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, *Optimization by simulated annealing*, Science, Number 4598, 13 May 1983 **220, 4598** (1983), 671–680.

[KS94] S. Kirkpatrick and B. Selman, *Critical behavior in the satisfiability of random Boolean expressions*, Science **264** (1994), no. 5163, 1297–1301.

[KS01] Henry Kautz and Bart Selman (eds.), *Proceedings of the workshop on theory and applications of satisfiability testing (sat2001)*, vol. 9, Elsevier Science Publishers, June 2001, LICS 2001 Workshop on Theory and Applications of Satisfiability Testing (SAT 2001).

[KS03] ———, *Ten challenges redux: Recent progress in propositional reasoning and search*, Invited paper, Ninth International Conference on Principles and Practice of Constraint Programming (CP 2003), 2003.

[LA97a] C. M. Li and Anbulagan, *Look-Ahead Versus Look-Back for Satisfiability problems*, In Proceedings of the International Conference on Principles and Practice of Constraint Programming, October 1997, pp. 341–355.

[LA97b] Chu Min Li and Anbulagan, *Heuristics based on unit propagation for satisfiability problems*, IJCAI (1), 1997, pp. 366–371.

[LaPMS02] Inês Lynce and Jo ao P. Marques-Silva, *Building state-of-the-art sat solvers*, European Conference on Artificial Intelligence (ECAI'02), 2002.

[Le 01] Daniel Le Berre, *Exploiting the real power of unit propagation lookahead*, in Kautz and Selman [KS01], LICS 2001 Workshop on Theory and Applications of Satisfiability Testing (SAT 2001).

[LG03] Chu-Min Li and Sylvain Grard, *On the limit of branching rules for hard random unsatisfiable 3-SAT*, Discrete Applied Mathematics **130/2** (2003).

[Li99] Chu Min Li, *A constraint-based approach to narrow search trees for satisfiability*, Information Processing Letters **71** (1999), no. 1, 75–80.

[Lib00] Paolo Liberatore, *On the complexity of choosing the branching literal in dpll*, Artificial Intelligence **116** (2000), no. 1–2, 315–326.

[LMS02] Inês Lynce and J. P. Marques-Silva, *Efficient data structures for backtrack search sat solvers*, Fifth International Symposium on the Theory and Applications of Satisfiability Testing (SAT'02), may 2002.

[LMS03] ———, *An overview of backtrack search satisfiability algorithms*, Annals of Mathematics and Artificial Intelligence (2003), 37(3):307–326.

[Lyn01] Inês Lynce, *Algebraic simplification techniques for propositional satisfiability*, Master's thesis, Instituto Superior Técnico, Lisboa, Portugal, 2001.

[ML96] David G. Mitchell and Hector J. Levesque, *Some pitfalls for experimenters with random SAT*, Artificial Intelligence **81** (1996), no. 1-2, 111–125.

[MMZ+01] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik, *Chaff: Engineering an Efficient SAT Solver*, Proceedings of the 38th Design Automation Conference (DAC'01), 2001.

[MS99] João Marques-Silva, *The impact of branching heuristics in propositional satisfiability algorithms*, In Proceedings of the the 9th Portuguese Conference on Artificial Intelligence (EPIA), September 1999, pp. 62–74.

[MS00] J.P. Marques-Silva, *On selecting problem instances for evaluating satisfiability algorithms*, ECAI Workshop on Empirical Methods in Artificial Intelligence (ECAI-EMAI), August 2000.

[MSG97] Bertrand Mazure, Lakhdar Sas, and Eric Grgoire, *Tabu search for sat*, Proceedings of the Fourteenth National Conference on Artificial Intelligence and 9th Innovative Applications of Artificial Intelligence Conference (AAAI-97/IAAI-97) (Providence, Rhode Island), 1997, pp. 281–285.

[MSK97] David McAllester, Bart Selman, and Henry Kautz, *Evidence for invariants in local search*, Proceedings of the Fourteenth National Conference on Artificial Intelligence (AAAI'97) (Providence, Rhode Island), 1997, pp. 321–326.

[MSL92] David G. Mitchell, Bart Selman, and Hector J. Levesque, *Hard and easy distributions for SAT problems*, Proceedings of the Tenth

National Conference on Artificial Intelligence (Menlo Park, California) (Paul Rosenbloom and Peter Szolovits, eds.), AAAI Press, 1992, pp. 459–465.

[MSS96] Joao P. Marques-Silva and Karem A. Sakallah, *GRASP - A New Search Algorithm for Satisfiability*, Proceedings of IEEE/ACM International Conference on Computer-Aided Design, November 1996, pp. 220–227.

[MSS99] J.P. Marques-Silva and K.A. Sakallah, *Grasp: A search algorithm for propositional satisfiability*, IEEE Transactions on Computers **48** (1999), no. 5, 506–521.

[Pap94] C.H. Papadimitriou, *Computational Complexity*, Addison-Wesley, Mass, 1994.

[PS96] David M. Pennock and Quentin F. Stout, *Exploiting a theory of phase transitions in three-satisfiability problems*, AAAI/IAAI, Vol. 1, 1996, pp. 253–258.

[PW96] Andrew J. Parkes and Joachim P. Walser, *Tuning local search for satisfiability testing*, Proceedings of the Thirteenth National Conference on Artificial Intelligence (AAAI'96), 1996, pp. 356–362.

[RN03] Stuart Russell and Peter Norvig, *Artificial intelligence: A modern approach*, second ed., Prentice Hall, New Jersey, 2003.

[SKC93] Bart Selman, Henry A. Kautz, and Bram Cohen, *Local search strategies for satisfiability testing*, Proceedings of the Second DIMACS Challange on Cliques, Coloring, and Satisfiability (Providence RI) (Michael Trick and David Stifler Johnson, eds.), 1993.

[SKC94] ———, *Noise strategies for improving local search*, Proceedings of the Twelfth National Conference on Artificial Intelligence (AAAI'94) (Seattle), 1994, pp. 337–343.

[SKM97] Bart Selman, Henry A. Kautz, and David A. McAllester, *Ten challenges in propositional reasoning and search*, Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence (IJCAI'97), 1997, pp. 50–54.

[SLM92] Bart Selman, Hector J. Levesque, and D. Mitchell, *A new method for solving hard satisfiability problems*, Proceedings of the Tenth National Conference on Artificial Intelligence (Menlo Park, California) (Paul Rosenbloom and Peter Szolovits, eds.), AAAI Press, 1992, pp. 440–446.

[SS01] Dale Schuurmans and Finnegan Southey, *Local search characteristics of incomplete SAT procedures*, Artificial Intelligence **132** (2001), no. 2, 121–150.

[Sub] K. Subramani, *Optimal length tree-like resolution refutations for 2sat formulas*, ACM Transactions on Computational Logic (TOCL), To appear.

[Zha97] Hantao Zhang, *SATO: an efficient propositional prover*, Proceedings of the International Conference on Automated Deduction (CADE'97), volume 1249 of LNAI, 1997, pp. 272–275.

[ZM02] Lintao Zhang and Sharad Malik, *The Quest for Efficient Boolean Satisfiability Solvers*, Proceedings of 8th International Conference on Computer Aided Deduction(CADE 2002), 2002.

[ZMMM01] Lintao Zhang, Conor F. Madigan, Matthew W. Moskewicz, and Sharad Malik, *Efficient conflict driven learning in boolean satisfiability solver*, ICCAD, 2001, pp. 279–285.