6

SAT and Constraint Satisfaction

The Satisfiability Problem in Propositional Logic (SAT) is a conceptually simple combinatorial decision problem which plays a prominent role in Complexity Theory and Artificial Intelligence. To date, stochastic local search methods are among the most powerful and successful methods for solving large and hard instances of SAT. In this chapter, we first give a general introduction to SAT and motivate its relevance to various areas and applications. Next, we give an overview of some of the most prominent and best-performing classes of SLS algorithms for SAT, covering algorithms of the GSAT and WalkSAT Architecture as well as dynamic local search algorithms. We discuss important properties of these algorithms such as the PAC property and give a sketch of their empirical performance and behaviour.

Constraint Satisfaction Problems (CSPs) can be seen as a generalisation of SAT; they form an important class of combinatorial problems in Artificial Intelligence. In the second part of this chapter, we introduce various types of CSPs and give an overview of prominent SLS approaches to solving these problems. These approaches include encoding CSP instances into SAT and solving the encoded instances using SAT algorithms, as well as various rather straight-forward generalisations of SLS algorithms for SAT and native CSP algorithms.

6.1 The Satisfiability Problem

As motivated and formally defined in Chapter 1, the Satisfiability Problem in Propositional Logic (SAT) is to decide for a given propositional formula F, whether there exists an assignment of truth values to the variables in F under which F evaluates to true; such satisfying assignments are called models of F and form the solutions of the respective instance of SAT. When applying SLS algorithms to SAT, we are typically more interested in solving the search variant of SAT (*i.e.*, in finding models of a given formula) rather than the decision variant. It should be noted that typical SLS algorithms for SAT (including all SAT algorithms covered in this chapter) are incomplete and hence cannot determine with certainty that a given formula is unsatisfiable, *i.e.* has no models.

CNF Representations and Transformations

Most algorithms for SAT, including all state-of-the-art SLS algorithms, are restricted to formulae in conjunctive normal form (CNF), *i.e.*, to formulae that are conjunctions over disjunctions over literals. Since any propositional formula can be transformed into a logically equivalent CNF formula, in principle this restriction does not limit the class of SAT instances that can be solved by such algorithms. The naive method of transforming a non-CNF formula into CNF (using the distributive laws of propositional logic to resolve nestings of $\wedge$ and $\vee$ that are not allowed in CNF) can lead to an exponential growth in the length of the formula. There is, however, an alternative CNF transformation that avoids this effect at the cost of introducing a linear number (w.r.t. to the length of the formula) of additional propositional variables in the worst case [Poole, 1984]. When representing problems from other domains as SAT instances, in many cases relatively natural and concise CNF formulations can be found directly and without using general CNF transformation methods. Particularly, this is the case for many classes of CSPs and we will discuss approaches for encoding CSP instances as SAT in Section 6.5.

Alternative Formulations of SAT

Alternative representations of SAT for CNF formulae are used in various contexts, specifically, when techniques for solving more general problems are applied to SAT. As we will discuss in some more detail in Section 6.5, SAT can be seen as a special case of the more general Finite Discrete Constraint Satisfaction Problem (CSP). Another prominent representation encodes the truth values $\perp$ and $\top$ as integers 0 and 1 and propositional variables as integer variables with domain $\{0, 1\}$. Negated literals $\neg x$ are then encoded as $I(\neg x) = 1 - x$, while positive literals remain unchanged, *i.e.*, $I(x) = x$. Finally, the encoding of a CNF clause $c_i = l_1 \vee l_2 \vee l_3 \dots l_k$ is given by $I(c_i) = I(l_1) + I(l_2) + \dots + I(l_k)$ and the entire CNF formula $F = c_1 \wedge c_2 \wedge \dots \wedge c_m$ is encoded as $I(F) = I(c_1) \cdot I(c_2) \cdot \dots \cdot I(c_m)$. Then, a truth assignment a satisfies c_i if and only if the corresponding 0-1 assignment satisfies the inequality $I(c_i) \geq 1$, and the CNF formula F is satisfied under a if and only if $I(F) \geq 1$. Using this representation, SAT can be seen as a special case of a discrete constrained optimisation problem: Let $u_i(F, a) = 1$ if clause c_i is unsatisfied under assignment a and $u_i(F, a) = 0$ otherwise and $N(F, a) = \sum_{i=1}^{m} u_i(F, a)$. Then any model of F corresponds to a solution of $a^* = \min\{N(F, a) \mid a \in \{0, 1\}^n\}$ subject to $\forall i \in \{1, 2, \dots, m\} : u_i(F, a) = 0$. This type of constrained optimisation problem is known as 0-1 Integer Linear Programming (ILP) or Boolean Programming.

Using these representations, SAT instances can in principle be solved using more general CSP or ILP algorithms. In practice, however, this approach has not been able to achieve sufficiently high performance to provide a viable alternative to native SAT solvers such as the SLS algorithms presented in this chapter (see, *e.g.*, [Schuurmans *et al.*, 2001; Battiti and Protasi, 1998; Mitchell and Levesque, 1996]). However, a number of SAT algorithms, particularly some of the dynamic local search methods presented in Section 6.4, are inspired by more general CSP or constrained optimisation solving techniques. Furthermore, successful SLS algorithms for SAT have been extended to more general classes of CSPs and ILPs, resulting in competitive solvers for these problems (some of these generalised SLS algorithms will be discussed in Section 6.6). Finally, it may be noted that the ILP formulation of SAT can be easily generalised to weighted MAX-SAT, a closely related optimisation problem for which in some cases more general ILP methods perform much better than for SAT [**?**; Resende *et al.*, 1997].

Polynomial Simplification of CNF Formulae

One of the advantages of the native, logical formulation of SAT is that propositional formulae in general, and CNF formulae in particular can often be substantially simplified using computationally cheap reduction techniques. Such reductions have been shown to be crucial in solving various types of SAT instances more effectively; as preprocessing techniques, they can be used for simplifying the input to any SAT algorithm for CNF formulae.

One of the simplest reductions is the elimination of duplicate literals and clauses from a given CNF formula. Obviously, this can be performed in time $O(n)$, where n is the size of the formula, and results in a logically equivalent CNF formula. Similarly, all clauses that contain a variable and its negation and are hence trivially satisfied (tautological clauses), can be detected and eliminated in linear time. A slightly more interesting reduction is the elimination of subsumed clauses. A clause $c = l_1 \vee l_2 \vee \dots l_k$ is subsumed by another clause $c' = l'_1 \vee l'_2 \vee \dots l'_j$ if every literal in c' also occurs in c, *i.e.*, $\{l'_1, l'_2, \dots, l'_j\} \subseteq \{l_1, l_2, \dots, l_k\}$. Detection and elimination of all subsumed clauses can be performed efficiently and leads to a logically equivalent formula. Another linear time reduction is the elimination of clauses containing pure literals, *i.e.*, variables that either do only occur negated or unnegated in the same formula. Setting such a variable to true or false, respectively, does not change the satisfiability of the formula; hence, all clauses containing such variables can be removed.

One of the most important reduction techniques is based on the unit resolution method: If a CNF formula contains a unit clause, *i.e.*, a clause consisting of only a single literal, this clause and all clauses containing the same literal can be removed (this is a special case of the subsumption reduction), and all remaining occurrences of the corresponding variable, *i.e.*, the complementary literal) can be removed (this can be seen as a special case of the general resolution rule, see, *e.g.*, [Russel and Norvig, 1995]). Performing unit resolution to all unit clauses in the original CNF formula leads to a logically equivalent CNF formula, we also refer to this transformation as a single pass of unit propagation. It may be noted that unit resolution can lead to empty clauses, making the resulting formula trivially unsatisfiable or eliminate all clauses, leaving an empty CNF formula which is trivially satisfiable. Furthermore, unit resolution can produce new unit clauses and hence make further unit resolution steps possible. Repeated application of unit resolution eventually leads to a formula without any unit clauses. We refer to this reduction as complete unit propagation; it can be performed in time $O(n)$ and forms a crucial component of basically any systematic search algorithm for SAT. Unit propagation alone is sufficient for deciding deciding the satisfiability of Horn formulae, *i.e.*, CNF formulae in which every clause contains at most one unnegated variable [Dowling and Gallier, 1984], in linear time w.r.t. to the size of the given formula. It also forms the basis of a linear-time algorithm for solving SAT for 2-CNF formulae [del Val, 2000].

Unit propagation provides the basis for two other efficient and practially useful simplification techniques, unary and binary failed literal reduction. The key idea behind unary failed literal reduction is the following: If setting a variable x occurring in the given formula F to true makes F unsatisfiable, then adding the unit clause $c' = \neg x$ to F yields a logically equivalent formula F'. Since F' contains at least one unit clause, c', it can be simplified using unit propagation, which can result in a substantially smaller formula. Whether setting x to true renders F unsatisfiable is determined by adding a unit clause $c = x$ to F and by checking whether subsequent application of unit propagation produces an empty clause. Complete unary failed literal reduction consists of performing this operation for each variable occurring in the given formula and has complexity $O(n^2)$. Binary failed literal reduction works analogously but checks whether simultaneously adding any two unary binary clauses, $c_1 = x$ and $c_2 = y$ and applying unit propagation leads to a trivially unsatisfiable formula. If this is the case, the binary clause $c' = \neg x \vee \neg y$ is added to F, which potentially leads to further simplifications. Binary failed literal reduction has time complexity $O(n^3)$; it is a fairly expensive operation, but sometimes leads to substantial reductions in the overall time required for solving a given SAT instance (see, *e.g.*, [Kautz and Selman, 1996a; Brafman and Hoos, 1999]).

Randomly Generated SAT Instances

Many empirical studies of SAT algorithms have made use of randomly generated CNF formulae. Various such classes of SAT instances have been proposed and studied in literature; in most cases, they are obtained by means of a random instance generator that samples SAT instances from an underlying probability distribution over CNF formulae. The probabilistic generation process is typically controlled by various parameters which mostly determine syntactic properties of the generated formulae, such as number of variables and clauses, in a deterministic or probabilistic way.

One of the earliest and most widely studied classes of randomly generated SAT instances is based on the *random clause length model* (also called fixed density model): Given a number of variables, n, and clauses, m, the clauses are constructed independently from each other by including each of the $2n$ literals with fixed probability p (see [Franco and Paull, 1983], a variant of this model was used in Goldberg's empirical study on the average case time complexity of the Davis Putnam algorithm [Goldberg, 1979]). Theoretical and empirical results show that this family of instance distributions is mostly easy to solve on average using rather simple deterministic algorithms [Cook and Mitchell, 1997; Franco and Swaminathan, 1997]. As a consequence, the random clause length model is no longer widely used for evaluating the performance of SAT algorithms. Similar considerations apply to other distributions of SAT instances, such as the instances obtained from the AIM instance generator [Asahiro *et al.*, 1996], which can be solved in polynomial time by binary failed literal reduction [Hoos and Stützle, 2000c].

To date, the most prominent class of randomly generated SAT instances that is used extensively for evaluating the performance of SAT algorithms is based on the *fixed-clause-length model* and known as *Uniform Random-k-SAT* [Franco and Paull, 1983], [Mitchell *et al.*, 1992]. For a given number of variables, n, number of clauses, m, and clause length k, Uniform Random k-SAT instances are obtained as follows: To generate a clause, k literals are chosen independently and uniformly at random from the set of $2n$ possible literals (the n propositional variables and their negations). Clauses are not included into the problem instance if they contain multiple copies of the same literal or if they are tautological, *i.e.*, they contain a variable and its negation. Using this mechanism, clauses are generated and added to the formula until it contains m clauses overall.

Random k-SAT Hardness and Solubility Phase Transition

One particularly interesting property of uniform Random k-SAT is the occurrence of a phase transition phenomenon, *i.e.*, a rapid change in solubility which can be observed when systematically increasing (or decreasing) the number m of clauses for a fixed number of variables n [Mitchell *et al.*, 1992; Kirkpatrick and Selman, 1994]. More precisely, for small m, almost all formulae are underconstrained and therefore satisfiable; when reaching some critical $m = m^*$, the probability of generating a satisfiable instance drops sharply to almost zero. Beyond m^*, almost all instances are overconstrained and thus unsatisfiable. For Random 3-SAT, it has been empirically shown that this phase transition occurs approximately at $m^* = 4.26n$ for large n; for smaller n, the critical clauses/variable ratio m^*/n is slightly higher [Mitchell *et al.*, 1992; Crawford and Auton, 1996]. For fixed k, the transition becomes increasingly sharp as n grows; furthermore, the critical value m^* increases with k [Kirkpatrick and Selman, 1994].

Empirical analyses have shown that problem instances from the phase transition region of Uniform Random 3-SAT tend to be particularly hard for both systematic SAT solvers [Cheeseman *et al.*, 1991; Crawford and Auton, 1996] and SLS algorithms [Yokoo, 1997]. Striving to evaluate their algorithms on hard problem instances, many researchers are using test-sets sampled from the phase transition region of Uniform Random 3-SAT. Particularly in the context of empirical studies including incomplete SAT algorithms, these test-sets are separated into satisfiable and unsatisfiable instances using state-of-the-art complete SAT solvers [Hoos and Stützle, 2000c]. Although similar results hold for Uniform Random k-SAT with $k > 3$, test-sets from these instance distributions are rarely used.

SAT-Encodings of Other Combinatorial Problems

Since SAT is an $\mathcal{NP}$-complete problem, any other problem in $\mathcal{NP}$ can be encoded into SAT in polynomial time and space. SAT-encoded instances of various combinatorial problems play an important role in evaluating and characterising the performance of SAT algorithms; these combinatorial problems stem from various domains, including mathematical logic, artificial intelligence and VLSI engineering.

Finite, discrete constraint satisfaction problems (CSPs) can be seen as a generalisation of SAT that allow variables to have domains other than truth values and constraints between the values assigned to individual variables that are different from the ones captures by CNF clauses. CSPs are also often a natural intermediate stage in encoding other combinatorial problems into SAT. CSP instances can be encoded into SAT in various ways; CSPs and their encodings into SAT will be further discussed in Section 6.5. It has been shown that certain types of randomly generated CSPs can be solved at least as efficiently by applying current SAT algorithms to SAT-encoded instances as by using state-of-the-art CSP algorithms [Hoos, 1998b; 1999b] (see also Section 6.6).

Other prominent examples of SAT-encoded instances of combinatorial problems include graph colouring, various types of planning and scheduling problems, boolean function learning, inductive inference, cryptographic key search, n-Queens (see [Gu *et al.*, 1997; Hoos and Stützle, 2000c]). For some of these, particularly in the case of SAT-encoded STRIPS planning problems from the Blocks World and Logistics domains, applying SAT solvers and reduction techniques to suitably encoded problem instances was shown to achieve performance competitive with state-of-the-art algorithms for the original problem [Kautz and Selman, 1996b]. Key factors underlying such results are the conceptual simplicity of SAT, which facilitates the design and efficient implementation of algorithms, and the large amount of knowledge on techniques for solving SAT and their specific properties. Furthermore, using suitable SAT encodings and reduction techniques is of crucial importance for solving the resulting SAT problems efficiently. Interestingly, the size of the SAT encodings is not always indicative of the difficulty of solving them. Particularly, it has been shown for various problem domains that compact SAT encodings that result in instances with small search spaces can be much more difficult to solve than sparser encodings that produce instances with substantially bigger search spaces [Ernst *et al.*, 1997; Hoos, 1998b; 1999b].

Some Practical Applications of SAT

Despite its conceptual simplicity and abstract nature, the SAT problem has various practical applications. Some of the most prominent industrially relevant SAT applications stem from hardware design and verification, in particular, from the verification of reactive systems, such as microprocessor components. In an approach called Bounded Model Checking (BMC), a system and a specification of its formal properties can be encoded into a propositional formula, whose models correspond to bugs, *i.e.*, situations in which the behaviour of the system violates its specifications [Biere *et al.*, 1999a; 1999b]. Similar to SAT encodings of planning problems that require the plan length to be bounded, in BMC, the size of the bug, *i.e.* the number of states of the system involved in the bug, is limited by a constant. It may be noted that for proving that a given system does not have any bugs below a certain size, a complete SAT solver is required. Incomplete SAT solvers, such as the SLS algorithms for SAT covered in Section **??**, can be used, however, to find bugs efficiently.

Symbolic model checking methods, such as BMC, are increasingly gaining industrial acceptance, because compared to traditional, simulation-based validation techniques, they detect a wider range of bugs, including subtle error conditions. Many traditional formal verification techniques use Binary Decision Diagrams (BDDs) [Bryant, 1986] for representing propositional formulae. By using CNF encodings and standard SAT algorithms in a BMC approach, it is often possible to find bugs faster, and to find bugs of minimal size; the latter is important since small bugs are typically easier to understand for a human system tester or designer. Furthermore, BDD based approaches often require extremely large amounts of memory as well as specialised techniques for finding models of the given propositional formula, while the CNF representations are typically more concise and can be solved using standard SAT algorithms [Biere *et al.*, 1999a].

Another application area in which SAT encodings and solvers have been successfully used for solving real-world problems is asynchronous circuit design [Vanbekbergen *et al.*, 1992; Gu and Puri, 1995]. In one prominent approach to asynchronous circuit synthesis, the circuits are specified using signal transition graphs (STGs). One of the core problems is then to assign a distinguishable binary code to every circuit state. This Complete State Coding Problem (CSC) can be modelled as a SAT problem, but the size and hardness of the formulae thus obtained limits the practical applicability of using SAT algorithms for solving the CSC problem. However, by partioning the STG into smaller components and using a SAT algorithm to solve the corresponding CSC subproblems, substantial performance improvements can be obtained for industrial asynchronous circuit design benchmarks [Gu and Puri, 1995].

Finally, SAT algorithms have been recently used for solving real world sports scheduling problems [Zhang, 2002]. Specifically, the problem of finding fair schedules for college conference basketball tournaments can be encoded into SAT. This encoding is based on a decomposition of the problem into three phases each of which deals with different constraints of the overall scheduling problem. Using a standard SAT algorithm for solving the SAT instances for the three phases, real-world college conference basketball scheduling problems were solved substantially more efficiently than by previous, specialised techniques and more balanced schedules were obtained than the ones that are currently used for these tournaments [Zhang, 2002].

Generalisations and Related Problems

Many generalisations of the Propositional Satisfiability Problem have been proposed and studied in the literature. As mentioned above, the Constraint Satisfaction Problem (CSP) can be seen as generalisations of SAT. Multi-Valued SAT [Béjar and Manyà, 1999; Frisch and Peugniez, 2001b] and Pseudo-Boolean CSP [Abramson *et al.*, 1996; Connolly, 1992; Walser, 1997; kketangen, 2002] are two special cases of CSP that are closely related to SAT. Multi-Valued SAT (MV-SAT) allows variables whose domains are arbitrary finite sets of values and uses logical constraints similar to CNF clauses. Pseudo-Boolean CSPs use binary variables with domain $\{0, 1\}$ but allow more general constraints. Both MV-SAT and Pseudo-Boolean CSP as well as general finite discrete CSPs will be further discussed in Section 6.5.

The optimisation variant of SAT in which the objective is to maximise the number of satisfied clauses of a given CNF formula rather than completely satisfying every clause is called MAX-SAT. In a further generalisation called weighted MAX-SAT, weights (usually positive integer or real numbers) are associated to the clauses of a given CNF formula and the objective is to find a variable assignment that maximises the total weight of the satisfied clauses. As one of the conceptually simplest combinatorial optimisation problems and because of its close relation to SAT, MAX-SAT plays an important role in the development and evaluation of search algorithms for hard combinatorial problems. In general, the best known methods for solving MAX-SAT problems are SLS algorithms. MAX-SAT problems and SLS algorithms for MAX-SAT will be discussed in more detail in Chapter 7.

Another interesting generalisation of SAT is dynamic SAT (DynSAT)

[Hoos and OŃeill, 2000]; intuitively, in DynSAT, a given CNF formula changes over time and a solution consists of a sequence of models such that at any time, the current CNF formula is satisfied by the current model. Equivalently, DynSAT can be defined in such a way that each problem instance consists of a conventional CNF formula some of whose variables are fixed to specific truth values at certain times. SLS algorithms for SAT can be generalised to DynSAT in a straight-forward way and appear to be well-suited for solving these problems.

Let us mention three other prominent problems that are closely related to SAT. In the Propositional Validity Problem (VAL), the objective is to decide whether a given propositional formula F is valid, *i.e.*, has no variable assignment that is not a model (see [Russel and Norvig, 1995]) VAL and SAT are dual problems in the sense that any formula F is valid if and only if $\neg F$ is unsatisfiable. Hence, any complete algorithm for SAT can be used for deciding VAL and vice versa. VAL is an important problem in theorem proving and has applications in Artificial Intelligence and other areas of Computer Science. The Satisfiability Problem for Quantified Boolean Formulae (QSAT) can be seen as a generalisation of both SAT and VAL. A Quantified Boolean Formula (QBF) is a propositional formula in which all variables are quantified existentially ($\exists$) or universally ($\forall$). A QBF of the form $\exists x : F$ is satisfiable if either assigning $x := \top$ or $x := \bot$ makes F satisfiable and a QBF of the form $\forall x : F$ is satisfiable if both $x := \top$ and $x := \bot$ render F satisfiable (see, *e.g.*, [Cadoli *et al.*, 2002; Rintanen, 1999b]). Many important problems in Artificial Intelligence can be mapped directly into QSAT, including conditional planning, abduction, and non-monotonic reasoning [Rintanen, 1999a; 1999b]. QSAT also plays a prominent role in complexity theory, where it is prototypical and complete for the problems in the polynomial hierarchy.

Finally, #SAT is a variant of SAT in which given a propositional formula F, the objective is to determine the number of models of F (counting variant) or to decide whether F has at least a given number i of models (decision variant) [Roth, 1996; Bailey *et al.*, 2001]. This problem has important applications to approximate reasoning problems in Artificial Intelligence; it is also of substantial theoretical interest, as the counting variant of #SAT is the prototypical complete problem for the complexity class $\#\mathcal{P}$ and the decision variant is a prototypical complete problem for the probabilistic complexity class $\mathcal{PP}$.

6.2 The GSAT Architecture

The GSAT algorithm [Selman *et al.*, 1992] was one of the first SLS algorithms for SAT; it had a very significant impact on the development of a broad range of SAT solvers, including most of the current state-of-the-art SLS algorithms for SAT. Like all SAT algorithms covered in this chapter, GSAT is based on a 1-exchange neighbourhood in the space of all complete truth assignments of the given formula; under this 'one-flip neighbourhood', two variable assignments are neighbours if and only if they differ in the truth assignment of exactly one variable. Furthermore, GSAT uses an evaluation function $g(F, a)$ that maps each variable assignment a to the number of clauses of the given formula F unsatisfied under a. Note that the models of F are exactly the assignments with evaluation function value zero. GSAT and most of its variants are iterative improvement methods that flip the truth value of one variable in each search step. The selection of the variable to be flipped is typically based on the *score* of a variable x under the current assignment a; this is defined as $g(F, a) - g(F, a')$, where a' is the assignment obtained from a by flipping the truth value of x. Algorithms of the GSAT architecture differ primarily in the underlying variable selection method. In the following, we describe some of the most widely known and best-perfoming GSAT algorithms.

Basic GSAT

The core of the basic GSAT algorithm [Selman *et al.*, 1992] consists of a simple best-improvement search strategy: Starting from a randomly chosen variable assignment, in each local search step, one of the variables with maximal score, *i.e.*, a variable that results in a maximal decrease in the number of unsatisfied clauses, is flipped. If there are several variables with maximal score, one of them is randomly selected according to a uniform distribution. The iterative best-improvement search used in GSAT gets easily stuck in local minima of the evaluation function. Therefore, GSAT uses a simple static restart mechanism that re-initialises the search at a randomly chosen assignment every $maxFlips$ flips. The search is terminated, when a model of the given formula F has been found, or after $maxTries$ sequences (also called 'tries') of $maxFlips$ variable flips each have been performed without finding a model of F (see Figure 6.1).

```
procedure GSAT ($F$, $maxTries$, $maxSteps$)
    input CNF formula $F$, positive integers $maxTries$ and $maxSteps$
    output model of $F$ or "no solution found"
    for $try$ := 1 to $maxTries$ do
        $a$ := randomly chosen assignment of the variables in formula $F$;
        for $step$ := 1 to $maxSteps$ do
            if $a$ satisfies $F$ then return $a$;
            $v$ := randomly selected variable the flip of which minimises
                 the number of unsatisfied clauses;
            $a$ := $a$ with $v$ flipped;
        end for;
    end for;
    return "no solution found";
end GSAT
```

Figure 6.1: The basic GSAT algorithm; all random selections are according to a uniform probability distribution over the underlying sets.

Straightforward implementations of GSAT are rather inefficient, since in each step the scores of all variables have to be calculated from scratch. The key to efficiently implementing GSAT is to compute the complete set of scores only once at the beginning of each try, and then after each flip to update only the scores of those variable which were possibly affected by the flipped variable. Details on these implementation issues for GSAT and related algorithms are discussed in an in-depth section on page 218.

For a any fixed number of restarts, GSAT is essentially incomplete [Hoos, 1998b; 1999a], and severe stagnation behaviour is observed on most SAT instances. Still, when it was introduced, GSAT outperformed the best systematic search algorithms for SAT available at that time. Todate, basic GSAT's performance is substantially weaker than that of any of the other algorithms described in the following, and the algorithm is mainly of historical interest.

GSAT with Random Walk (GWSAT)

Basic GSAT can be significantly improved by extending the underlying search strategy into a randomised best-improvement method. This is achieved by introducing an additional type of local search steps, so-called *conflict-directed random walk steps*. In such a random walk step, first a currently unsatisfied clause c' is selected uniformly at random. Then, one of the variables appearing in c' is randomly selected and flipped, thus effectively forcing c' to become satisfied. A simple SLS algorithm that initialises the search by randomly picking an assignment (like basic GSAT) and then performs a sequence of these conflict-directed random walk steps has been proven to solve 2-SAT in quadratic expected time [Papadimitriou, 1991]; this result inspired the use of this type of random walk to extend basic GSAT.

The basic idea of GWSAT is to decide at each local search step with a fixed probability *wp* (called *walk probability* or noise setting) whether to do a standard GSAT step or a conflict-directed random walk step. For any $wp > 0$, this algorithm allows arbitrarily long sequences of random walk steps; as detailed in [Hoos, 1999a], this implies that from arbitrary assignments, a model (if existent) can be reached with a positive, bounded probability. In particular, this allows the algorithm to escape from any local minima region of the underlying search space. Hence, the probability that GWSAT (without random restart), applied to a satisfiable formula finds a solution converges to one as the run-time approaches infinity, *i.e.*, GWSAT is probabilistically approximately complete (PAC). Like all GSAT algorithms, GWSAT uses the same static restart mechanism as basic GSAT.

Generally, GWSAT achieves substantially better performance than basic GSAT. It has been shown that when using sufficiently high noise settings (the precise threshold varies between problem instances), GWSAT does not suffer from stagnation behaviour. Furthermore, for hard SAT instances, it typically shows exponential RTDs [Hoos, 1998a; Hoos and Stützle, 1999]; hence, static restarts are ineffective, and optimal speedup can be obtained by a multiple independent runs parallelisation (*cf.* Section 4.4). For low noise settings, stagnation behaviour is frequently observed; recently, there has been evidence that the corresponding RTDs can be characterised by mixtures of exponential distributions [Hoos, 2002a].

GSAT with Tabu Search (GSAT/TABU)

The best-improvement search underyling basic GSAT can be easily extended into a tabu search strategy (see also Example 2.4 on page 69). GSAT/TABU is obtained from basic GSAT by associating a tabu status with propositional variables of the given formula [Mazure *et al.*, 1995; McAllester *et al.*, 1997; Steinmann *et al.*, 1997]. In GSAT/TABU, after a variable x has been flipped, it cannot be flipped back within the next *tl* steps, where the tabu tenure, *tl*, is a parameter of the algorithm. In each search step, the variable to be flipped is selected like in basic GSAT, except that the choice is restricted to variables that are currently not tabu. Upon search initialisation, the tabu status of all variables is cleared. Efficient implementations of GSAT/TABU store for each variable x the time (*i.e.*, search step number) t_x when it was last flipped. When initialising the search, all the t_x are set to $-tl$; subsequently, every time a variable x is flipped, t_x is set to the current search step number t since the last initialisation of the search process. A variable x is tabu if and only if $t - t_x \leq tl$.

Unlike in the case of GWSAT, it is not clear whether GSAT/TABU with fixed cutoff parameter *maxTries* has the PAC property. Intuitively, for low *tl* the algorithm might not be able to escape from extensive local minima regions without using restart, while for high *tl* settings all the routes to a solution might be cut off because too many variables are tabu. In practice, for very short tabu tenure, GSAT/TABU often shows severe stagnation behaviour (the *tl* value for which this occurs depends on the given problem instance). For sufficiently high tabu tenure settings, GSAT/TABU does not suffer from stagnation behaviour and for hard problem instances, shows exponential RTDs. As with GWSAT's noise parameter, very high settings of *tl*, although not causing stagnation behaviour, uniformly decrease GSAT/TABU's performance.

Using instance specific optimised tabu tenure settings for GSAT/TABU and noise settings for GWSAT, GSAT/TABU typically performes significantly better than GWSAT, particularly when applied to large and structured SAT instances [Hoos and Stützle, 2000a]. (There are, however, a few exceptional cases where GSAT/TABU performs substantially worse than GWSAT, including well-known SAT-encoded instances of logistics planning problems.) Analogous to basic GSAT, GSAT/TABU can be extended with a random walk mechanism; limited experimentation suggests that typically this hybrid algorithm does not perform better than GSAT/TABU [Steinmann *et al.*, 1997]. Overall, with the exception of the dynamic local search algorithms covered in Section 6.4, GSAT/TABU is one of the best-performing variants of GSAT known todate.

HSAT and HWSAT

The intuition behind HSAT [Gent and Walsh, 1993b] is based on the observation that in basic GSAT some variables might never get flipped although they are frequently eligible to be chosen. This can cause stagnation behaviour, since one of these variables might have to be flipped to allow the search to make further progress. Therefore, when in a search step there are several variables with identical score, HSAT selects the least recently flipped variable, that is, the variable that was flipped longest ago. Only shortly after search initialisation, when there are still variables that have not been flipped, HSAT performs the same random tiebreaking between variables with identical score as plain GSAT. Apart from this difference in the variable selection mechanism, HSAT is identical to basic GSAT.

Although HSAT was found to show superior performance over basic GSAT [Gent and Walsh, 1993b], it is clear that it is even more likely to get stuck in local minima from which it cannot escape, since the history-based tie-breaking rule effectively restricts the search trajectories when compared to GSAT. To counteract this problem, HSAT can be extended with the same random walk mechanism as used in GWSAT. The resulting variant is called HWSAT [Gent and Walsh, 1995]; like GWSAT, HWSAT has the PAC property. Generally, HWSAT shows improved peak performance over GWSAT. Compared to GSAT/TABU, HWSAT's performance appears to be somewhat better on hard Uniform Random-3-SAT instances and certain types of structured SAT problems, and significantly worse in many other cases [Hoos and Stützle, 2000a].

In Depth: Efficiently Implementing GSAT

The key to implementing GSAT algorithms efficiently lies in caching and updating the variable scores that form the basis for selecting the variable to be flipped in each search step. Typically, not all variable scores change after each search step; this suggests that rather than recomputing all variable scores in each step, it should be more efficient to compute all scores when the search is initialised, but to subsequently only update the scores affected by a variable flip in each search step. The following definitions will help to explain the precise mechanism for updating the scores and to analyse its time complexity.

Definition 6.1 (Variable and Clause Dependencies)

Given a CNF formula F and two variables x, x' appearing in F. Then x' is dependent on x (and vice versa) if there is a clause in which both x and x' appear.

Furthermore, we define the set of variables dependent on x as

$$V_{dep}(F,x) := \{x' \in \mathit{Var}(F) \mid x' \text{ is dependent on } x\}$$

A clause c of F is dependent on x, if x appears in c, and the set of clauses dependent on x is defined as

$$C_{dep}(F,x) := \{c \text{ is a clause of } F \mid c \text{ is dependent on } x\}$$

A clause c is *critically satisfied by a variable x under assignment a* if x appears in c, c is satisfied under a, and flipping the value of x makes c unsatisfied. Finally, a variable x' is critically dependent on a variable x under assignment a, if there is a clause c that is dependent on x and x', and flipping x results in the clause to change its satisfaction status from (i) satisfied to unsatisfied or vice versa, or (ii) satisfied to critically satisfied or vice versa. □

After flipping a variable x, only clauses dependent on x can change their satisfaction status; hence, in order to update the evaluation function value, *i.e.*, the number of unsatisfied clauses, only the clauses in $C_{dep}(x, F)$ need to be considered. According to the definition of a variable's score, the score of x just changes its sign as a consequence of flipping x. For all other variables $x' \neq x$, the score of x' remains unchanged if x' is not dependent on x, *i.e.*, if $x' \notin V_{dep}(F,x)$. Hence, after flipping x, only the scores of the variables in $V_{dep}(F,x)$ need to be updated. In fact, among those, only the scores of variables that critically depend on x can actually change.

For a given formula F with n variables, m clauses, and a clause length (number of literals per clause) bounded from above by $CL(n)$, the time complexity of computing all variable scores is $O(m \cdot CL(n))$. This is achieved by going through all clauses, checking their satisfaction status and increasing or decreasing the scores of the variables appearing in a clause c, depending on whether c is currently unsatisfied, or whether it is critically satisfied by a given variable. At the end of this process, the evaluation function value, a list of all unsatisfied clauses, and all variable scores have been computed.

After each search step, all variable scores that are affected by the respective flip can be updated in time $O((CD(n) \cdot CL(n))$, where $CD(n)$ is an upper bound on the cardinality of the sets $C_{dep}(F, x)$. This is achieved by going through all clauses that are dependent on the flipped variable, x, and updating the scores of the variables occuring in these, depending on the (critical) satisfaction status of the respective clause before and after the flip of x. In order to perform this operation efficiently, for each variable x a list is kept of the clauses that are dependent on x; these lists are built when parsing the input formula. For each variable, we furthermore store its current truth value and score, and for each clause, we store its (critical) satisfaction status under the current assignment.

For Uniform Random-k-SAT formulae with fixed clauses/variable ratio, using this implementation of GSAT achieves, because the average number of dependent clauses for each variable is constant, independent of instance size, a time complexity of $O(1)$ for each search step, compared to $\Theta(n^2)$ for a naive implementation in which all variable scores are computed before every variable flip. For SAT-encoded instances of other combinatorial problems, there are typically more extensive variable dependencies, leading to a somewhat reduced, but still substantial performance advantage of the efficient implementation described above.

The efficient mechanism for caching and updating variable scores described here is also used in Selman and Kautz' publically available reference implementation of GSAT. Very similar techniques can be used for efficiently implementing other SLS algorithms, such as Galinier and Hao's Tabu Search algorithm for CSP, which is outlined in Section 6.6. Interestingly, for the WalkSAT algorithms described in the following, a more straight-forward implementation achieves slightly better performance.

6.3 The WalkSAT Architecture

The WalkSAT architecture is based on ideas first published by Selman, Kautz, and Cohen in 1994 [Selman *et al.*, 1994] and was later formally defined as an algorithmic framework by McAllester, Selman, and Kautz in 1997 [McAllester *et al.*, 1997]. WalkSAT can be seen as an extension of the conflict directed random walk method that is also used in Papadimitriou's algorithm [Papadimitriou, 1991] and GWSAT. It is based on a 2-stage variable selection process focused on the variables occurring in currently unsatisfied clauses. For each local search step, in a first stage a currently unsatisfied clause c' is randomly selected. In a second stage, one of the variables appearing in c' is then flipped to obtain the new assignment. Thus, while the GSAT architecture is characterised by a static neighbourhood relation between assignments with Hamming distance one, using this

```
procedure WalkSAT ($F$, $maxTries$, $maxSteps$, $slc$)
    input CNF formula $F$, positive integers $maxTries$ and $maxSteps$,
        heuristic function $slc$
    output model of $F$ or "no solution found"
    for $try$ := 1 to $maxTries$ do
        $a$ := randomly chosen assignment of the variables in formula $F$;
        for $step$ := 1 to $maxSteps$ do
            if $a$ satisfies $F$ then return $a$;
            $c'$ := randomly selected clause unsatisfied under $a$;
            $v$ := variable selected from $c'$ according to heuristic function $slc$;
            $a$ := $a$ with $v$ flipped;
        end for;
    end for;
    return "no solution found";
end WalkSAT
```

Figure 6.2: The WalkSAT algorithm family. All random selections are according to a uniform probability distribution over the underlying sets; WalkSAT algorithms differ in the variable selection heuristic *slc*.

procedure, WalkSAT algorithms are effectively based on a dynamically determined subset of the GSAT neighbourhood relation. As a consequence of this substantially reduced effective neighbourhood size, WalkSAT algorithms can be implemented efficiently without caching variable scores and still achieve substantially lower CPU times per search step than efficient GSAT implementations [Hoos, 1998a; Hoos and Stützle, 2000a]. All WalkSAT algorithms considered here use the same random search initialisation and static random restart as GSAT. A pseudo-code representation of the WalkSAT architecture is shown in Figure 6.2.

WalkSAT/SKC

The first WalkSAT algorithm, originally introduced in [Selman *et al.*, 1994], differs in one important aspect from most of the other SLS algorithms for SAT: The scoring function $score_b(x)$ used by WalkSAT/SKC counts the number of currently satisfied clauses that will be broken, *i.e.*, become unsatisfied, by flipping a given variable x. Using this scoring function, the following variable selection scheme is applied: If there is a variable with $score_b(x) = 0$ in the clause c' selected in stage 1, that is, if c' can be satisfied without breaking another clause, this variable is flipped ("zero-damage" step). If no such variable exists, with a certain probability *1-p* the variable with minimal $score_b$ value is selected; in the remaining cases, *i.e.* with probability *p* (noise setting), one of the variables from c' is randomly selected (random walk step).

Conceptually as well as historically, WalkSAT/SKC is closely related to GWSAT. However, there are a number of significant differences between both algorithms, which in combination account for the generally superior performance of WalkSAT/SKC. While both algorithms use the same kind of random walk steps, WalkSAT/SKC applies them only under the condition that there is no variable with $score_b(x) = 0$. In GWSAT, however, random walk steps are done in an unconditional probabilistic way. From this point of view, WalkSAT/SKC is greedier, since random walk steps, which usually increase the number of unsatisfied clauses, are only done when every variable occurring in the selected clause would break some clauses when flipped. Yet, in a greedy step, due to its two-stage variable selection scheme, WalkSAT/SKC chooses from a significantly reduced set of neighbours and can therefore be considered less greedy than GWSAT. Finally, because of the different scoring function, in some sense, GWSAT shows a greedier behaviour than WalkSAT/SKC: In a best-improvement step, GWSAT would prefer a variable which breaks some clause but compensates for this by fixing some other clauses, while in the same situation, WalkSAT/SKC would select a variable with a smaller total score, but breaking also a smaller number of clauses.

It has been proven that WalkSAT/SKC with fixed cutoff parameter *maxSteps* has the PAC property for 2-SAT formulae [Culberson *et al.*, 2000], but it is not known whether the algorithm is PAC in the general case. Different from GWSAT, it is not clear whether WalkSAT/SKC can perform arbitrarily long sequences of random walk steps, since random walk steps are only possible when the selected clause does not allow any "zero-damage" steps. In practice, however, WalkSAT/SKC does not appear to suffer from any stagnation behaviour when using sufficiently high (instance specific) noise settings, in which case its run-time behaviour is characterised by exponential RTDs [Hoos, 1998a; Hoos and Stützle, 1999; 2000a]. Like in the case of GWSAT, stagnation behaviour is frequently observed for low noise settings, and there is some evidence that the corresponding RTDs can be characterised by mixtures of exponential distributions [Hoos, 2002a].

Generally, when using (instance specific) optimised noise settings, WalkSAT/SKC probabilistically dominates GWSAT in terms of variable flips required for finding a model to a given formula, but it does not always reach the performance of HWSAT or GSAT/TABU. When comparing CPU time, however, WalkSAT/SKC typically outperforms all GSAT variants presented in Section 6.2.

WalkSAT with Tabu Search (WalkSAT/TABU)

Analogously to GSAT/TABU, there is also an extension to WalkSAT/SKC that uses a tabu search mechanism. WalkSAT/TABU [McAllester *et al.*, 1997] uses the same two stage selection mechanism and the same scoring function $score_b$ as WalkSAT/SKC and additionally enforces a tabu tenure of *tl* steps for each flipped variable. (To implement this tabu mechanism efficiently, the same approach is used as described in Section 6.2 for GSAT/TABU.) In WalkSAT/TABU, if the selected clause c' does not allow a zero damage step, of all the variables occurring in c' that are not tabu the one with the highest $score_b$ value is picked; when there are several variables with the same maximal score, one of them is randomly selected according to a uniform probability distribution. It may happen, however, that all variables appearing in c' are tabu, in which case no variable is flipped (a so-called *null-flip*).

As shown in [Hoos, 1998b; 1999a], WalkSAT/TABU with fixed cutoff parameter *maxSteps* is essentially incomplete. Although this is mainly caused by null-flips, it is not clear whether replacing null-flips by random walk steps, for instance, would be sufficient for obtaining the PAC property. In practice, when using sufficiently high (instance specific) tabu tenure settings, WalkSAT/TABU's run-time behaviour is characterised by exponential RTDs; but there are cases (particularly for structured SAT instances) in which extreme stagnation behaviour is observed. Typically, however, WalkSAT/TABU performs significantly better that WalkSAT/SKC, and there are structured SAT instances (*e.g.*, large SAT-encoded blocks world planning problems), where WalkSAT/TABU appears to achieve better performance than any other SLS algorithm currently known.

Novelty and Novelty$^+$

Novelty [McAllester *et al.*, 1997] is a recent WalkSAT algorithm that uses history-based variable selection mechanism in the spirit of HSAT. Novelty, too, is based on the intuition, that repeatedly flipping back and forth the same variable should be avoided. Additionally the number of local search steps that have been performed since a variable was last flipped (also called the variable's *age*) is taken into consideration. An important difference of Novelty compared to WalkSAT/SKC and WalkSAT/TABU is that it uses the same scoring function as GSAT.

In Novelty, after an unsatisfied clause has been chosen, the variable to be flipped is selected as follows. If the variable with the highest score does not have minimal age among the variables within the same clause, it is always selected. Otherwise, it is only selected with a probability of *1-p*, where *p* is a parameter called the noise setting. In the remaining cases, the variable with the next lower score is selected. If there are several variables with identical score, the reference implementation by Kautz and Selman always chooses the one appearing first in the selected clause.

Note that for $p > 0$, the age-based variable selection of Novelty probabilistically prevents flipping the same variable over and over again; at the same time, flips can be immediately reversed with a certain probability if a better choice is not available. Generally, the Novelty algorithm is significantly greedier than WalkSAT/SKC, since always one of the two most improving variables from a clause is selected, where WalkSAT/SKC may select any variable if no improvement without breaking other clauses can be achieved. Also, Novelty is more deterministic than WalkSAT/SKC and GWSAT, since its probabilistic decisions are more limited in their scope and take place under more restrictive conditions. For example, different from WalkSAT/SKC, the Novelty strategy for variable selection within a clause is deterministic for both $p = 0$ and $p = 1$.

On one hand side, this typically leads to a significantly improved performance of Novelty when compared to WalkSAT/SKC. On the other hand, because of this property, it can be shown that Novelty is essentially incomplete [Hoos, 1998a], as selecting only among the best two variables in a given clause can lead to situations where the algorithm gets stuck in local minima of the objective function. As shown in [Hoos and Stützle, 2000a], this situation can be observed for a number of commonly used benchmark

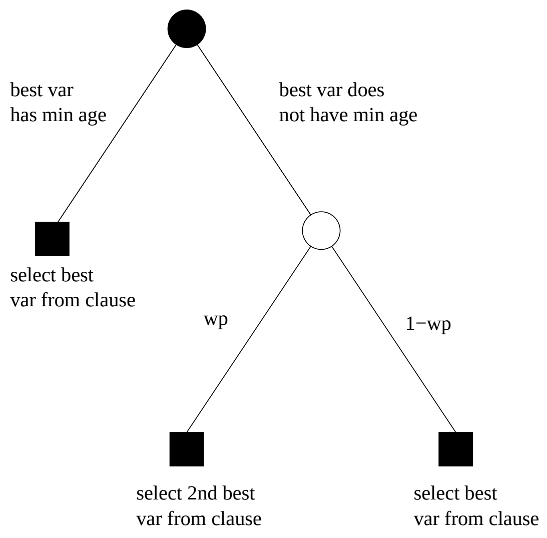
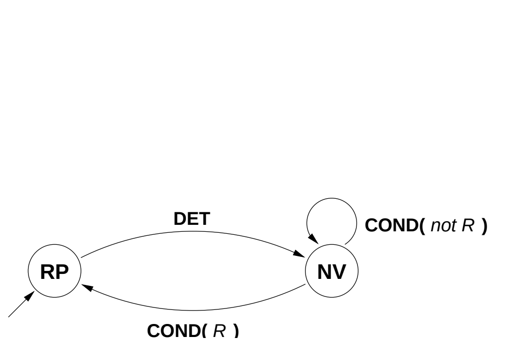


Figure 6.3: Decision tree representation of Novelty's mechanism for selecting a variable to be flipped within a given clause. Deterministic and probabilistic choices are represented by black and white circles, respectively; edges are labelled with the respective conditions and probabilities. Black boxes indicate variable decision actions.

instances, where it severely compromises Novelty's performance.

By extending Novelty with conflict-directed random walk analogously to GWSAT, the essential incompleteness as well as the empirically observed stagnation behaviour can be overcome. The Novelty$^+$ algorithm [Hoos, 1998b; 1999a] selects the variable to be flipped according to the standard Novelty mechanism with probability $1 - wp$, and performs a random walk step, as defined above for GWSAT, in the remaining cases. A GLSM model of the resulting algorithm is shown in Figure 6.4.

Novelty$^+$ is provably PAC for $wp > 0$ and shows exponential RTDs for sufficiently high (instance specific) settings of the primary noise parameter, *p*. In practice, small walk probabilities, *wp*, are generally sufficient to prevent the extreme stagnation behaviour that is occasionally observed for Novelty and to achieve substantially superior performance to Novelty. In fact, a uniform setting of $wp = 0.01$ seems to result in uniformly good



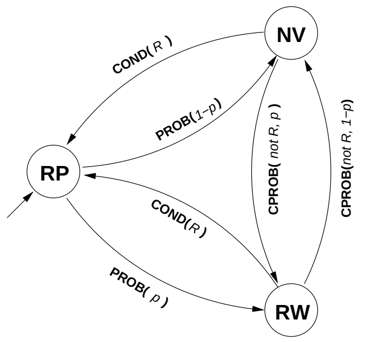


Figure 6.4: GLSM models for Novelty (left) and Novelty$^+$ (right); the restart predicate R is equal to countm(m), GSLM state RP initialises the search at randomly selected variable assignment, NV performs a Novelty step, and RW performs a random walk step (see text for details).

performance [Hoos, 1999a], and the algorithm's performance appears to be much more robust w.r.t. to the *wp* parameter than w.r.t. to the primary noise setting, *p*. In cases where Novelty does not suffer from stagnation behaviour, Novelty$^+$'s performance for $wp = 0.01$ is typically almost identical to Novelty's. Overall, Novelty$^+$ is one of the best-performing WalkSAT algorithms currently known and one of the best SLS algorithms for SAT available to date [Hoos and Stützle, 2000a; Hutter *et al.*, 2002].

R-Novelty and R-Novelty$^+$

R-Novelty [McAllester *et al.*, 1997] is a variant of Novelty which is based on the intuition that, when deciding between the best and second best variable (using the same scoring function as for Novelty), the actual difference of the respective scores should be taken into account. The exact mechanism for choosing a variable from the selected clause can be seen from the decision tree representation given in Figure **??**. Note that the R-Novelty heuristic is quite complex – as reported in [McAllester *et al.*, 1997], it was discovered by systematically testing a large number of WalkSAT variants.

R-Novelty's variable selection strategy is even more deterministic than Novelty's; in particular, it is completely deterministic for $p \in \{0, 0.5, 1\}$.

Since the pure R-Novelty algorithm gets too easily stuck in local minima, a simple loop breaking strategy is used: every 100 steps, a variable is randomly chosen from the selected clause and flipped. As shown in [Hoos, 1998b; 1999a], this loop breaking strategy is generally not sufficient for effectively escaping from local minima and leaves R-Novelty essentially incomplete; as for Novelty, severe stagnation behaviour is observed in practice for some SAT instances [Hoos and Stützle, 2000a]. R-Novelty's performance is often, but not always, superior to Novelty's.

Extending R-Novelty with a random walk mechanism exactly analogous to the one used in Novelty$^+$ leads to the R-Novelty$^+$ algorithm [Hoos, 1998b; 1999a]. Like Novelty$^+$, R-Novelty$^+$ is provably PAC for $wp > 0$ and shows exponential RTDs for sufficiently high noise (instance specific) settings. Again, a small walk probability of $wp = 0.01$ appears to be generally sufficient for avoiding stagnation behaviour and for robustly achieving good performance in practice. R-Novelty$^+$'s performance for instances on which R-Novelty does not suffer from stagnation behaviour is very similar to R-Novelty's, There is some indication that R-Novelty and R-Novelty$^+$ do not reach the performance of Novelty$^+$ on several classes of structured SAT instances, including SAT-encoded hard graph colouring and planning problems [Hoos and Stützle, 2000a].

WalkSAT with Adaptive Noise

The noise parameter, *p*, which is common to all WalkSAT algorithms discussed here with the exception of WalkSAT/TABU (where the tabu tenure *tl* plays a similar role), has a major impact on the performance and run-time behaviour of the respective algorithm. For low noise settings, stagnation behaviour is typically observed, and as a consequence, using an appropriate *maxSteps* setting for the static restart mechanism becomes crucial for obtaining good performance [Hoos and Stützle, 2000a]. For sufficiently high noise settings, however, the *maxSteps* setting has typically little or no impact on the behaviour of the algorithm [Parkes and Walser, 1996; Hoos and Stützle, 1999], since the corresponding RTDs are closely approximated by exponential distribution. (There are exceptions to this general observation, including instances on which essentially incomplete WalkSAT variants show extreme stagnation behaviour as well as the irregular instances recently described by Hoos [Hoos, 2002a].) Fortunately, for many of the most prominent and best-performing WalkSAT algorithms, including WalkSAT/SKC, WalkSAT/TABU, Novelty$^+$, and R-Novelty$^+$, the noise settings required for reaching peak performance are generally high enough that the cutoff parameter, *maxSteps*, does not affect performance unless it is chosen too low, in which case performance is degraded. This leaves the noise setting, p, to be optimised in order to achieve maximal performance of these WalkSAT algorithms.

Unfortunately, finding the optimal noise setting is typically a difficult task. Because optimal noise settings appear to differ considerably depending on the given problem instance, this task often requires experience and substantial experimentation with various noise values [Hoos and Stützle, 2000a]. It has been shown that even relatively minor deviations from the optimal noise setting can lead to a substantial increase in the expected time for solving a given instance; and to make matters worse, the sensitivity of WalkSAT's performance w.r.t. the noise setting seems to increase with the size and hardness of the problem instance to be solved [Hoos, 2002b]. This complicates the use of WalkSAT for solving SAT instances as well as the evaluation, and hence the development, of new WalkSAT algorithms.

The key idea behind Adaptive WalkSAT [Hoos, 2002b] is to use high noise values only when they are needed to escape from stagnation situations in which the search procedure appears to make no further progress towards finding a solution. This idea is closely related to the motivation behind Reactive Tabu Search [Battiti and Tecchiolli, 1994] and Iterated Local Search [Lourenço *et al.*, 2002]. More precisely, Adaptive WalkSAT dynamically adjusts the noise setting p, *i.e.*, the probability for performing greedy steps, based on search progress, as reflected in the time elapsed since the last improvement in the evaluation function has been achieved. At the beginning of the search process, the search is maximally greedy ($p = 0$). This will typically lead to a series of rapid improvements in the evaluation function value, followed by stagnation (unless a solution to the given problem instance is found). In this situation, the noise value is increased. If this increase is not sufficient to escape from the stagnation situation, *i.e.*, if it does not lead to an improvement in evaluation function value within a certain number of steps, the noise value is further increased. Eventually, the noise value should be high for the search process to overcome the stagnation situation, at which point the noise can be gradually decreased until the next stagnation situation is detected or a solution to the given problem instance is found.

As an indicator for search stagnation, Adaptive WalkSAT uses a predicate that is true if and only if no improvement in evaluation function value has been observed over the last $\theta \cdot m$ search steps, where m is the number of clauses of the given problem instance and θ is a parameter. Every incremental increase in the noise value is realised as $p := p + (1 - p) \cdot \phi$. The decrements are defined as $p := p - p \cdot 2\phi$, where p is the noise level and ϕ is an additional parameter. The asymmetry between increases and decreases in the noise setting is motivated by the fact that detecting search stagnation is computationally more expensive than detecting search progress and by the observation that it is advantageous to approximate optimal noise levels from above rather than from below [Hoos, 2002b]. After the noise setting has been increased or decreased, the current evaluation function value is stored and becomes the basis for measuring improvement, and hence for detecting search stagnation. As a consequence, between increases in noise level there is always a phase during which the trajectory is monitored for search progress without further increasing the noise. No such delay is enforced between successive decreases in noise level.

It may be noted that the behaviour of the adaptive noise mechanism is controlled by two internal parameters, θ and ϕ. While one might assume that this merely replaced the problem of tuning one parameter, p, by the potentially more difficult problem of tuning these new parameters, it appears that the performance of Adaptive WalkSAT is much more robust w.r.t. to the settings of these parameters, than WalkSAT is w.r.t. to the noise setting. Using fixed settings of $\theta = 1/6$ and $\phi = 0.2$ for Adaptive Novelty$^+$ generally seems to result in similar performance as observed for Novelty$^+$ with approx. optimal, instance specific noise settings; in some cases, Adaptive Novelty$^+$ achieves significantly better performance than Novelty$^+$ approx. optimal static noise [Hoos, 2002b], making Adaptive Novelty$^+$ one of the best-performing and most robust SLS algorithms for SAT currently available.

6.4 Dynamic Local Search Algorithms for SAT

The first application of Dynamic Local Search to SAT was proposed around the same time as GWSAT. Since then, a number of DLS algorithms for SAT have been developped, the most recent of which achieve better performance than the best GSAT and WalkSAT variants for many types of SAT instances and can therefore be seen as the best performing SLS algorithms for SAT currently known.

Most DLS algorithms for SAT are based on variants of GSAT as their underlying local search procedure. Furthermore, the clauses of the given formula represent the solution components that are selectively penalised. In this section, we denote the penalty associated with clause i by $clp(i)$. Consistent with the general DLS algorithm from Section 2.2, typically a modified evaluation function of the form

$$g'(F,a) = g(F,a) + \sum\{clp(i) \mid \text{clause } i \text{ is unsatisfied by } a\}$$

is used within the local search procedure. Many DLS algorithms for SAT use the notion of clause weights $clw(i)$ instead of clause penalties, where

$$clw(i) = clp(i) + 1$$

and

$$g'(F,a) = \sum\{clw(i) \mid \text{clause } i \text{ is unsatisfied by } a\};$$

for

$$g(F,a) = \#\{i \mid \text{clause } i \text{ is unsatisfied by } a\},$$

the standard evaluation function used by most SLS algorithms for SAT, both definitions of $g'(F,a)$ are equivalent. The major differences between DLS algorithms for SAT are in the details of the local search procedure and in the scheme used for updating the clause penalties or weights.

Most DLS algorithms for SAT perform excellently in terms of the number of variable flips required for finding a model of a given formula. However, the time complexity and frequency of the weight updates is typically rather high, which makes it difficult for DLS algorithms to reach or exceed the time performance of the best-performing WalkSAT variants. Unfortunately, the run-time behaviour of DLS algorithms for SAT has not been as thoroughly investigated as that of GSAT and WalkSAT algorithms. In particular, little is know about these algorithms in terms of their asymptotic run-time behaviour, search stagnation, and RTD characterisations.

GSAT with Clause Weights

This early DLS algorithm for SAT is based on the observation that when applied to certain types of structured SAT instances, basic GSAT often finds the same set of clauses unsatisfied at the end of a run [Selman and Kautz, 1993]. In this GSAT variant, weights are associated with each clause. These weights are initially set to one; before each restart, the weights of all currently unsatisfied clauses are increased by $\delta = 1$. The underlying local search procedure is a variant of basic GSAT that uses the modified evaluation function $g'(F, a)$ introduced above. It may be noted that since basic GSAT is a best-improvement search method, for sufficiently high *maxSteps* setting, this local search procedure will terminate in or very close to a local minima region of the underlying search space. Different from the other DLS methods discussed in this section, GSAT with Clause Weights begins its next local search phase from a randomly selected variable assignment. (A further extension, called “Averaging In”, uses a modified search initialisation that introduces a bias towards the best candidate solutions reached in previous local search phases [Selman and Kautz, 1993].)

GSAT with Clause Weights performs substantially better than basic GSAT on various classes of structured SAT instances, including SAT-encoded asymmetrical graph colouring problems; there is also some indication that by using the same clause weighting mechanism with GWSAT, further performance improvements can be achieved [Selman and Kautz, 1993]. Today, since its performance is not competitive with any of the more recent DLS algorithms for SAT presented in the following, GSAT with Clause Weights is mainly of historical interest.

Several variants of the Breakout algorithm have been studied by Cha and Iwama [Cha and Iwama, 1995]. In particular, they tested a variant that uses weigthed GSAT as the underlying local search procedure, but different from GSAT with Clause Weights, their algorithm performs weight updates whenever a local minimum of the modified evaluation function is encountered and, in its basic form, it does not perform restarts. This algorithm appears to perform substantially better than GSAT and GWSAT when applied to a class of randomly generated SAT instances that have only a single model [Asahiro *et al.*, 1996]. (These instances, however, are not intrinsically hard because they can be solved by polynomial simplifications and hence they are only of limited use as benchmark problems [Hoos and Stützle, 2000b].)

There is no evidence that this variant performs better than the original GSAT with Clause Weights algorithm.

Cha and Iwama also investigated slight variations of the weight update scheme as well as combinations of their basic algorithm with static restarts and a simple tabu strategy that, different from GSAT/TABU or WalkSAT/TABU, associates tabu status with the most recently visited variable assignments rather than with recently flipped variables [Cha and Iwama, 1995]. From their limited empirical results it appears that none of these variations achieves significant performance improvements over their basic variant of GSAT with Clause Weights.

Methods using Rapid Weight Adjustments

Frank introduced several variants of GSAT with Clause Weights that perform weight updates after each local search step [Frank, 1996; 1997]. The underlying idea is that GSAT should benefit from discovering which clauses are most difficult to satisfy relative to recent assignments. The most basic of these variants, called WGSAT, uses the same weight initialisation and update procedure as GSAT with Clause Weights, but performs only a single weighted GSAT step before updating the clause weights. On hard Random-3-SAT instances, WGSAT achieves a significantly improved performance over HSAT (and hence, basic GSAT) when measuring run-time in terms variable flips required for finding a solution [Frank, 1996; 1997]. When comparing CPU times however, it appears that due to the computational overhead caused by the frequent weight updates, WGSAT's performance cannot reach that of HSAT or GWSAT.

A modification of this algorithm, called UGSAT, uses a best-improvement local search like weighted GSAT, but restricts the neighbourhood considered in each search steps to the set of variables appearing in currently unsatisfied clauses [Frank, 1996]. While this leads to considerable speedups for naive implementations of the underlying local search procedure, the difference to efficient implementations is likely to be insufficient to render UGSAT competitive with HSAT or GWSAT.

Another variant of WGSAT implements a uniform decay of clause weights over time. The underlying idea is that the relative importance of clauses w.r.t. their satisfaction status can change during the search, and hence a mechanism is needed that focusses the weighted search on the most recently unsatisfied clauses. In WGSAT with Decay, this idea is implemented by uniformly decaying all clause weights in each weight update phase before the weights of the currently unsatisfied clauses are increased; this decay is performed according to the formula $clw(i) \leftarrow clw(i) \cdot \rho$, where the decay rate ρ (with $0 < \rho < 1$) is a parameter of the algorithm [Frank, 1997]. Empirical results suggest that on larger instances from the phase transition region of Uniform Random-3-SAT, using this decay mechanism slightly improves the performance of WGSAT when measured in terms of variable flips; this improvement, however, appears to be insufficient to amortise the added time complexity of the frequent weight update steps. Nevertheless, as we will see later in this section, similar mechanisms for focussing the search on recently unsatisfied clauses play a crucial role in state-of-the-art DLS algorithms for SAT.

Guided Local Search (GLS)

This relatively recent DLS algorithm has been applied to a number of combinatorial problems [Voudouris, 1997; Voudouris and Tsang, 1999] Guided Local Search for SAT (GLSSAT) [Mills and Tsang, 1999; 2000] is based on a local search algorithm that, similar to HSAT, Novelty, and R-Novelty, implements a bias towards flipping variables whose values was not changed for a while. More precisely, in each local search step, from the set of all variables that, when flipped, would lead to a strict decrease in the total penalty of unsatisfied clauses, the one whose last flip has occurred least recently is flipped. If no such strictly improving variable exists, the same selection is made from the set of all variables that, when flipped, do not cause an increase in the evaluation function value.[1] The subsidiary local search procedure terminates when a satisfying assignment is found, or after a fixed number $smax$ of consecutive non-improving flips have been made without an improving flip becoming available.

Before the actual search begins, GLSSAT performs a complete pass of unit propagation in order to simplify the given formula. Then, all clause penalties are initialised to zero, and the search starts from a variable assignment that is chosen uniformly at random.

[1] Interestingly, it has been shown that using simple random selection from the same sets results in only slightly worse performance of GLSSAT.

After each local search phase, the penalties of all clauses with maximal utilities are incremented by $\delta = 1$, where the utility of a clause i under assignment a is defined as $util(a, i) = 1/(1 + clp(i))$ if clause i is unsatisfied under x and zero otherwise. Note that this corresponds to incrementing the smallest clause penalties occurring in currently unsatisfied clauses. An important extension of GLSSAT uses an additional mechanism for bounding the range of the clause penalties: If after updating the clause penalties, the maximum penalty exceeds a given threshold, $pmax$, all clause penalties are uniformly decayed by multiplying them with a factor $pdecay$. This clause penalty decay mechanism has a substantial impact on the performance of GLSSAT and significantly improves the algorithm's efficacy in solving large and hard structured instances. A similar modification of GLSSAT, called GLSSAT2, was used in another study [Mills and Tsang, 2000]; in this variant, all clause penalties are multiplied by a factor $pdecay = 0.8$ after every 200 penalty updates.

GLSSAT achieves better performance than WalkSAT/SKC on some widely used benchmark instances when measuring run-time in terms of variable flips, but in many cases WalkSAT/SKC is superior in terms of CPU time [Mills and Tsang, 2000]. There are some hard structured SAT instances, however, for which GLSSAT2 appears to perform significantly better than WalkSAT/SKC. There is some indirect evidence that GLSSAT is generally outperformed by the most recent DLS algorithms for SAT, such as ESG and SAPS (see below).

The Discrete Lagrangian Method (DLM)

The basic DLM algorithm for SAT [Shang and Wah, 1998] is motivated by the theory of Lagrange multipliers for continuous optimisation. Basic DLM is a DLS algorithm based on GSAT/TABU with clause weights (which flips non-tabu variables maximising the decrease in the total weight of all unsatisfied clauses) as its underlying local search procedure. This subsidiary local search is terminated when an assignment is reached for which the number of neighbouring assignments with larger or equal evaluation function value exceeds a given threshold θ_1. After each local search phase, the penalties for all unsatisfied clauses are increased by $\delta^+ = 1$; in order to bound the range of the clause penalties, additionally all penalties are reduced by $\delta^- = 1$ after every θ_2 local search phases. Before the actual search begins, DLM simplifies the given formula by performing a complete pass of unit propagation. As usual, all clause penalties are initialised to zero, and the search process starts from a variable assignment that is chosen uniformly at random.

This basic DLM algorithm has been extended in various ways. DLM-99-SAT [Wu and Wah, 1999] uses an additional mechanism for escaping more effectively from local minima of the evaluation function. The idea behind this mechanism is to identify clauses that are frequently unsatisfied in local minima and to additionally increase their penalties. This is achieved by means of temporary clause penalties t_i that are initialised at zero and increased by $\delta_w = 1$ for all unsatisfied clauses, whenever a local minimum is encountered. After each regular clause penalty update, if the ratio between the maximal t_i and average t_i over all clauses exceeds a threshold θ_3, the regular penalty of the clause with the largest t_i is increased by $\delta_s = 1$.[2]

A different extension of DLM-2000-SAT uses a long-term memory mechanism for preventing the search process from getting stuck repeatedly in certain attractive non-solution areas of the search space. This is implemented by using a list of previously visited assignments and by adding an additional distance penalty to the evaluation function for assignments that are close to the elements of this list. More precisely, during the search process, every w_s variable flips, the current variable assignment is added to a fixed-length queue. Using the assignments a_j in this queue, a distance term for a given variable assignment a is computed as $d = \sum_j \min(\theta_t, hd(a, a_j))$, where $hd(a, a_j)$ is the Hamming distance (*i.e.*, the number of variables assigned different values) between assignments a and a_j. The evaluation function used in the subsidiary local search procedure is then extended to $g'(F, a) = g(F, a) + \sum\{clw(i) \mid \text{clause } i \text{ is unsatisfied by } a\} - d$. Note that by using a bound $\theta_t \ll n$ on the distance contribution from each assignment a_j, the impact of this mechanism on the search process is fairly localised.

DLM-99-SAT shows substantially better performance than the basic DLM algorithm, particularly on large and structured SAT instances. DLM-2000-SAT, the most recent DLM variant, typically seems to perform better than DLM-99-SAT as well as WalkSAT/SKC; until very recently, this Dynamic Local Search algorithm was one of the best known SLS algorithms for SAT.

[2]In another variant, only the t_i of currently unsatisfied clauses are considered in computing the ratio and determining the clause penalty that receives the additional increase.

The Exponentiated Subgradient Algorithm (ESG)

The Exponentiated Subgradient (ESG) algorithm [Schuurmans *et al.*, 2001] is motivated by subgradient optimisation, a well-known method for minimising Lagrangian functions that is often used for generating good lower bounds for branch and bound techniques or as a heuristic in local search algorithms.

ESG starts its search from a randomly selected variable assignment after initialising all clause weights to one. As its underlying local search procedure, ESG uses a best improvement search method that can be seen as a simple variant of weighted GSAT: In each local search step, the variable to be flipped is selected uniformly at random from the set of all variables that appear in currently unsatisfied clauses and whose flip leads to a maximal reduction in the total weight of unsatisfied clauses. When reaching a local minimum state, *i.e.*, an assignment in which flipping any variable that appears in an unsatisfied clause would not lead to a decrease in the total weight of unsatisfied clauses, with probability η, the search is continued by flipping a variable that is uniformly chosen at random from the set of all variables appearing in unsatisfied clauses. Otherwise, the local search phase is terminated.

After each local search phase, the clause weights are updated. This involves two stages: First, the weights of all clauses are multiplied by a factor depending on their satisfaction status; weights of satisfied clauses are multiplied by α_{sat}, weights of unsatisfied clauses by α_{unsat} (scaling stage). Then, all clause weights are smoothed using the formula $clw(i) \leftarrow clw(i) \cdot \rho + (1 - \rho) \cdot \bar{w}$ (smoothing stage), where $\bar{w}$ is the average of all clause weights after scaling, and the parameter ρ has a fixed value between zero and one. The algorithm terminates when a satisfying assignment for F has been found or when a maximal number of iterations has been performed.

In a straight-forward implementation of ESG, the weight update steps are computationally much more expensive than the weighted search steps, whose cost is determined by the underlying basic local search procedure. Each weight update step requires accessing all clause weights, while a weighted search step only needs to access the weights of the critical clauses, *i.e.*, clauses that can change their satisfaction status when a variable appearing in a currently unsatisfied clause is flipped.[3] Typically, for the major part of

[3]The complexity of all other operations is dominated by these operations.

the search only few clauses are unsatisfied; hence, only a small subset of the clauses is critical, rendering the weighted search steps computationally cheaper than weight updates.

If weight updates would typically occur very infrequently as compared to weighted search steps, the relatively high complexity of the weight update steps might not have a significant effect on the performance of the algorithm. However, experimental evidence indicates that the fraction of weighting steps performed by ESG is quite high; it ranges from around 7% (for SAT encodings of large flat graph colouring problems) to more than 40% percent (for SAT-encoded all-interval-series problems).

Efficient implementations of ESG therefore critically depend on additional techniques in order to achieve the competitive performance results reported in [Schuurmans *et al.,* 2001]. The most recent publically available ESG-SAT software by Southey and Schuurmans (Version 1.4), for instance, uses $\alpha_{sat} = 1$ (which avoids the effort of scaling satisfied clauses), replaces $\bar{w}$ by 1 in the smoothing step, and utilises a lazy weight update technique which updates clause weights only when they are needed.

When measuring run-time in terms of search steps, ESG typically performs substantially better than state-of-the-art WalkSAT variants, such as Novelty$^+$. In terms of CPU-time, even the highly optimised ESG-SAT implementation by Southey and Schuurmans does not always reach the performance of Novelty$^+$. Compared to DLM-2000-SAT, ESG-SAT typically requires fewer steps for finding a model of a given formula but in terms of CPU-time, both algorithms show very similar performance [Schuurmans *et al.,* 2001; Hutter *et al.*, 2002].

Scaling and Probabilistic Smoothing (SAPS)

The SAPS algorithm [Hutter *et al.*, 2002] can be seen as variant of ESG that uses a modified weight update scheme, in which the scaling stage is restricted to the weights of currently unsatisfied clauses, and smoothing is only performed with a certain probability P_{smooth}. Note that restricting the scaling operation to the weights of unsatisfied clauses ($\alpha_{sat} = 1$) does not affect the variable selection in the weighted search phase, since rescaling all clause weights by a constant factor does not affect the variable selection mechanism. (Southey's and Schuurmans' efficient ESG implementation also makes use of this fact.) This reduces the complexity of the scaling step from $\Theta(|C(F)|)$ to $\Theta(|C_{unsat}(F,a)|)$, where $C(F)$ is the set of clauses in the given CNF formula F and $C_{unsat}(F,a)$ is the set of clauses in F that are unsatisfied under assignment a.

After a short initial search phase, typically only few clauses remain unsatisfied such that $|C_{unsat}(F,a)|$ becomes rather small compared to $|C(F)|$; this effect seems to be more pronounced for larger SAT instances with many clauses. The smoothing step, however, has complexity $\Theta(|C(F)|)$, and now dominates the complexity of the weight update. Therefore, by applying the expensive smoothing operation only occasionally, the time complexity of the weight update procedure can be substantially reduced. It has been shown experimentally that this does not have a detrimental effect on the performance of the algorithm in terms of the number of weighted search steps required for solving a given instance [Hutter *et al.*, 2002]. By having the weight update procedure perform smoothing of all clause weights (using the same formula as shown in our description of ESG above) only with a probability $P_{smooth} < 1$, compared to ESG the time complexity of a weight update is reduced from $\Theta(|C(F)| + |C_{unsat}(F,a)|)$ to $\Theta(P_{smooth} \cdot |C(F)| + |C_{unsat}(F,a)|)$. As a result, the amortised cost of smoothing no longer dominates the algorithm's run-time. Performing the smoothing probabilistically rather than a deterministically after a fixed number of steps (as the occasional clause weight reduction in DLM), has the theoretical advantage of preventing the algorithm from getting trapped in the same kind of cyclic behaviour that renders R-Novelty essentially incomplete.

The SAPS algorithm as described here does not require additional implementation tricks other than the standard mechanism for efficiently accessing critical clauses that is used in all efficient implementations of SLS algorithms for SAT. Compared to ESG, SAPS has similar performance in terms of the number of variable flips required for finding a model of the given formula, but SAPS shows generally significantly improved time performance over ESG, DLM-2000-SAT, and the best known WalkSAT variants [Hutter *et al.*, 2002]. However, there are some cases (in particular, hard and large SAT encoded graph colouring instances), for which SAPS does not reach the performance of Novelty$^+$.

A reactive variant of SAPS (RSAPS), which automatically adjusts the smoothing probability P_{smooth} during the search using a very similar mechanism as Adaptive WalkSAT, sometimes achieves significantly better perfor-

```
procedure UpdateClauseWeights($F$, $a$; $\alpha$, $\rho$, $P_{smooth}$)
    input:
        propositional formula $F$, variable assignment $a$;
        scaling factor $\alpha$, smoothing factor $\rho$, smoothing probability $P_{smooth}$
    $C$ = $\{$clauses of $F\}$
    $C_u$ = $\{c \in C \mid c$ is unsatisfied under $a\}$
    for each $i$ s.t. $c_i \in C_u$ do
        $clw(i) := clw(i) \times \alpha$
    end
    with probability $P_{smooth}$ do
        for each $i$ s.t. $c_i \in C$ do
            $clw(i) := clw(i) \times \rho + (1 - \rho) \times \bar{w}$
        end
    end
end
```

Figure 6.5: The SAPS weight update procedure; $\bar{w}$ is the average over all clause weights.

mance than SAPS [Hutter *et al.*, 2002]. However, different from Adaptive WalkSAT, RSAPS still has other parameters, in particular ρ, that need to be tuned manually in order to achieve optimal performance.

6.5 Constraint Satisfaction Problems

An instance of a Constraint Satisfaction Problem (CSP) is defined by a set of variables, a set of possible values (or domain) for each variable, and a set of constraining conditions (constraints) involving one or more of the variables. The Constraint Satisfaction Problem is to decide for a given CSP instance whether all variables can be assigned values from their respective domains such that all constraints are simultaneously satisfied. Depending on whether the variable domains are discrete or continuous, finite or infinite, different types of CSP instances and respective subclasses of CSP can be distinguished. Here, we restrict our attention to the *finite discrete CSP*, a widely studied type of CSP with many practical applications.

Definition 6.2 (Finite discrete CSP)

A *CSP instance* is a triple $P = (V, \mathcal{D}, \mathcal{C})$, where $V = \{x_1, \ldots, x_n\}$ is a finite set of n variables, $\mathcal{D}$ is a function that maps each variable x_i to the set D_i of possible values it can take (D_i is called the *domain of* x_i), and $\mathcal{C} = \{C_1, \ldots, C_m\}$ is a finite set of constraints. Each constraint C_j is a relation over an ordered set *Var*(C_j) of variables from V, *i.e.*, for *Var*$(C_j) = (y_1, \ldots, y_k)$, $C_j \subseteq D(y_1) \times \cdots \times D(y_k)$. The elements of the set C_j are referred to as *satisfying tuples of* C_j, and k is called the *arity of the constraint* C_j.

P is a *finite discrete CSP instance* if all variables in P have discrete and finite domains.

A *variable assignment* of P is a mapping $a : V \mapsto \bigcup\{\mathcal{D}\}$ that assigns to each variable $x \in V$ a value from its domain $D(x)$. Let *Assign*(P) denote the set of all possible variable assignments for P; then a variable assignment $a \in$ *Assign*(P) is a *solution of* P if and only if it simultaneously satisfies all constraints in $\mathcal{C}$, *i.e.*, if for all $C_j \in \mathcal{C}$ with, say, *Var*$(C_j) = (y_1, \ldots, y_k)$ the assignment a maps $y_1, \ldots, y_k$ to values $v_1, \ldots, v_k$ such that $(v_1, \ldots, v_k) \in C_j$.

CSP instances for which at least one solution exists are also called *consistent*, while instances that do not have any solutions are called *inconsistent*.

Finite discrete CSP is the problem of deciding whether a given finite discrete CSP instance P is consistent. $\square$

Remark: In many cases, the constraint relations involved in CSP instances can be represented more compactly by using standard mathematical relations, such as $=, \neq, <, \geq, >, \leq$. In other cases, a more compact representation of a given constraint C_j is obtained by explicitly listing the complement of the set of satisfying tuples, *i.e.*, the set of unsatisfying tuples of C_j.

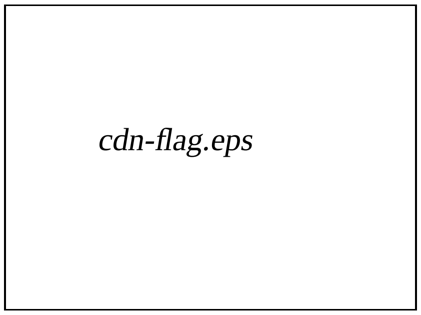


Figure 6.6: A simple CSP instance: The problem of colouring the Canadian flag (see text for details).

Example 6.1: The Canadian Flag Problem

Let us consider the problem of colouring the Canadian flag by assigning colours red (r) and white (w) to the four fields L, C, R, M in such a way that any two neighbouring fields are coloured differently (see Figure 6.6). This problem can be formulated as a binary CSP instance as follows:

$$\begin{array}{rcl}
V & = & \{L, C, R, M\} \\
\mathcal{D}(L) & = & \mathcal{D}(C) = \mathcal{D}(R) = \mathcal{D}(M) = \{r, w\} \\
\mathcal{C} & = & \{C_1, C_2, C_3\} \\
\text{with} & & \mathit{Var}(C_1) = (L, C) \\
 & & \mathit{Var}(C_2) = (C, M) \\
 & & \mathit{Var}(C_3) = (C, R) \\
\text{and} & & C_1 = C_2 = C_3 = \{(r, w), (w, r)\}
\end{array}$$

There are two solutions to this CSP instance, one assigns red to M, L, R and white to C, while the other assigns white to M, L, R and red to C. By adding a forth, unary constraint to $\mathcal{C}$ that forces M to be coloured red, the instance can be modified such that only the solution corresponding to the correct colouring of the Canadian flag remains.

This simple CSP instance is an example of a Map Colouring Problem, which in turn can be seen as a special case of the Graph Colouring Problem (GCP). GCP is an important subclass of CSP in which the objective is to colour the vertices of a given graph in such a way that two vertices connected by an edge are never assigned the same colour. The Graph Colouring Problem will be covered in more detail in Chapter 10.

Like SAT, finite discrete CSP is an $\mathcal{NP}$-complete combinatorial problem. This can be proven rather easily based on the following close relationship between propositional SAT and finite discrete CSP: Any instance of SAT (for CNF formulae) can be seen as a finite discrete CSP instance where all the domains contain only the truth values $\top, \bot$ and each constraint contains exactly all the satisfying assignments of one particular clause of the given CNF formula F. Vice versa, as we will show in the next section, any finite discrete CSP instance can be directly transformed into a SAT instance.

Encoding CSP Instances into SAT

CSP instances can be encoded into SAT in a rather straight-forward way. The basic idea is to use propositional variables to represent the assignment of values to single CSP variables and clauses to express the constraint relations [de Kleer, 1989]. For the sake of simplicity, we assume in the following, without loss of generality, that domains of all variables are equal to $\mathbb{Z}_k = \{0, 1, \ldots, k-1\}$, where k is an arbitrary positive integer. Furthermore, we use $\sigma(C_j)$ to denote the arity of a constraint C_j, *i.e.*, the number of variables involved in C_j.

Given a finite discrete CSP instance $P = (V, \mathcal{D}, \mathcal{C})$ with $V = \{x_1, \ldots, x_n\}$, $\mathcal{D}(x) = \mathbb{Z}_k$ for all $x \in V$, and $\mathcal{C} = \{C_1, \ldots, C_m\}$, a very natural SAT encoding is based on propositional variables $c_{i,\nu}$ that, if assigned the value $\top$, represent the assignment $x_i = v$, where $v \in \mathcal{D}(x_i)$. P can then be represented by a CNF formula comprising the following sets of clauses:

$$
\begin{array}{lll}
(1) & \neg c_{i,v_1} \vee \neg c_{i,v_2} & (1 \leq i \leq n; v_1, v_2 \in \mathbb{Z}_k; v_1 < v_2) \\
(2) & c_{i,1} \vee c_{i,2} \vee \ldots \vee c_{i,k} & (1 \leq i \leq n) \\
(3) & \neg c_{i_1,v_1} \vee \neg c_{i_2,v_2} \vee \ldots \vee \neg c_{i_s,v_s} & (x_{i_1} = \nu_1; x_{i_2} = \nu_2; \ldots; x_{i_s} = \nu_s) \\
 & & \text{violates some constraint } C_j \in \mathcal{C} \\
 & & \text{with } \sigma(C_j) = s)
\end{array}
$$

Intuitively, these clause sets ensure that each constraint variable is assigned exactly one value from its domain (lines 1 and 2) and that this assignment is compatible with all constraints (line 3). The number of propositional variables required for encoding a given CSP instance is linear in the number of constraint variables and their domain sizes while the number of clauses is at least linear in the number of constraint variables and depends critically on the domain sizes and the arity of the constraints. This encoding is frequently used in the context of translations of combinatorial problems into SAT that use CSP as an intermediate domain (*cf.* Section 6.1). It is known as the *sparse encoding*, because it generates large SAT instances whose models have only a small fraction of the propositional variables set to $\top$. (In the literature, this encoding has also been referred to as the unary transform or direct encoding.)

By using an alternative SAT encoding of CSP instances, the number of propositional variables required for encoding a given CSP instance can be significantly reduced compared to the sparse encoding. The *compact encoding* (in the literature also referred to as the binary transform or log encoding) is based on the idea of representing the value ν assigned to any constraint variable x_i by a group of $\lceil \log_2 k \rceil$ propositional variables $c_{i,j}$ using a binary encoding of ν [Iwama and Miyazaki, 1994; Hoos, 1998b; 1999b]. This leads to a CNF formula with $n \cdot \lceil \log_2 k \rceil$ propositional variables; particularly for large domain sizes k, this can be a substantial reduction compared to the $n \cdot k$ propositional variables required by the sparse encoding. The number of clauses, however, is similar for both encodings, since in either case the same number of clauses is needed for representing the constraint relations (these clauses typically dominate the overall number of clauses). Although the SAT instances generated by the compact encoding have search spaces that are substantially smaller than those obtained from the sparse encoding, they appear to be much harder to solve using standard SLS algorithms for SAT [Hoos, 1998b; 1999b; Frisch and Peugniez, 2001b].

CSP Simplification and Local Consistency Techniques

Similar to the case of SAT, native CSP instances can often be substantially reduced in size and complexity by applying polynomial-time simplification methods. Also known as local consistency techniques, these methods are transformations that are applied to (local) subsets of a given CSP instance P [Mackworth, 1977; Debruyne and Bessière, 2001]. Local consistency techniques can reduce the effective domains of CSP variables by eliminating values that cannot occur in any solution.

One of the most prominent simplification techniques for CSP is the enforcement of arc consistency. A given CSP instance P is made arc consistent w.r.t. to one of its constraints, C, by removing any value d from the domain of any variable x involved in C if there exists no CSP assignment that satisfies C, *i.e.*, no tuple $t \in C$, for which x has value d. A CSP instance P is arc consistent if it is arc consistent w.r.t. all of its constraints. For binary CSP instances with e constraints (there is a maximal number of $n(n-1)/2$ constraints, where n is the number of variables) and maximal domain size k, the best known algorithms for enforcing arc consistency have a time complexity of $O(ek^2)$ and a space complexity of $O(ek)$ [Bessière *et al.*, 1999]. A number of further local consistency techniques is presented in [Debruyne and Bessière, 2001].

Combined with backtracking mechanisms, simplification methods such as forward checking or enforcing arc consistency play a crucial role in systematic search algorithms for CSP [Haralick and Elliot, 1980; Grant and Smith, 1996]. They can also be used as preprocessing techniques before applying SLS-based, incomplete CSP solvers. The high computational cost of enforcing higher levels of local consistency, such as path consistency, are often not amortised by the reduced run-times of CSP solvers that are subsequently applied to the resulting CSP instances. One method for improving this situation is to apply the corresponding local consistency methods to heuristically selected subsets of a given CSP only [Kask and Dechter, 1995].

Some Prominent Benchmark Instances for CSP

There are numerous types of CSP instances that have been routinely used in the literature on CSP. Many studies have focused on a particular class of randomly generated CSP instances with binary constraint relations which we call *Uniform Random Binary CSP*. Besides the number of CSP variables and the size of the variable domains, this problem distribution is characterised by two parameters, the *constraint graph density* α and the *constraint tightness* β; α specifies the probability that a constraint relation exists between an arbitrary pair of CSP variables and β is the expected fraction of value pairs that are allowed by a given constraint relation between two variables. For this class of CSP instances, a solubility phase transition phenomenon with an associated peak in hardness, similar to the one for Uniform Random-3-

SAT, has been observed [Smith, 1994] and test–sets of hard instances can be obtained for specific combinations of α and β values.

Another widely used class of CSP instances stem from the Graph Colouring Problem (see Example 6.1), which can be seen as a special case of Finite Discrete CSP in which all variables have the same domain, and all constraint relations are binary, allowing a pair of values (x, y) if and only if $x \neq y$ (inequality constraint). The Graph Colouring Problem and specific instance classes are discussed in more detail in Chapter 10. Graph colouring instances with three colours are amongst the most commonly used benchmark instances for CSP.

An prominent special case of the Graph Colouring Problem is the Quasigroup Completion Problem (QCP), which is derived from the following Quasigroup Problem or Latin Square Problem: Given an $n \times n$ quadratic grid and n colours, the objective it to assign a colour to each grid cell in such a way that every row and column contains all n colours. In the Quasigroup Completion Problem, the objective is to decide whether a partial solution of the Quasigroup Problem, *i.e.*, an incomplete assignment of colours to the given grid such that no two cells in the same row or column have the same colour, can be extended into a complete solution by assigning colours to the remaining cells. In the CSP formulation, the pre-assigned cells can be easily represented by unary constraints. The QCP is known to be $\mathcal{NP}$-complete [**?**], and a phase-transition phenomenon with an associated peak in hardness has been observed [Gomes and Selman, 1997a]. It has applications in experimental design, scheduling, and timetabling.

The n-Queens Problem is another prominent CSP; it can be seen as a generalisation of the problem of placing eight queens on a chessboard such that no queen is threatened by any of the other queens. This is achieved by distributing the queens in such a way that no row, column, or diagonal has more than a single queen on it. This 8-Queens Problem can be represented as a CSP instance with eight variables and 28 binary constraints. In the n-Queens Problem, the objective is to place n queens on an $n \times n$ board subject to the same constraints.

Most of the work on CSP has been focused on binary CSP; One of the reasons for this is that any non-binary CSP instance can be transformed into as binary CSP instance in a rather straight-forward way [Dechter and Pearl, 1989; Rossi *et al.*, 1990; Bacchus *et al.*, 2002]. Another reason lies in the fact that algorithms restricted to binary CSP are typically easier to implement than general CSP solvers.

There are numerous other classes of CSP instances, including CSP encodings of the real-world problems mentioned in Section 6.1. Some application relevant problems include frequency assignment in radio networks, scheduling problems, vehicle routing and many more. A description of many of the different types of CSPs can be found at CSPLIB, a benchmark library for constraints which is accessible at `www.csplib.org`.

6.6 SLS Algorithms for CSPs

Because of the close relationship between CSP and SAT, the respective SLS algorithms for solving these problems are quite similar; historically, there has been significant cross-fertilisation between both domains in terms of SLS algorithm design and development. We distinguish three types of SLS techniques for solving CSPs: SLS algorithms for SAT applied to SAT-encoded CSP instances; generalisations of SLS algorithms for SAT; and native SLS algorithms for CSPs. In the following, we will discuss each of these approaches in more detail and present some of the most prominent and best performing SLS algorithms for CSPs.

The "Encode and Solve as SAT" Approach

In principle, any CSP instance P can be solved by encoding it into SAT and subsequently applying standard SAT solvers to determine the satisfiability of the resulting CNF formula F. If P is soluble, its solutions can be determined from the models of F. By using either of the two SAT encodings of CSPs discussed in Section 6.5, encoding CSP instances as well as decoding their solutions are efficient processes and the resulting SAT instances are typically not prohibitively large compared to the original CSP instances.

The main advantage of this approach lies in the fact that it allows the use of highly optimised and efficiently implemented "off-the-shelve" SAT solvers. Besides the SLS algorithms described earlier in this chapter, this includes competitive systematic SAT solvers and other state-of-the-art SAT algorithms (see Section 6.7 for references). Furthermore, standard polynomial preprocessing techniques for SAT can be used to simplify SAT-encoded CSP instances prior to applying a SAT solver. CSP preprocessing techniques, such as efficiently computable forms of k-consistency, can be applied before encoding a CSP instance into SAT. However, one potentially major disadvantage of the "encode and solve as SAT" approach may arise from the inability of standard SAT algorithms to exploit the structure present in given CSP instances.

There is some indication that by using the sparse encoding and high-performing SAT algorithms, such as Novelty or Novelty$^+$, competitive performance compared to state-of-the-art SLS algorithms for CSP, such as Galinier and Hao's Tabu Search algorithm (which will be described later in this section), can be obtained for Uniform Random Binary CSP instances [Hoos, 1998a; 1999b]. Similar results appear to hold for graph colouring instances, but there is some evidence that native CSP algorithms might achieve superior performance for random instances with large variable domains [Frisch and Peugniez, 2001b]. Interestingly, when using the compact encoding, SLS algorithms for SAT show substantially weaker performance; this performance difference appears to be caused by aspects of the search space structure induced by the respective encodings; in particular it has been shown that applied to the same CSP instances, the compact encoding generates search spaces with substantially higher local minima branching (see Chapter 5) than the sparse encoding [Hoos, 1998a; 1999b].

Clearly, the "Encode and Solve as SAT" approach is not limited to CSP, but can in principle be applied to any $\mathcal{NP}$-complete problem. For CSP, this approach is particularly attractive, since as a result of the close relationship between SAT and CSP, the encoding of CSP instances into SAT is conceptually simple and very efficiently computable in practice. Whether or not it can achieve competitive performance compared to the best native CSP algorithms, particularly when applied to structured CSP instances, is somewhat unclear at the present time and needs to be further investigated.

Pseudo-Boolean CSP and WSAT($\mathcal{PB}$)

An alternative to the "encode and solve as SAT" approach discussed in the previous section is to extend high performing SLS algorithms for SAT to more general subclasses of CSP. One such generalisation of SAT is obtained by maintaining the restriction to Boolean variables while allowing constraints that are more expressive than CNF clauses. In Pseudo-Boolean CSP, also known as (linear) Pseudo-Boolean Programming, all variables have Boolean values represented by integers zero and one, and the constraints between variables l_i are of the form

$$\sum_i a_i \cdot l_i \sim d,$$

where the a_i as well as d are integers and $\sim$ can be any comparison operator from the set $\{=, \leq, <, \geq, >\}$. It may be noted that without loss of generality, this can be restricted to constraints that only use the comparison operator $\leq$, since any constraint using one of the other operators may be equivalently expressed as a "$\leq$" constraint except for equality constraints, which can be modelled by two "$\leq$" constraints. Pseudo-Boolean constraints are more expressive than CNF clauses because any CNF clause can be expressed by a single pseudo-Boolean constraint, but pseudo-Boolean constraints can generally not be captured by a single clause. From an Operations Research point of view, Pseudo-Boolean CSP can be seen as a special case of Integer Programming (IP) with linear constraints and variable domains restricted to $\{0, 1\}$ [Nemhauser and Wolsey, 1988].

Example 6.2: Pseudo-Boolean Constraints

As an example for a Pseudo-Boolean constraint between three variables y_1, y_2, y_3 with domain$\{0, 1\}$, consider the inequality $y_1 + y_2 - y_3 \geq 0$. This constraint is equivalent to $y_1 + y_2 + (1 - y_3) \geq 1$, and hence to the CNF clause $x_1 \vee x_2 \vee \neg x_3$.

The following constraint limits the number of variables that are assigned the value one to a maximum of k:

$$y_1 + \ldots + y_n \leq k$$

Note that in order to express this constraint by a CNF formula, $\binom{n}{k}$ clauses of size $k + 1$ each are required; these encode the condition that for every possible subset of $k + 1$ variables, at least one variable needs to be assigned the value false.

There are relatively few SLS algorithms for Pseudo-Boolean CSP [Abramson *et al.*, 1996; Connolly, 1992; Walser, 1997; kketangen, 2002]. Among these, Walser's WSAT($\mathcal{PB}$) algorithm is of particular interest, since it is based on a direct generalisation of the WalkSAT architecture to Pseudo-Boolean CSP. The WSAT($\mathcal{PB}$) algorithm follows the WalkSAT framework as presented in Section 6.3, but uses a generalised evaluation function and variable selection mechanism. The evaluation function is based on the notion of *net integer distance* of a constraint from being satisfied. More precisely, for each constraint c, let $d(a, c)$ denote the minimal integer difference between any value of the right hand side of the corresponding inequality (or equality) that satisfies c and the value under assignment a; the evaluation function value of assignment a is then defined as the sum of the $d(a, c)$ values over all constraints unsatisfied in a. As in the SAT case, an assignment that satisfies all constraints has an evaluation function value of zero.

Based on this evaluation function, WSAT($\mathcal{PB}$) uses a modified version of the WalkSAT variable selection strategy to determine the variable to be flipped in each search step. First, a constraint c' is uniformly selected at random from the set of all currently unsatisfied constraints. Then, a variable involved in c' is selected according to the following criteria: If flipping any of the variable involved in c' leads to a decrease in the evaluation function, the variable that leads to the largest such decrease is selected; if there are several such variables, the one that was flipped least recently is chosen. Otherwise, with a small probability *wp*, the variable that has been flipped least recently is selected; and in the remaining cases, the variable whose flip would cause a minimal increase in the evaluation function is chosen; again, ties are broken in favour of the least recently flipped variable. (At the beginning of the search, ties might arise between variables that haven't been flipped; such ties are broken uniformly at random.) Additionally, WSAT($\mathcal{PB}$) uses a simple tabu mechanism that excludes all variables that have been flipped within the previous *tl* search steps from the selection process.

Different from conventional WalkSAT, WSAT($\mathcal{PB}$) supports a biased random initialisation of the search process in which each variable is independently set to zero with probability p_z and to one otherwise; however, experimental results suggest that using a biased initialisation (*i.e.*, $p_z \neq 0.5$) generally does not lead to performance improvements [Walser, 1997].

Furthermore, the WSAT($\mathcal{PB}$) algorithm as presented in [Walser, 1997] can also be used to solve an optimisation variant of Pseudo-Boolean CSP in which a subset of the constraints are considered as "soft constraints" and the objective is to find variable assignments that satisfy all conventional (hard) constraints while minimising the number of unsatisfied soft constraints. This problem can be seen as a special case of MAX-CSP, the optimisation variant of CSP, which will be discussed in more detail in Chapter 7, where we also describe the mechanism used by WSAT($\mathcal{PB}$) to handle soft constraints.

A large number of practically relevant problems can be formulated easily and naturally within the Pseudo-Boolean CSP framework. WSAT($\mathcal{PB}$) was tested on radar surveillance problems (which include soft constraints) and the Progressive Party Problem [Smith *et al.*, 1996]. For both problems, WSAT($\mathcal{PB}$) showed significantly improved performance over a state-of-the-art commercial integer programming package (CPLEX) and results from the literature. More recently, WSAT($\mathcal{PB}$) was extended to handle non-Boolean integer programming problems [Walser *et al.*, 1998]; the resulting WSAT(OIP) algorithm has been shown to achieve excellent performance on various combinatorial problems, including capacitated production planning and AI planning problems [Walser *et al.*, 1998; Kautz and Walser, 1999].

WalkSAT Algorithms for Many-Valued SAT

Another interesting subclass of CSP is the classof non-Boolean or many-valued satisfiability problems [Frisch and Peugniez, 2001b; Béjar and Manyà, 1999]. In non-Boolean SAT (NB-SAT), each variable may take values from some finite domain D, which may contain more than twovalues [Frisch and Peugniez, 2001b]. Formally, a non-Boolean literal is of the form z/d,or $\neg z/d$, where z is a variable and d a value from the domain of z. The value of z/d under the (non-Boolean) variable assignment a is *true* if and only if$z$ is set to d in a, and *false* otherwise; the value of $\neg z/d$ under a is obtained by negating the value of z/d under a.Analogously to conventional SAT, non-Boolean SAT is the problem to decide for a given non-Boolean CNF formula, *i.e.*, for a conjunction over disjunctions of non-Boolean literals, whether or not it has a satisfying (non-Boolean) assignment. Obviously, any conventional CNF formula can be represented by a non-Boolean CNF formula with the same number of clauses and variables.When encoding NB-SAT instances into SAT, however, a significantly higher number of variables and CNF clauses as used in the non-Boolean formula may be required.

Because NB-SAT instances have the same clause structure as conventional SAT instances, WalkSAT can be generalised to non-Boolean SAT in a rather straightforward way; the only major difference lies in the fact that in NB-SAT, the concept of a variable flip needs to be redefined. In NB-WalkSAT, the non-Boolean variant of WalkSAT by Frisch and Peugniez, a variable flip corresponds to assigning a different value to a non-Boolean variable such that the literal selected in the corresponding search step, and hence the clause in which appears, becomes satisfied [Frisch and Peugniez, 2001a]. (It may be noted that this constitutes an important difference to WSAT($\mathcal{PB}$), in which search steps do not necessarily satisfy any previously unsatisfied constraints.) Otherwise, NB-WalkSAT is identical to WalkSAT/SKC, although other WalkSAT variants can easily be extended to NB-SAT in an analogous way.

Béjar and Manyà have introduced a similar extension of WalkSAT, called MV-WalkSAT, which solves a variant of many-valued SAT that is slightly richer than the non-Boolean CNF formulae underlying NB-SAT [Béjar and Manyà, 1999; **?**]. Both, NB-WalkSAT and MVwsat were applied to many-valued SAT encodings of various combinatorial decision problems, such as graph colouring, where they showed excellent performance. However, as of todate, the question whether these and other SLS algorithms for many-valued SAT can substantially outperform state-of-the-art SLS algorithms for SAT applied to suitably encoded instances has not been answered conclusively.

The Min-Conflicts Heuristic and Variants

There are a number of SLS algorithms for general CSP, although in many cases, the implementations are restricted to binary constraints. Among the most widely known of these are the *Min Conflicts Heuristic* (MCH) [Minton *et al.*, 1990; 1992] and its variants. MCH iteratively modifies the assignment of a single variable in order to minimise the number of violated constraints, which is achieved in the following way: Given a CSP instance P, the search process is initialised by assigning each variable in P a value that is chosen uniformly at random from its domain. Then, in each local search step, first a CSP variable x_i is selected uniformly at random from the *conflict set* $K(A)$, which is the set of all variables that appear in a constraint violated under the current assignment A. A new value d is then chosen from the domain of x_i, D_i, such that by assigning d to x_i the number of constraint violation (conflicts) is minimised. If there is more than one value of d with that property, one of the minimising values is chosen uniformly at random. In many ways, MCH is analogous to the SLS algorithms for SAT described earlier in this chapter. Like all SAT algorithms covered here, MCH is based on a one-exchange neighbourhood. Considering that CNF clauses in SAT play the same role as constraint relations in CSP, the evaluation function underlying MCH, defined as the number of constraints violated under a given assignment, is the same as used by GSAT or Novelty. The way in which MCH selects the variable whose assignment is to be modified in each local search step is similar to the two-stage variable selection process underlying the WalkSAT architecture.

Like most iterative improvement methods, MCH is essentially incomplete, since it can get stuck in local minima of the evaluation function. The simplest way to overcome this problem is to use a static restart mechanism analogous to the one found in GSAT. Not surprisingly, however, there are other, substantially more effective solutions. These are mainly derived from mechanisms used in the better performing GSAT and WalkSAT algorithms, which is somewhat surprising, considering that MCH itself predates all of the SLS algorithms for SAT discussed above, including GSAT and basic WalkSAT.

WMCH is a variant of MCH that uses a random walk mechanism analogous to GWSAT [Wallace and Freuder, 1995]. In each WMCH step, first a variable x_i is chosen uniformly at random from the conflict set (as in MCH). Then, with probability $wp > 0$, x_i is assigned a value from its domain D_i that has been chosen according to a uniform distribution; this kind of search step is called a random walk step. In the remaining cases, *i.e.*, with probability $1 - wp$, a conflict-minimising value is chosen and assigned, as in a conventional MCH step. As in the case of GWSAT, this random walk mechanism renders WMCH probabilistically approximately complete for $wp > 0$. Furthermore, WMCH has been empirically observed to perform substantially better than MCH with random restart [Stützle, 1998].

Note that different from the random walk steps used in SLS algorithms for SAT, such as GWSAT, random walk steps in WMCH do not necessarily have the effect of satisfying a previously unsatisfied constraint. WMCH can be varied slightly such that in each random walk step, after choosing a variable x_i involved in a currently violated constraint C', x_i is assigned a value such that C' becomes satisfied; if no such value exists, a value is chosen at random. This variant, however, performs only marginally better than the random walk mechanism used in WMCH [Stützle, 1998].

Analogous to GSAT and WalkSAT, MCH can be extended with a tabu search mechanism. In TMCH, after each search step, *i.e.*, after the value of variable x_i is changed from d to d', the variable/value pair (x_i, d) is tabu for the next tl steps. While (x_i, d) is tabu, value d is excluded from the selection of values for x_i, except if assigning d to x_i leads to an improvement over the best assignment encountered so far (aspiration criterion). TMCH appears to generally perform better than WMCH. Interestingly, a tabu tenure setting of $tl = 2$ was found to consistently result in good performance for CSP instances of different types and size [Stützle, 1998].

Galinier and Hao's Tabu Search Algorithm

The tabu search algorithm by Galinier and Hao [Galinier and Hao, 1997], TSGH, is currently one of the best performing SLS algorithms for CSP. TSGH is based on the same neighbourhood and evaluation function as MCH, but uses a different mechanism for selecting the variable/value pair involved in each search step: among all pairs (x, d') such that variable x appears in a currently violated constraint and d' is any value from the domain of x, TSGH chooses the one that leads to a maximal decrease in the number of violated constraints. If multiple such pairs exist, one of them is selected uniformly at random. As in MCH, the actual search step is then performed by assigning d' to x. This best-improvement strategy is augmented with the same tabu mechanism used in TMCH: After changing the assignment of x from d to d', the variable value pair (x, d) is tabu for tl search steps. As in TMCH, an aspiration criterion is used to override the tabu status of variable/value pairs corresponding to search steps that lead to improvements over the best assignment encountered since the search was initialised.

In order to achieve competitive performance of TSGH, it is crucial to avoid computing evaluation function values for every variable/value pair that might potentially be involved in a search step. Instead, to implement TSGH efficiently, a caching and updating technique analogous to the one used for efficient implementations of GSAT (see in-dept section on page 218), is used [Fleurent and Ferland, 1996; Galinier and Hao, 1997]: After initialising the search, the effects on the evaluation function of changing the assignment of any variable x to any value d from its domain, are computed and stored in a two-dimensional table of size $n \times k$, where n is the number of constrain variables, and k is the size of the largest domain in the given CSP instance P. Based on the entries in this table, the (non-tabu) variable/value pair that results in the maximal improvement in the evaluation function value can be selected in time $O(n \cdot k)$ in the worst case. After each search step, only the entries in the table that are affected by the corresponding change in the current assignment need to be updated. For CSP with binary constraint relations, initialising the table takes time $O(n^2 \cdot k)$ in the worst case; the update after a search step can be performed in $O(n \cdot k)$ in the worst case, but is substantially faster for CSP instances with sparse constraint graphs. Using this technique and an efficient implementation of the tabu mechanism, as described for GSAT/TABU, the search steps of TSGH are as efficient as those of MCH.

It may be noted that TSGH was originally introduced as an algorithm for MAX-CSP, the optimisation variant of CSP in which the objective is to find a variable assignment that satisfies a maximal number of constraints. SLS algorithms for MAX-CSP will be further discussed in Chapter 7. Empirical studies suggest that when applied to conventional CSP, TSGH generally achieves better performance than any of the MCH variants, including TMCH, rendering it one of the best known SLS algorithms for CSP. Unlike in the case of TMCH, for TSGH, the optimal setting of the tabu tenure parameter, *tl*, increases with instance size, which makes it harder to solve previously unknown CSP instances with peak efficiency.

6.7 Further Readings and Related Work

SAT and CSP have been extensively studied for several decades, and there is an extensive body of literature on these problems and on algorithms for solving them. The SLS algorithms presented in this chapter have been selected primarily based on their performance and historical significance; however, there are many other SLS algorithms for SAT and CSP that are interesting and fairly widely known.

One of the earliest applications of SLS techniques to SAT is found in Jun Gu's SAT1 algorithm family. Developed independently and published around the same time as the basic GSAT algorithm, the first SAT1 algorithms are based on a simple iterative improvement method augmented with various techniques for overcoming search stagnation [Gu, 1992]. Subsequently, these early SAT1 algorithms have given rise to numerous variants and extensions, including parallel SLS algorithms for SAT, complete SAT algorithms obtained from combining SLS techniques and backtracking algorithms, and special cases of iterated local search. Many of these algorithms have been applied successfully to SAT-encodings of real-world VLSI circuit testing and synthesis and scheduling problems. A good overview of this line of work can be found in [Gu *et al.*, 1997]. Both, basic GSAT and the earliest SAT1 algorithms are predated by the Steepest Ascent Mildest Descent (SAMD) algorithm for MAX-SAT [Hansen and Jaumard, 1990], which will be covered in some more detail in Chapter 7.

Since the early 1990s, a large number SLS algorithms for SAT have been introduced and studied in the literature. These include methods based on Simulated Annealing [Spears, 1993; Beringer *et al.*, 1994; Selman *et al.*, 1994], Evolutionary Algorithms [Gottlieb *et al.*, 2002], and Greedy Randomized Adaptive Search Procedures (GRASP) [Resende and Feo, 1996]. While some of these algorithms have been directly compared to some of the SAT algorithms presented in this chapter, there is no evidence that any of them might generally perform better than the best WalkSAT or dynamic local search algorithms.

SLS algorithms also play an important role in the theoretical complexity analysis of SAT. Using a variant of Christos Papadimitriou's Random Walk algorithm [Papadimitriou, 1991] that restarts the search from a randomly chosen assignment after $3n$ variable flips, Uwe Schöning proved that any k-CNF formula with n variables can be solved in time $poly(n) \cdot 2(k-1)/k)^n$, where $poly(n)$ is an arbitrary polynomial function over n [Schöning, 1999; Schöning, 2002]. By using the same algorithm with a modified search initialisation, which exploits sets of mututally independent clauses, the currently best known lower bound on the time complexity of SAT for 3-CNF formulae of $poly(n) \cdot 1.3303^n$ was obtained [Schuler *et al.*, 2001]. For k-CNF with $k > 3$, the currently best lower bounds on the time complexity of SAT were obtained by Paturi *et al.* based on an algorithm that first calculates the closure of the given formula F under bounded-length resolution, and then performs a simple stochastic iterated construction search in order to find models of the resulting CNF formula [Paturi *et al.*, 1997; 1998]. This algorithm forms the basis of another recent SAT solver, UnitWalk [Hirsch and Kojevnikov, 2001], which was empirically shown to reach the performance of state-of-the-art SLS algorithms for SAT for various classes of benchmark instances and has been shown to be probabilistically approximately complete.

The survey paper by Gu *et al.* [Gu *et al.*, 1997] provides an excellent overview of the SAT problem, including an interesting classification of SAT algorithms, complexity results, various types of benchmark instances, and a large number of practical applications. It also presents a number of SLS algorithms for SAT which, however, is somewhat incomplete and now rather outdated, as well as a comprehensive list of references. A more recent study by Hoos and Stützle [Hoos and Stützle, 2000b; 2000a] presents a fairly complete and up-to-date overview of GSAT and WalkSAT algorithms, including detailed results on the run-time behaviour and performance of these algorithms.

The GSAT architecture can be generalised to CSP in a rather straightforward way; a GSAT variant that includes various additional SLS mechanisms, such as random walk, clause weighting, and a dynamic restart strategy, was described by Kask and Dechter, who used it in an empirical study regarding the effectivity of preprocessing techniques for SAT and CSP [Kask and Dechter, 1995]. An interesting extension that combines GSAT with a tree search mechanism based on cycle-cutsets, called GSAT+CC, has been applied to Random Uniform Binary CSP [Kask and Dechter, 1996]. Empirical results suggest that for a limited class of CSP instances (those with small cutsets), using the additional tree search mechanism results in substantially improved performance, while on other subclasses of CSP, GSAT+CC performs rather poorly.

[hh: I'd really like to include a statement about the performance of Kask & Dechter's GSAT and GSAT+CC as compared to MCH, WMCH, TSGH, but I am not aware of any published results that would provide a solid basis for such a statement]

GENET [Davenport *et al.*, 1994], one of the first extensions of MCH, is another interesting and relatively widely known SLS algorithm for CSP. Originally inspired by the Breakout Method [Morris, 1993], GENET is based on a dynamic local search algorithm and is a precursor of the Guided Local Search algorithm by Voudouris and Tsang [Voudouris, 1997; Voudouris and Tsang, 1999] (see also Section 2.2). A recent study has produced empirical evidence suggesting that a version of the Breakout Method based on the same type of neighborhood relation as Galinier and Hao's Tabu Search algorithm performs significantly better than random walk extensions of MCH [J. P. Williams and Dozier, 2001]. This indicates that dynamic local search is a promising approach for future CSP algorithms.

Binary CSP instances are commonly used for the evaluation of Evolutionary Algorithms, where they serve as a benchmark for investigating algorithm behaviour for constrained problems [Eiben, 2001; Marchiori and Steenbeek, 2000; Craenen *et al.*, 2000; Dozier *et al.*, 1998]. ¿From the published results, however, it is unclear how these algorithms compare to state-of-the-art SLS algorithms for CSP in terms of performance; given the experience for SAT, it is doubtful that the proposed EAs can reach state-of-the-art performance.

Recently, Christine Solnon developped an Ant Colony Optimisation algorithm for CSP, using a local search procedure based on MCH [Solnon, 2002b; 2002a]. This algorithm was successfully applied to Uniform Random Binary CSP and graph coloring instances; for hard Uniform Random Binary CSP instances from the soubility phase transition region, the ACO algorithm performed better than WMCH. Furthermore, ACO algorithms were successfully applied to subclasses of CSP, such as the car sequencing problem [Solnon, 2000].

As general references for CSP, the interested reader is referred to the book by Tsang [Tsang, 1993] (in parts now somewhat outdated) as well as the new book by Rina Dechter [**?**].

6.8 Summary

In this chapter, we presented and discussed SLS algorithms for two important and prominent combinatorial decision problems, the Propositional Satisfiability Problem (SAT) and the Constraint Satisfaction Problem (CSP). Both problems are of substantial theoretical interest and also have a range of real-word applications.

SAT is one of the most prominent and widely studied $\mathcal{NP}$-complete decision problems. Most SAT algorithm operate on propositional formulae in conjunctive normal form (CNF); since any formula can be transformed into CNF, this is not a serious limitation. Moreover, instances of other combinatorial problems can often be encoded into SAT using reasonably compact and natural CNF representations. While SAT can be formulated as a special case of CSP as well as of 0-1 Integer Linear Programming, the conventional logical formulation appears to provide a much better basis for solving SAT instances efficiently. Polynomial time simplification techniques, such as unit propagation, play a crucial role for preprocessing SAT instances before applying a general SAT solver, as well as within systematic search algorithms for SAT; on their own, they can be used for solving several interesting subclasses of SAT efficiently.

We discussed various types of SAT instances, including Random-k-SAT, one of the most prominent classes of randomly generated SAT instances, and the solubility phase transition phenomenon observed for this subclass of SAT; SAT-encodings of other combinatorial problems; and instances from several practical applications of SAT, such as circuit verification and design. We briefly mentioned a number of generalisations of SAT, including Multi-Valued SAT, MAX-SAT, and the Satisfiability Problem for Quantified Boolean Formulae (QBF-SAT), as well as problems related to SAT, such as the Propositional Validity Problem (VAL).

We presented three classes of SLS algorithms for SAT: the GSAT architecture, the WalkSAT architecture, and dynamic local search algorithms for SAT. While GSAT and related algorithms played a pivotal role in the early development of SLS algorithms for SAT, recent WalkSAT and dynamic local search algorithms, such as Novelty$^+$ and SAPS, are amongst the best SAT solvers currently known.

The Constraint Satisfaction Problem (CSP) can be seen as a generalisation of SAT in which the variable domains can be different from the set $\{\top, \bot\}$ and the constraining conditions that have to be simultaneously satisfied by any solution can be arbitrary relations between a subset of CSP variables. Our discussion was focussed on finite discrete CSP, an $\mathcal{NP}$-complete subproblem in which all variable domains are finite and discrete sets. We gave a brief overview of various widely used classes of benchmark instances for CSP, including Uniform Random Binary CSP, as well as Graph Colouring and Quasigroup Completion instances.

We discussed three SLS approaches for solving CSP: (1) Encoding CSP instances into SAT and solving them using SLS algorithms for SAT (or any other type of SAT solver); (2) using direct generalisations of SAT algorithms for solving CSP instances; (3) and applying native SLS algorithms for CSP. It is presently not clear whether any of these approaches achieves substantially better performance than the others.

For the first approach, different SAT encodings of CSP can be used.

Amongst the two encodings discussed here, the sparse encoding and the compact encoding, the former produces SAT instances that appear to be consistently easier to solve for standard SLS algorithms for SAT. In the context of the second approach, we discussed direct generalisations of WalkSAT for two interesting subclasses of CSP, Pseudo-Boolean CSP (also known as Pseudo-Boolean Programming) and Many-Valued SAT. Our discussion of the third approach was focussed on the Min-Conflicts Heuristic (MCH) and the tabu search algorithm by Galinier and Hao; while the former played a pivotal role in the development of SLS algorithms for SAT and CSP, the latter achieves substantially better performance than MCH and its more recent variants.

Overall, SAT is (and continues to be) an ideal problem for developing and evaluating algorithmic ideas, including SLS techniques, because of its conceptual simplicity as well as its theoretical and practical significance. While many problems are more naturally encoded into CSP than into SAT, it is presently not clear that native CSP algorithms can substantially outperform highly optimised SAT algorithms on suitable chosen encodings. It may, however, be expected that when dealing with more specialised constraints, generalisations of successful SLS algorithms for SAT that are augmented with specific methods for handling certain types of complex constraints might be the best approach.

Furthermore, since the development and understanding of SLS algorithms appears to be significantly further advanced for SAT than for CSP, there appears to be substantial room for further improvements in native SLS algorithms for CSP.

Finally, it may be noted that for both, SAT and CSP, the potential of many advanced SLS techniques, such as ILS, variable depth search, or ACO, has not been fully explored and it is quite likely that by using such advanced techniques, further significant improvements in our ability to solve these problems can be achieved.

6.9 Exercises

Exercise 6.1 (Easy) Consider the problem G of colouring the vertices of the graph shown in Figure **??** with 4 colours such that no two vertices connected by an edge have the same colour. **[hh: add figure – TODO(hh)]**

(a) Formulate this problem as a SAT instance, *i.e.*, give a CNF formula F such that any model of F corresponds to a solution of G.

(b) Formulate this problem as a CSP instance.

Exercise 6.2 (Medium) Design and describe an Ant Colony Optimisation algorithm for SAT.

Exercise 6.3 (Easy) Describe how you can use a WalkSAT algorithm to solve the Propositional Validity Problem (VAL) for a given formula in disjunctive normal form (DNF).

Exercise 6.4 (Medium) When allowing an arbitrary number of tries, GSAT is probabilistically approximately complete. Explain why nevertheless other mechanisms for achieving the PAC property, such as Random Walk, are preferable over the simple static restart mechanism.

Exercise 6.5 (Easy) How many clauses and variables are required in the worst case for encoding an NB-SAT instances with n variables and m clauses into a semantically equivalent SAT instances using a sparse encoding?

Exercise 6.6 (Medium) Develop the details of caching and updating scheme for the evaluation function values for the TSGH algorithm.

[hh: might want to replace one of the exercises with a harder exercise; should have an experimental and an implementation exercise]