Handbook of Combinatorial Optimization (Vol. 1)
D.-Z. Du and P.M. Pardalos (Eds.) pp. 77-148

Approximate Algorithms and Heuristics for *MAX–SAT*

Roberto Battiti
Dipartimento di Matematica, Università di Trento
38050 Povo (Trento), Italy
E-mail: battiti@science.unitn.it

Marco Protasi
Dipartimento di Matematica, Università di Roma "Tor Vergata"
Via della Ricerca Scientifica, 00133 Roma, Italy
E-mail: protasi@mat.utovrm.it

Contents

1 Introduction

In the Maximum Satisfiability (*MAX–SAT*) problem one is given a Boolean formula in conjunctive normal form, i.e., as a conjunction of clauses, each clause being a disjunction. The task is to find an assignment of truth values to the variables that satisfies the maximum number of clauses.

In our work, n is the number of variables and m the number of clauses, so that a formula has the following form:

$$\bigwedge_{1 \le i \le m} \left(\bigvee_{1 \le k \le |C_i|} l_{ik} \right)$$

where $|C_i|$ is the number of literals in clause C_i and l_{ik} is a literal, i.e., a propositional variable u_j or its negation $\overline{u_j}$, for $1 \le j \le n$. The set of clauses in the formula is denoted by $\mathbf{C}$. If one associates a *weight* w_i to each clause C_i one obtains the weighted *MAX–SAT* problem, denoted as

MAX W–SAT: one is to determine the assignment of truth values to the n variables that maximizes the sum of the weights of the satisfied clauses. Of course, *MAX–SAT* is contained in *MAX W–SAT* (all weights are equal to one). In the literature one often considers problems with different numbers k of literals per clause, defined as *MAX–k–SAT* , or *MAX W–k–SAT* in the weighted case. In some papers *MAX–k–SAT* instances contain *up to k* literals per clause, while in other papers they contain *exactly k* literals per clause. We consider the second option unless otherwise stated.

MAX–SAT is of considerable interest not only from the theoretical side but also from the practical one. On one hand, the decision version *SAT* was the first example of an $\mathcal{NP}$–complete problem [25], moreover *MAX–SAT* and related variants play an important role in the characterization of different approximation classes like $\mathcal{APX}$ and $\mathcal{PTAS}$ [8]. On the other hand, many issues in mathematical logic and artificial intelligence can be expressed in the form of satisfiability or some of its variants, like constraint satisfaction. Some exemplary problems are consistency in expert system knowledge bases [69], integrity constraints in databases [7, 35], approaches to inductive inference [49, 56], asynchronous circuit synthesis [46, 74].

The main purpose of this work is that of summarizing the basic approaches for the exact or approximated solution of the *MAX W–SAT* and *MAX–SAT* problem. The presentation of algorithms for the related *SAT* problem is therefore limited to a quick overview of some basic techniques and of methods that can be used also for *MAX–SAT* . Of course, given the impressive extension of the research in this area, we are not aiming at a comprehensive survey of the literature, and we have confined ourselves to citing the sources that we have used, some sources of historical significance, and some papers that are paradigmatic for the different approaches.

1.1 Notation and graphical representation

A clause will be represented either as $C = \overline{u} \vee v \vee z$ or as a set of literals, as in $C = \{\overline{u}vz\}$.

For the following discussion, it can be useful to help the intuition with a graphical representation of a formula in conjunctive normal form, as depicted in Fig. 1. In the figure, one has a case of *MAX 3–SAT* : all clauses have three literals and the formula is:

$$(u_1 \vee \overline{u_3} \vee u_5) \wedge (\overline{u_2} \vee \overline{u_4} \vee \overline{u_5}) \wedge (\overline{u_1} \vee u_3 \vee \overline{u_4})$$

Truth values to variables are assigned by placing a black triangle to the left if the variable is **true**, to the right if it is **false**. Each literal is depicted with

a small circle, placed to the left if the corresponding variable is **true**, to the right in the other case. If a literal is *matched* by the current assignment (e.g., if the literal asks for a **true** value and the variable is set to **true**, or if is asks for **false** and the variable is **false**), it is shown with a gray shade. The *coverage* of a clause is the number of literals in the clause that are matched by the current assignment, and it is illustrated by placing a black square in the appropriate position of an array with indices ranging from 0 to the number of literals in each clause $|C|$.

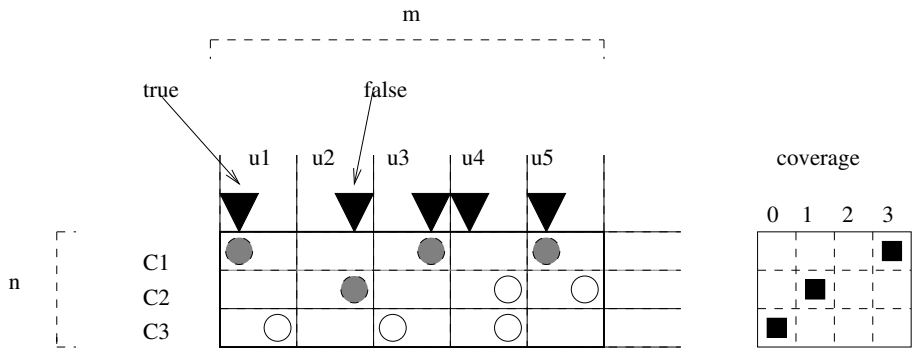

Figure 1: A formula in conjunctive normal form (CNF).

2 Resolution and Linear Programming

2.1 Resolution and backtracking for *SAT*

The basic method to solve *SAT* formulae is given by the recursive replacement of a formula by one or more formulae, the solution of which implies the solution of the original formula.

In *resolution* a variable is selected and a new clause, called the *resolvent* is added to the original formula. The process is repeated to exhaustion or until an empty clause is generated. The original formula is not satisfiable if and only if an empty clause is generated [77].

Let us now consider some details: A clause R is the *resolvent* of clauses C_1 and C_2 iff there is a literal $l \in C_1$ with $\bar{l} \in C_2$ such that $R = (C_1 \setminus \{l\}) \cup (C_2 \setminus \{\bar{l}\})$ and $u(l)$, the variable associated to the literal, is the only variable appearing both positively and negatively.

For the two clauses $C_1 = (l \vee a_1 \vee ... \vee a_A)$ and $C_2 = (\bar{l} \vee b_1 \vee ... \vee b_B)$ the resolvent is therefore the clause $R = (a_1 \vee ... \vee a_A \vee b_1 \vee ... \vee b_B)$. The resolvent

is a logical consequence of the logical *and* of the two clauses. Therefore, if the resolvent is added to the original set of clauses, the set of solutions does not change. It is immediate to check that, if both C_1 and C_2 are satisfied, i.e., have at least one matched literal, the resolvent must also be satisfied. In fact, if it is not, in the original clauses there are no matched literals apart from either $\overline{l}$ or l, but this implies that both clauses cannot be satisfied (see also Fig. 2 for a graphical illustration).

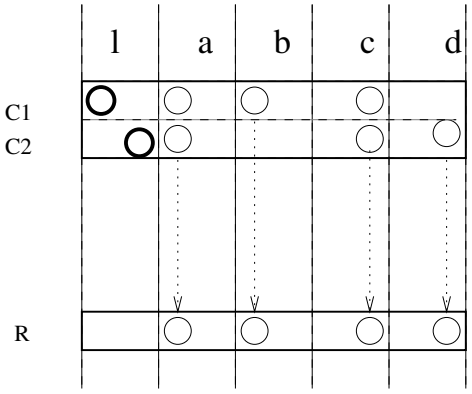

Figure 2: How to construct a resolvent, an example with variables l, a, b, c, d.

Davis and Putnam [29] started in 1960 the investigation of useful strategies for handling resolution. In addition to applying transformations that preserve the set of solutions they eliminate one variable at a time in a chosen order by using all possible resolvents on that variable. During resolution the lengths and the number of added clauses can easily increase and become extremely large.

Davis, Logemann and Loveland [28] avoid the memory explosion of the original DP algorithm by replacing the resolution rule with the *splitting rule* (Davis, Putnam, Logemann and Loveland, or DPLL algorithm for short). In splitting, a variable u in a formula is selected. Now, if there exist a satisfying truth assignment for the original formula then either u is **true** or $\overline{u}$ is **true** in the assignment. In the first case the formula obtained by eliminating all clauses containing u and by deleting all occurrences of $\overline{u}$ must be satisfied, see Fig 4. This derived formula is called $\mathbf{C}(u)$ in Fig. 3. In the second case, the formula obtained by eliminating all clauses containing $\overline{u}$ and all occurrences of u must be satisfied. *Vice versa*, if both derived formulae cannot be satisfied, neither can the original problem.

A *tree* is therefore generated. At the root one has the original problem

DPLL(**C** : set of clauses)
Input: Boolean CNF formula $\mathbf{C} = \{C_1, C_2, \ldots, C_m\}$
Output: Yes or No (decision about satisfiability)
1 **if C** is empty **then return** Yes
2 **if C** contains an empty clause **then return** No
3 **if** there is a pure literal l in **C** **then return** DPLL($\mathbf{C}(l)$)
4 **if** there is a unit clause $\{l\} \in \mathbf{C}$ **then return** DPLL($\mathbf{C}(l)$)
5 Select a variable u in **C**
6 **if** DPLL($\mathbf{C}(u)$) = Yes **then return** Yes
7 **else return** DPLL($\mathbf{C}(\overline{u})$)

Figure 3: The DPLL algorithm by Davis, Logemann and Loveland in recursive form. The recursive calls are executed on the problems derived after setting the truth value of the selected variable.

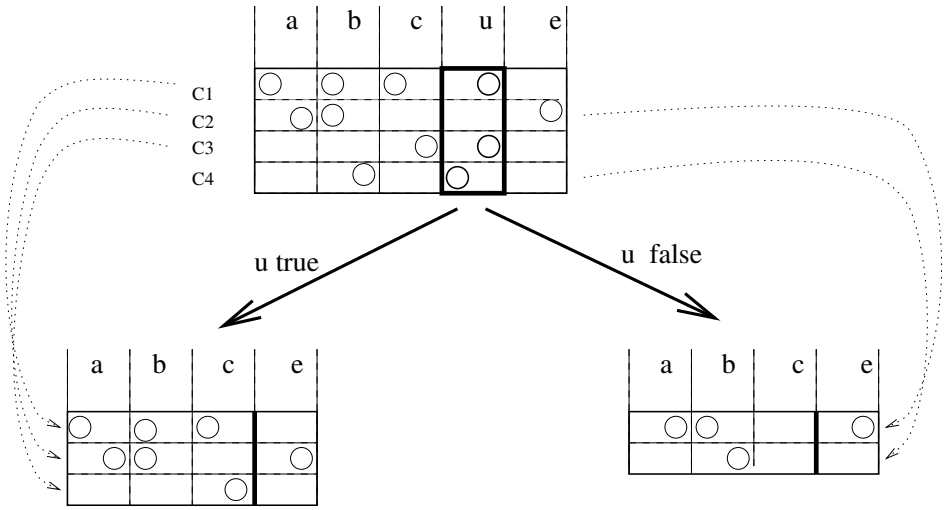

Figure 4: Example of splitting on a variable u.

and no variables are assigned values. At each node of the tree one generates two children by *selecting* one of the yet unassigned variables in the problem corresponding to the node and by generating the two problems derived by setting the variable to **true** or **false**. A trivial upper bound on the number of nodes in the tree is proportional to the number of possible assignments, i.e., $O(2^n)$. In fact, sophisticated techniques are available to reduce the number of nodes, that nonetheless remains exponential in the worst case.

The techniques include:

- avoiding the examination of a subtree when the fate of the current problem is decided (problems with an empty clause have no solutions, problems with no clauses have a solution). If the current problem cannot be solved, or if it is solved but one wants all possible solutions, one *backtracks* to the first unexplored branch of the tree. Note that, when splitting is combined with a depth-first search of the tree (as in the DPLL algorithm) one avoids the memory explosion because only one subproblem is active at a given time.
- *selecting* the next variable for the splitting based on appropriate criteria. For example, one can prefer variables that appear in clauses of length one (*unit clause rule*), or select a *pure literal* (such that it occurs only positive, or only negative), or select a literal occurring in the *smallest clause*.

A recent review of advanced techniques for resolution and splitting is presented in [45], and a summary of algorithms for deciding propositional tautologies is presented in [64].

It is worth noting that approaches based on the Davis-Putnam scheme tend to achieve the fastest speed when solving *SAT* problems. A recent state-of-the-art parallel implementation in given in [16]. Their previous sequential implementation turned out to be the fastest program in a *SAT* competition [18].

2.2 Integer programming approaches

The *MAX W–SAT* problem has a natural integer linear programming formulation (*ILP*). Let $y_j = 1$ if Boolean variable u_j is **true**, $y_j = 0$ if it is **false**, and let the Boolean variable $z_i = 1$ if clause C_i is satisfied, $z_i = 0$ otherwise. The integer linear program is:

$$\max \sum_{i=1}^{m} w_i \, z_i$$

subject to the following constraints:

$$\begin{aligned} \sum_{j \in U_i^+} y_j + \sum_{j \in U_i^-} (1 - y_j) \quad &\geq \quad z_i \ , \ i = 1, \cdots, m \\ y_j &\in \{0, 1\} \ , \ j = 1, \cdots, n \\ z_i &\in \{0, 1\} \ , \ i = 1, \cdots, m \end{aligned}$$

where U_i^+ and U_i^- denote the set of indices of variables that appear unnegated and negated in clause C_i, respectively.

Because the sum of the z_i w_i is maximized and because each z_i appears as the right-hand side of one constraint only, z_i will be equal to one if and only if clause C_i is satisfied.

If one neglects the objective function and sets all z_i variables to 1, one obtains an integer programming feasibility problem associated to the SAT problem [15].

The integer linear programming formulation of MAX–SAT suggests that this problem could be solved by a *branch-and-bound* method. A tree is generated, see also the DPLL method, where the root corresponds to the initial instance and two children are obtained by *branching*, i.e., by selecting one free variable and setting it **true** (left child) and **false** (right child). An *upper bound* on the number of satisfied clauses can be obtained by using a linear programming relaxation: the constraints $y_j \in \{0,1\}$ and $z_i \in \{0,1\}$ are replaced by $y_j \in [0,1]$ and $z_i \in [0,1]$. One obtains a Linear Programming (LP) problem that can be solved in polynomial time and, because the set of admissible solutions is enlarged with respect to the original problem, one obtains an upper bound.

Unfortunately this is not likely to work well in practice [48] because the solution $y_j = 1/2, j = 1, \cdots, n,\ z_i = 1, i = 1, \cdots, m$ is feasible for the LP relaxation unless there exist some constraint containing only one variable. The bounds so obtained would be very poor.

Better bounds can be obtained by using Chvátal cuts. In [49] it is shown that the resolvents in the propositional calculus correspond to certain cutting planes in the integer programming model of inference problems.

A general cutting plane algorithm for ILP , see for example [73], works as follows. One solves the LP *relaxation* of the problem: if the solution is integer the algorithm terminates, otherwise one adds linear constraints to the ILP that do not exclude integer feasible points. The constraints are added one at a time, until the solution to the LP relaxation is integer.

The application considered in [49] is to determine whether a formula in the propositional calculus implies another one. The propositional calculus is a formal logic involving propositions and logic connectives such as "not", "and", "or" and "implies". Any formula is equivalent to a conjunction of clauses (in particular, a rule in a knowledge base, such as "if A and B then C" can be written as a clause "not-A or not-B or C"). Thus, a set of clauses can be represented by a particular linear system $Ax \geq a$, also called *generalized set covering* problem. Finally, one can determine whether $Ax \geq a$ implies a formula F by expressing not-F as a system $Bx \geq b$ of

clauses, see for example [65], and checking whether the combined system $Ax \geq a$, $Bx \geq b$ has a binary solution. If it does *not*, $Ax \geq a$ implies F. In the applications $Ax \geq a$ can represent a knowledge base known to be consistent.

Cutting planes are generated in [49] by finding *separating resolvents*, i.e., a resolvent (expressed as a linear inequality) that is violated by the current solution of the LP relaxation. When no separating resolvents are found, the current system is solved with branch-and-bound. The experimental results are that the cutting plane algorithm is orders of magnitude faster on problems in which the premises do not imply the given proposition (the majority of random problems), and moderately faster on other random problems [49].

LP relaxations of integer linear programming formulations of MAX–SAT have been used to obtained upper bounds in [47, 84, 39]. A linear programming and rounding approach for $MAX\ 2$–SAT is presented in [22]. Their cutting plane algorithm starts from the LP relaxation of $MAX\ 2$–SAT , and has separation routines for two families of cuts: cycle and wheel inequalities. Upper and lower bounds are found, the latter by using a *rounding* procedure to convert a fractional LP solution to a $\{0, 1\}$ solution. A method for strengthening the Generalized Set Covering formulation is presented in [70], where Lagrangian multipliers guide the generation of cutting planes.

3 Continuous approaches

3.1 An interior point algorithm

The ILP feasibility problem obtained from SAT as described in the previous section is solved with an *interior point algorithm* in [56, 57]. In the *interior point algorithm* one applies a function minimization method based on *continuous* mathematics to the inherently discrete SAT problem.

In [57] the application is to a problem of *inductive inference*, in which one aims at identifying a hidden Boolean function using outputs obtained by applying a limited number of random inputs to the hidden function. The task is formulated as a SAT problem, which is in turn formulated as an integer linear program:

$$A^T y \leq c\ ,\ y \in \{-1, 1\}^n \tag{1}$$

where A^T is an $m \times n$ real matrix and c a real m vector.

The interior point algorithm is based on finding a local minimum in the

box $-1 \leq y_j \leq 1$ of the *potential function*:

$$\phi(y) = \log \left\{ \frac{n - y^T y}{\prod_{k=1}^{m} (c_k - {a_k}^T y)^{1/m}} \right\} \tag{2}$$

by an iterative method. The denominator of the argument of the log is the geometric mean of the *slacks* (a_k is the k-th column of matrix A). It is shown that, if the integer linear program has a solution, y^* is a global minimum of this potential function if and only if y^* solves the integer program. The next iterate y^{k+1} (interior point solution, i.e., such that $A^T y < c$) is obtained by moving in a descent direction Δy from the current iterate y^k such that $\phi(y^{k+1}) = \phi(y^k + \alpha \Delta y) < \phi(y^k)$. Each iteration in [57] is based on the *trust region approach* of continuous optimization where the Riemannian metric used for defining the search region is dynamically modified. The feasibility of the approach for inductive inference is demonstrated in [57].

3.2 Continuous unconstrained optimization

In some techniques the *MAX–SAT* (or *SAT*) problem is transformed into an unconstrained optimization problem on the real space R^n and solved by using existing global optimization techniques.

Some examples of this approach include the UniSAT models [43] and the *neural network* approaches [54, 19]. In general, these techniques do not have performance guarantees because they assure only the local convergence to a locally optimal point, not necessarily the global optimum.

The local convergence properties of some optimization algorithms are considered in [44]. The main results are that, for any CNF formula, if y^* is a solution point of the objective function f defined on R^n associated to the problem, i.e., $f(y^*) = 0$, the *Hessian* matrix $H(y^*)$ is positive definite and therefore the convergence ratios of the steepest descent, Newton's method and coordinate descent methods can be derived, see [44] for the details. Let us note that, to obtain these results, one assumes that the initial solution is "sufficiently close" to the optimal solution.

4 Approximate algorithms

The present section presents the first important approximate algorithms for *MAX–SAT* . However, in order to evaluate the goodness of the algorithms, one needs to define the meaning of *approximation algorithm* with a "guaranteed" quality of approximation.

In the following it is assumed that the reader is familiar with the concepts of the complexity classes classes $\mathcal{P}$ and $\mathcal{NP}$ and with elementary concepts from probability theory.

4.1 Definitions and basic results

First of all, let us present a general definition of optimization problem.

Definition 4.1 *An optimization problem $P = (I, sol, m, opt)$ belongs to the class $\mathcal{NPO}$ if the following holds:*

1. *the set of instances I is recognizable in polynomial time,*
2. *given an instance $x \in I$, $sol(x)$ is the set of the feasible solutions of x; moreover there exists a polynomial q such that, given an instance $x \in I$, for any $y \in sol(x)$, $|y| < q(|x|)$ and, besides, for any y such that $|y| < q(|x|)$, it is decidable in polynomial time whether $y \in sol(x)$,*
3. *given an instance $x \in I$, a feasible solution y of x, $m(x, y)$ is the objective function and is computable in (deterministic) polynomial time.*
4. *$opt \in \{max, min\}$ specifies whether one has a maximization or minimization problem.*

Finally $m^*(x)$ will denote the optimal value of instance x. When it is clear from the context, one will use simply m^*.

A problem belonging to $\mathcal{NPO}$ will be called an $\mathcal{NPO}$ problem.

Note that the difficulty of solving an $\mathcal{NPO}$ problem is based on the fact that, in many cases, the set of feasible solutions is exponentially large.

Even if not explicitly stated, there is a nondeterministic polynomial time computation model underlying this definition. The nondeterministic machine of polynomial complexity may run in the following way: in non-deterministic polynomial time, all strings y such that $|y| < q(|x|)$ are generated. Afterwards any string is tested for membership in $sol(x)$ in polynomial time. If the test is positive, $m(x, y)$ is computed (again in polynomial time) and both y and $m(x, y)$ are returned.

The definition of the class $\mathcal{NPO}$ formalizes the notion of optimization problem with an associated decision version which is in $\mathcal{NP}$. In addition, let as define as $\mathcal{PO}$ the subclass of $\mathcal{NPO}$ formed by problems that can be solved in polynomial time. Many classical combinatorial problems belong to the class $\mathcal{NPO}$; for instance, the traveling salesperson problem, the knapsack problem, the minimal covering of a graph and so on.

MAX W–SAT is another important example of $\mathcal{NPO}$ problem. In this case one has

1. I = sets U of Boolean variables and a collection $\mathbf{C} = C_1, \ldots, C_m$ of clauses over U, a set $W = w_1, \ldots, w_m$ of integers (weights) associated to the clauses.

2. Given an instance x of I, $sol(x)$ = set of truth assignments U to the variables in the problem. Moreover $|U| < (|x|)$ and it is possible to decide in polynomial time whether a string is a truth assignment for the formula;

3. Given an instance x of I and a feasible solution y of x, $m(x, y)$ = sum of the weights associated to the satisfied clauses; m is trivially computable in polynomial time

4. opt= max.

A subset is given by the *MAX–SAT* problem, obtained when all weights are equal to one. Note that, in this definition, the set of Boolean variables and a truth assignment are denoted with the same symbol U; even if formally questionable, this identification will allow to simplify the presentation of many results. However, when this abuse of notation could raise problems to the reader, different symbols will be used.

Because an $\mathcal{NPO}$ problem that is not in $\mathcal{PO}$ cannot be solved in polynomial time unless $\mathcal{P} = \mathcal{NP}$, a natural approach consists of looking for "good" approximate solutions.

Definition 4.2 *Given an $\mathcal{NPO}$ problem $P = (I, sol, m, opt)$, an algorithm A is an* approximation algorithm *if, for any given instance $x \in I$, it returns an* approximate solution*, that is a feasible solution $A(x) \in sol(x)$.*

Because the present work is dedicated to *MAX–SAT* the following definitions will be restricted to the case of maximization problems.

An approximation algorithm can be usefully applied only if it achieves approximate solutions whose values are "near" to the optimum value. Therefore one is interested in determining how far from the optimal value is the value of the achieved solution.

Definition 4.3 *Given an $\mathcal{NPO}$ problem P, an instance x and a feasible solution y, the* performance ratio *of y is*

$$R(x, y) = \frac{m(x, y)}{m^*(x)}.$$

When the performance ratio is close to 1, the value of the approximate solution is close to the optimum one.

Definition 4.4 *Given an $\mathcal{NPO}$ problem P and an approximation algorithm A, A is said to be an* ε-approximate algorithm *if, given any input instance x, the performance ratio of the approximate solution $A(x)$ verifies the following relation:*

$$R(x, A(x)) \geq \varepsilon.$$

In other words, the solution provided by the algorithm must guarantee at least a value $\varepsilon m^*(x)$.

Definition 4.5 *An $\mathcal{NPO}$ problem P is* ε-approximable *if there exists a polynomial-time ε-approximate algorithm for P, with $0 < \varepsilon < 1$*

Definition 4.6 *$\mathcal{APX}$ is the class of all $\mathcal{NPO}$ problems that are ε-approximable.*

For a problem to join $\mathcal{APX}$ it is sufficient that the performance ratio is greater than or equal to ε for a particular value of ε. Of course, the goodness of the approximation algorithm strictly depends on how near ε is to 1.

In fact, in the class $\mathcal{APX}$ there are problems with different values of ε and therefore with different approximation properties. After taking this fact into account, the definition of ε-approximate algorithm can be strengthened in the following way:

Definition 4.7 *Let P be an $\mathcal{NPO}$ problem. An algorithm A is said to be a* polynomial-time approximation scheme ($\mathcal{PTAS}$) *for P if, for any instance x of P and any rational value $0 < \varepsilon < 1$, $A(x, \varepsilon)$ returns an ε-approximate solution of x in time polynomial in the size of the instance x.*

Definition 4.8 *$\mathcal{PTAS}$ is the class of $\mathcal{NPO}$ problems that allow a polynomial-time approximation scheme.*

The following theorem holds:

Theorem 4.1

- *MAX W–SAT belongs to the class $\mathcal{APX}$.*
- *MAX W–SAT does not belong to the class $\mathcal{PTAS}$ unless $\mathcal{P} = \mathcal{NP}$.*

The first part of the theorem will be demonstrated in the sequel, while citations will be given for the second part.

Finally, let us conclude this Subsection by introducing the important notion of *completeness in an approximation class*. As it is done in the $\mathcal{NP}$–completeness theory, one is interested in finding the "hardest" problem in the classes $\mathcal{APX}$ and $\mathcal{PTAS}$ that is, the most difficult ones from a computational point of view. One looks for problems which cannot have stronger approximation properties unless $\mathcal{P} = \mathcal{NP}$.

In complexity theory, the notion of hardest problem in a class is equivalent to saying that a problem is complete with respect to a suitable *reduction*. The same approach can be followed for approximation classes. Therefore, a definition of approximation–preserving reduction is presented and one will be able to define a complete problem. Actually, many different reductions have been proposed. In this paper we consider the reduction presented in [8] which has the relevant advantage that it can be used for defining the notion of completeness both in $\mathcal{APX}$ and in $\mathcal{PTAS}$.

Intuitively, in order to map an optimization problem P_1 into another optimization problem P_2, we need not only a function f mapping instances of P_1 into instances of P_2 but also a second function g mapping back feasible solutions of P_2 into feasible solutions of P_1, see also Fig. 5.

Definition 4.9 *Let P_1 and P_2 be two $\mathcal{NPO}$ problems. P_1 is said to be $\mathcal{PTAS}$–reducible to P_2 ($P_1 \leq_{\mathcal{PTAS}} P_2$) if three functions f, g, and c exist such that*

1. *For any $x \in I_{P_1}$, $f(x) \in I_{P_2}$ is computable in polynomial time.*
2. *For any $x \in I_{P_1}$, for any $y \in sol_{P_2}(f(x))$, and for any $\epsilon \in (0,1)^Q$ (set of positive rational numbers smaller than 1), $g(x, y, \epsilon) \in sol_{P_1}(x)$ is computable in time polynomial with respect to both $\mid x \mid$ and $\mid y \mid$.*
3. *$c : (0,1)^Q \to (0,1)^Q$ is computable and surjective.*
4. *For any $x \in I_{P_1}$, for any $y \in sol_{P_2}(f(x))$, and for any $\epsilon \in (0,1)^Q$, $1 - c(\epsilon) \leq R_{P_2}((f(x), y)$ implies $1 - \epsilon \leq R_{P_1}(x, g(x, y, \epsilon))$.*

The triple (f, g, c) is said to be a $\mathcal{PTAS}$–reduction from P_1 to P_2.

It is easy to demonstrate the following Lemma:

Lemma 4.2 *If $P_1 \leq_{\mathcal{PTAS}} P_2$ and $P_2 \in \mathcal{APX}$ (respectively, $P_2 \in \mathcal{PTAS}$), then $P_1 \in \mathcal{APX}$ (respectively, $P_1 \in \mathcal{PTAS}$).*

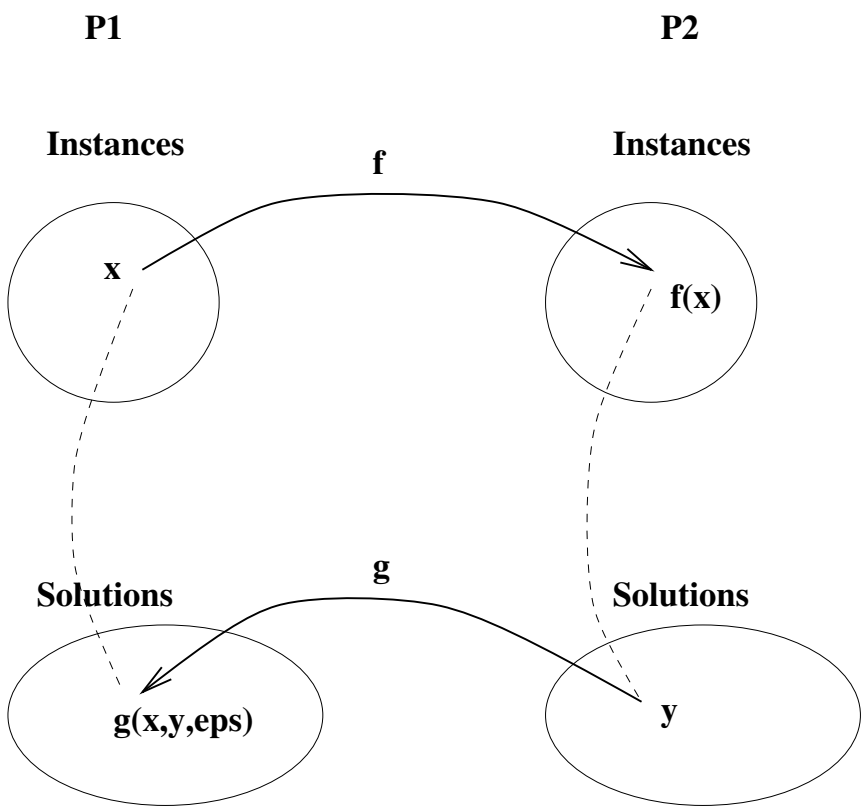

Figure 5: Approximation preserving reduction.

For a proof see [8].

Definition 4.10 *A problem $P_1 \in \mathcal{NPO}$ (respectively, $P_1 \in \mathcal{APX}$) is $\mathcal{NPO}$–complete (respectively $\mathcal{APX}$–complete) if, for any $P_2 \in \mathcal{NPO}$ (respectively, $P_2 \in \mathcal{APX}$), $P_2 \leq_{\mathcal{PTAS}} P_1$.*

The above Lemma shows that the reduction we have introduced is really capable of preserving the level of approximability. Moreover, a consequence of the definition of $\mathcal{NPO}$–completeness (respectively $\mathcal{APX}$–completeness) is that an $\mathcal{NPO}$–complete (respectively $\mathcal{APX}$–complete) problem does not belong to $\mathcal{APX}$ (respectively to $\mathcal{PTAS}$).

4.2 Johnson's approximate algorithms

Let us now present the two first approximate algorithms for *MAX W–SAT*. They were proposed by Johnson [52] and use *greedy* construction strategies.

The original paper [52] demonstrated for both of them a performance ratio 1/2. Recently it has been proved in [21] that the second one reaches a performance ratio 2/3. The two algorithms by Johnson are presented for the unweighted case: it is a simple exercise to add weights.

The first algorithm chooses, at each step, the literal that occurs in the maximum number of clauses. If the literal is positive, the corresponding variable is set to **true**; if the literal is negative, the corresponding variable is set to **false**. The clauses satisfied by the literal are deleted from the formula and the algorithm stops when the formula is satisfied or all variables have been assigned values. More formally, this procedure is developed in algorithm GreedyJohnson1 of Fig. 6.

```
GreedyJohnson1
 Input:  Boolean CNF formula C = {C_1, C_2, ..., C_m};
 Output: Truth assignment U;
△ The satisfied clauses will be incrementally inserted in the set S;
△ U is the truth assignment;
△ for every literal l, u(l) is the corresponding variable;
1    S ← ∅;  Left ← C;  V ← {u | u variable in C};
2    repeat
3        ⌈ Find l, with u(l) ∈ V, that is in max. no. of clauses in Left
4        | Solve ties arbitrarily
5        | Let {C_{l_1}, ..., C_{l_k}} be the clauses in which l occurs
6        | S ← S ∪ {C_{l_1}, ..., C_{l_k}}
7        | Left ← Left \ {C_{l_1}, ..., C_{l_k}}
8        | if l is positive  then u(l) ← true else u(l) ← false
9        ⌊ V ← V \ {u(l)}
10   until no literal l with u(l) ∈ V is contained in any clause of Left
11   if V ≠ ∅ then forall u ∈ V do u ← true
12   return U
```

Figure 6: The GreedyJohnson1 algorithm, a $k/(k+1)$-approximate algorithm.

Theorem 4.3 *Algorithm* GreedyJohnson1 *is a polynomial time 1/2-approximate algorithm for MAX–SAT .*

Proof. One can prove that, given a formula with m clauses, algorithm GreedyJohnson1 always satisfies at least $m/2$ clauses, by induction on the number of variables. Because no optimal solution can be larger than

m, the theorem follows. The result is trivially true in the case of one variable. Let us assume that it is true in the case of $i-1$ variables ($i > 1$) and let us consider the case in which one has i variables. Let u be the last variable to which a truth value has been assigned. We can suppose that u appears positive in k_1 clauses, negative in k_2 clauses and does not appear in $m - k_1 - k_2$ clauses. Without loss of generality suppose that $k_1 \geq k_2$. Then, by inductive hypothesis, algorithm GreedyJohnson1 allows us to choose suitable values for the remaining $i-1$ variables in such a way to satisfy at least $(m - k_1 - k_2)/2$ clauses; if according to the algorithm we now choose $u =$ **true** we satisfy

$$\frac{m - k_1 - k_2}{2} + k_1 \geq \frac{m}{2}$$

clauses.

■

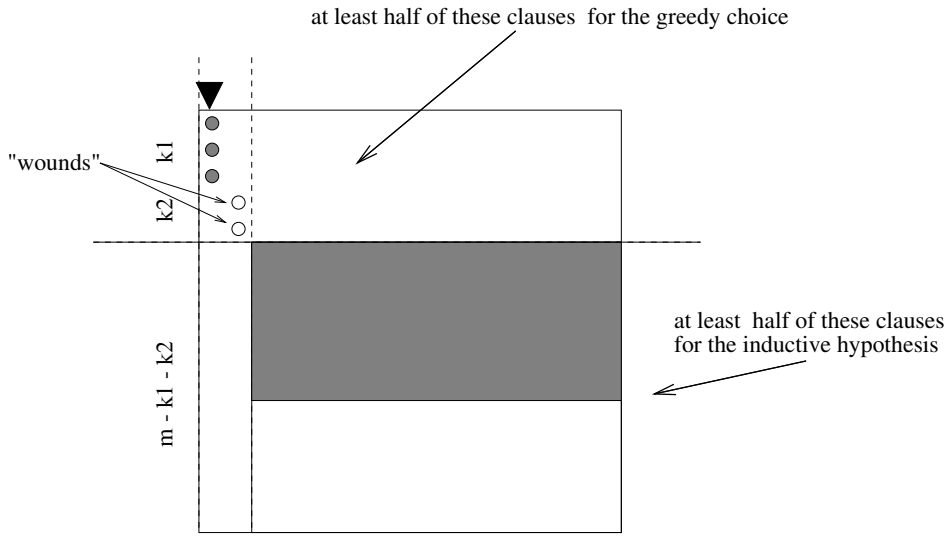

Figure 7: Illustration of the GreedyJohnson1 algorithm.

Let us note that one does not use the fact that the chosen literal occurs in the *maximum* number of clauses for the above proof. What is required is that, given an unset variable that appears in at least an unsatisfied clause, the variable is set to **true** or **false** in a way that maximizes the number of newly satisfied clauses.

This result can be made more specific by considering the number of variables in a clause.

Theorem 4.4 *Let k be the minimum number of variables occurring in any clause of the formula. For any integer $k \geq 1$, algorithm* GreedyJohnson1 *achieves a feasible solution y of an instance x such that*

$$\frac{m(x,y)}{m^*(x)} \geq 1 - \frac{1}{k+1}.$$

Proof. Because of the greediness, when literal l is picked in line 3 of Fig. 6, the number of newly satisfied clauses is at least as large as the number of new *wounds*, defined as the number of occurrences of literal $\bar{l}$ in clauses of Left that will never be matched in the future steps, given the choice of l, see Fig. 7. When the algorithm halts, the only clauses remaining in Left are those that have a number of wounds equal to the number of their literals, and hence are *dead*. This means that, when the algorithm halts, there are at least $k|\text{Left} \mid$ wounds, and therefore $|S| \geq k|\text{Left} \mid$. Thus $m^* \leq m = |S| + |\text{Left} \mid \leq \frac{(k+1)}{k}|S|$. The bound follows.
■

Note that, according to the definition of performance ratio, algorithm GreedyJohnson1 is $\frac{k}{k+1}$–approximate. In particular, for $k = 1$, the performance ratio is $1/2$, for $k = 2$ the performance ratio is $2/3$, for $k = 3$ the performance ratio is $3/4$ and so on. This means that the goodness of the algorithm improves for larger values of k. Therefore the worst case is given by $k = 1$, that is, when one has unit clauses (clauses with just one literal).

Johnson introduced a second algorithm (GreedyJohnson2). This algorithm improves the performance ratio and obtains a bound $2/3$ [21]. Until very recently, only a performance ratio $1/2$ was demonstrated [52]. The original theorem in [52] is here presented, because of its simplicity and paradigmatic nature and because it gives a better performance as a function of k, the minimum number of literals in some clause. In the algorithm one associates a *mass* $w(C_i) = 2^{-|C_i|}$ to each clause. The term *mass* is used instead of the original term "weight" in order to avoid confusions with the clause *weight* in the *MAX W–SAT* problem. The mass will be proportional to the weight in the version of the algorithm for the *MAX W–SAT* problem ($w(C_i) = w_i \ 2^{-|C_i|}$). In [52] the analysis of the performance of algorithm GreedyJohnson2 leads to the following:

Theorem 4.5 *Let k be the minimum number of clauses occurring in any clause of the formula. For any integer $k \geq 1$, algorithm* GreedyJohnson2 *achieves a feasible solution y of an instance x such that*

$$\frac{m(x,y)}{m^*(x)} \geq 1 - \frac{1}{2^k}.$$

```
GreedyJohnson2
 Input:  Boolean CNF formula C = {C_1, C_2, ..., C_m};
 Output: Truth assignment U;
△ The satisfied clauses will be incrementally inserted in the set S;
△ U is the truth assignment;
△ for every literal l, let u(l) be the corresponding variable;
1     S ← ∅;  Left ← C;  V ← {u | u variable in C};
2     Assign to each clause C_i a mass w(C_i) = 2^{-|C_i|}
3     repeat
4         Determine u ∈ V, appearing in at least a clause ∈ Left
5         Let CT be the clauses ∈ Left cont. u, CF those cont. ū
6         if Σ_{C_i∈CT} w(C_i) ≥ Σ_{C_i∈CF} w(C_i) then
7             u(l) ← true
8             S ← S ∪ CT
9             Left ← Left \ CT
10            forall C_i ∈ CF do w(C_i) ← 2 · w(C_i)
11        else
12            u(l) ← false
13            S ← S ∪ CF
14            Left ← Left \ CF
15            forall C_i ∈ CT do w(C_i) ← 2 · w(C_i)
16    until no literal l in any clause of Left is such that u(l) is in V
17    if V ≠ ∅ then forall u ∈ V do u ← true
18    return U
```

Figure 8: The GreedyJohnson2 algorithm, a $(1 - 1/2^k)$-approximate algorithm.

Proof. Initially, because each clause has at least k literals, the total mass of all the clauses in Left cannot exceed $m/2^k$. During each iteration, the total mass of the clauses in Left cannot increase. In fact, the mass removed from Left is at least as large as the mass added to those remaining clauses which receive new *wounds*, see lines 6–15 of Fig. 8. Therefore, when the algorithm halts, the total mass still cannot exceed $m/2^k$. But each of the *dead* clauses in Left when the algorithm halts must have been wounded as many times as it had literals, hence must have had its mass doubled that many times, and so must have final mass equal to one. Therefore $|\text{Left}| \leq m/2^k$, and so $|S| \geq m(1 - 1/2^k)$ and the bound follows.
■

Again, for larger values of k, algorithm GreedyJohnson2 obtains better performance ratios and, generally speaking, because $1 - \frac{1}{2^k} > 1 - \frac{1}{k+1}$ for any integer $k \geq 2$, algorithm GreedyJohnson2 has a better performance than that of algorithm GreedyJohnson1.

The performance ratio 2/3 has been proved in a paper by Chen, Friesen, and Zheng [21]. Because they consider the *MAX W–SAT* problem, line 2 in Fig. 8 must be modified to take the weights w_i into account: the *mass* becomes $w(C_i) = w_i \; 2^{-|C_i|}$. The preceding bound 1/2 depends on the fact that the only upper bound used in the above proofs was given by the total weight of the clauses; of course this upper bound can be far from the optimal value. The novelty of the approach of [21] is that the performance ratio can be derived by using the correct value of the optimal solution. In order to prove that algorithm GreedyJohnson2 has this better performance ratio let us introduce a generalization of the algorithm. It is important to stress that this generalization is introduced to perform a more accurate analysis of the performance ratio and it is used in the following as a theoretical tool.

The difference between GreedyJohnson2 and its generalization is rather subtle. The generalized algorithm, that we denote as GenJohnson2, considers an arbitrary Boolean array $b[1..n]$ of size n as additional input, and examines b to decide what to do if an equality is present in line 6 of Fig. 8. Let us assume that the variable one is considering is u_j. In line 6 of GreedyJohnson2 in Fig. 8, when $\sum_{C_i \in \mathbf{CT}} w(C_i) = \sum_{C_i \in \mathbf{CF}} w(C_i)$, the **if** condition is true and u_j is set to **true**. Now, instead, when one obtains an equality one considers two different cases: if the variable $b[j]$ is **true** u_j is set to **true**; if the variable $b[j]$ is **false** u_j is set to **false**.

This generalized algorithm is then used in the proof with this Boolean array equal to the optimal assignment. Of course the optimal assignment cannot be derived in polynomial time but here we are not interested in running an algorithm but in performing a theoretical analysis.

We will prove that GenJohnson2 has a performance ratio 2/3 and this fact will imply that also GreedyJohnson2 has performance ratio 2/3.

Let us give some definitions needed in the proof.

Definition 4.11

- *A literal is positive if it is a Boolean variable u_i for some i.*
- *A literal is negative if it the negation $\bar{u_i}$ of a Boolean variable for some i.*

Definition 4.12 *Assume that algorithm* GenJohnson2 *is applied to a formula* **C** *and consider a fixed moment in the execution.*

- *A literal l is active if it has not been assigned a truth value yet.*
- *A clause C_j is killed if all literals in C_j are assigned value* **false**.
- *A clause C_j is negative if it is neither satisfied nor killed, and all active literals in C_j are negative literals.*

Definition 4.13 *Let $0 \leq t \leq n$. Assume that in* GenJohnson2 *the t–th iteration has been completed (a truth assignment has been given to t variables). Then $\mathbf{S}^t$ denotes the set of satisfied clauses, $\mathbf{K}^t$ denotes the set of killed clauses, $\mathbf{N}_i^t$ denotes the set of negative clauses with exactly i active liberals.*

Without less of generality, one assumes that each clause in the formula has at most r literals. The proof of the performance ratio 2/3 depends on the following Lemma.

Given a set of clauses $\mathbf{C}$, let us define as $w(\mathbf{C})$ the sum of the weights of all clauses of $\mathbf{C}$.

Lemma 4.6 *For any formula $\mathbf{C}$ of MAX W–SAT and for any Boolean array $b[1..n]$, when the algorithm* GenJohnson2 *is applied on $\mathbf{C}$ the following inequality holds at all iterations $0 \leq t \leq n$:*

$$w(\mathbf{S}^t) \geq 2w(\mathbf{K}^t) + \sum_{i=1}^{r} \frac{1}{2^{i-1}} w(\mathbf{N}_i^t) - A_0 \tag{3}$$

where $A_0 = \sum_{i=1}^{r} \frac{1}{2^{i-1}} w(\mathbf{N}_i^0)$

The proof of the Lemma proceeds by induction on t and can be found in [21].

Theorem 4.7 *The performance ratio of algorithm* GreedyJohnson2 *is* 2/3.

Proof. Let $\mathbf{C}$ be an instance of *MAX W–SAT* and let U_0 an optimal truth assignment for $\mathbf{C}$. Now one considers another formula $\mathbf{C}'$ that is derived from $\mathbf{C}$ as follows. If $U_0(u_t) =$ **false** for a variable u_t then one negates u_t (u_t and $\bar{u_t}$ are interchanged) in $\mathbf{C}'$. No change on the weights is done. Therefore there exists a one–to–one correspondence between the set of clauses in $\mathbf{C}$ and the set of clauses in $\mathbf{C}'$; moreover the corresponding clauses have the same weight. In addition, the Boolean array $b[1..n]$ is constructed such that $b[j] =$ **false** if and only if $U_0(u_j) =$ **false**.

It is easy to see (for the details, see again [21]) that

- the weight of an optimal assignment to $\mathbf{C}'$ is equal to the weight of an optimal assignment to $\mathbf{C}$.
- the truth assignment for $\mathbf{C}$ found by GreedyJohnson2 and the truth assignment for $\mathbf{C}'$ found by GenJohnson2 have the same weight.

This means that, if we prove that GenJohnson2 has a performance ratio 2/3 on the formula $\mathbf{C}'$, the theorem is shown.

Note that the truth assignment U_0' for $\mathbf{C}'$ that gives value **true** to all variables corresponds to the optimal truth assignment U_0 for $\mathbf{C}$. Therefore U_0' is optimal for $\mathbf{C}'$.

When GenJohnson2 stops, that is, for $t = n$, $\mathbf{S}^n$ is the set satisfied by the algorithm and $\mathbf{K}^n$ is the set of clauses not satisfied. $\mathbf{N}_i^n$ is the empty set for any i.

Applying the inequality 3 of Lemma 4.6 to this case, one obtains:

$$w(\mathbf{S}^n) \geq 2w(\mathbf{K}^n) - A_0. \tag{4}$$

On the other hand, A_0 can be upperbounded in the following way:

$$A_0 = \sum_{i=1}^{r} \frac{1}{2^{i-1}} w(\mathbf{N}_i^0) \leq \sum_{i=1}^{r} w(\mathbf{N}_i^0) \leq 2 \sum_{i=1}^{r} w(\mathbf{N}_i^0). \tag{5}$$

From inequalities 4 and 5 one has:

$$\frac{3}{2} w(\mathbf{S}^n) \geq w(\mathbf{S}^n) + w(\mathbf{K}^n) - \sum_{i=1}^{r} w(\mathbf{N}_i^0) \tag{6}$$

Note that, on one hand, $w(\mathbf{S}^n)$ is the weight of the truth assignment found by GenJohnson2. On the other hand, $\mathbf{S}^n \cup \mathbf{K}^n$ is the whole set of clauses in $\mathbf{C}'$ and the optimal truth assignment U_0' for $\mathbf{C}'$ that gives value **true** to all variables satisfies all clauses in $\mathbf{C}'$ except those belonging to $\mathbf{N}_i^0$ for $i = 1, 2, \ldots, r$.

Therefore an optimal truth assignment for $\mathbf{C}'$ has weight exactly

$$w(\mathbf{S}^n) + w(\mathbf{K}^n) - \sum_{i=1}^{r} w(\mathbf{N}_i^0).$$

Then the inequality 6 says that the weight of the truth assignment found by GenJohnson2 is at least 2/3 of the weight of an optimal assignment to $\mathbf{C}'$. In consequence, the weight of the assignment constructed by the original GreedyJohnson2 algorithm for the instance $\mathbf{C}$ is at least 2/3 of

the weight of an optimal assignment to $\mathbf{C}$, thus proving the theorem.
■

Finally, it is worthwhile to note this performance ratio 2/3 is tight. There are formulae for which GreedyJohnson2 finds a truth assignment such that the ratio is 2/3. Therefore this bound cannot be improved. In [21] a set of formulae with this characteristic has been presented.

Let us consider the following formula $\mathbf{C}_h$ formed by $3h$ clauses with h integer greater than 0 where all the clauses have the same weight:

$$\mathbf{C}_h = \{(u_{3k+1} \vee u_{3k+2}), (u_{3k+1} \vee u_{3k+3}), (\overline{u}_{3k+1}) | 0 \leq k \leq h-1\}$$

GreedyJohnson2 gives value **true** to all variables so satisfying $2h$ clauses,that is $(u_{3k+1} \vee u_{3k+2})$, $(u_{3k+1} \vee u_{3k+3})$ for $0 \leq k \leq h-1$. On the other hand, the truth assignment u_{3k+1} =**false**, $u_{3k+2} = u_{3k+3}$ =**true** for $0 \leq k \leq h-1$ satisfies all the $3h$ clauses of the formula.

4.3 Randomized algorithms for *MAX W–SAT*

4.3.1 A randomized 1/2–approximate algorithm for *MAX W–SAT*

One of the most interesting approaches in the design of new algorithms is the use of *randomization*. During the computation, random bits are generated and used to influence the algorithm process.

In many cases randomization allows to obtain better (expected) performance or to to simplify the construction of the algorithm. Particularly in the field of approximation, randomized algorithms are widely used and, for many problems, the algorithm can be "derandomized" in polynomial time while preserving the approximation ratio. However, it is important to note that, often, the derandomization leads to algorithms which are very complicated in practice.

Let us now use this approach to present more efficient approximate algorithms for *MAX W–SAT*. More precisely, this section introduces two different randomized algorithms that achieve a performance ratio of 3/4. Moreover, it is possible to derandomize these algorithms, that is, to obtain deterministic algorithms that preserve the same bound 3/4 for every instance.

The derandomization is based on the *method of conditional probabilities* that has revealed its usefulness in numerous cases and is a general technique that often permits to obtain a deterministic algorithm from a randomized one while preserving the quality of approximation.

Let us first present the algorithm RANDOM, a simple randomized algorithm, that, while just achieving a performance ratio 1/2, will be used in the following subsections as an ingredient to reach the performance ratio 3/4.

RANDOM
Input: Set $\mathbf{C}$ of weighted clauses in conjunctive normal form
Output: Truth assignment U, $\mathbf{C}'$, $\sum_{C_j \in \mathbf{C}'} w_j$
1 Independently set each variable u_i to **true** with probability 1/2
2 Compute $\mathbf{C}' = \{C_j \in \mathbf{C} : C_j \text{ is satisfied } \}$
3 Compute $\sum_{C_j \in \mathbf{C}'} w_j$

Figure 9: The RANDOM algorithm, a randomized $(1 - 1/2^k)$-approximate algorithm.

Because the algorithm is randomized, one is interested in the *expected* performance when the algorithm is run with different sequences of random bits (i.e., with different random assignments).

Lemma 4.8 *Given an instance of MAX W–SAT in which all clauses have at least k literals, the expected weight W of the solution found by algorithm* RANDOM *is such that*

$$W \geq (1 - \frac{1}{2^k}) \sum_{C_j \in \mathbf{C}} w_j.$$

Proof. The probability that any clause with k literals is not satisfied by the assignment found by the algorithm is 2^{-k} (all possible k matches must fail). Therefore the probability that a clause is satisfied is $1 - 2^{-k}$. Then

$$W = (1 - \frac{1}{2^k}) \sum_{C_j \in \mathbf{C}} w_j.$$

■

As an immediate consequence of Lemma 4.8, one obtains the following Corollary.

Corollary 4.9 *Algorithm* RANDOM *finds a solution for MAX W–SAT whose expected value is at least one half of the optimum value.*

The performance of algorithm RANDOM is the same, in a probabilistic setting, as that of algorithm GREEDYJOHNSON2.

Actually it is possible to show that, by applying the method of conditional probabilities to algorithm RANDOM, one essentially obtains algorithm GREEDYJOHNSON2. In this way the algorithm RANDOM is derandomized.

Let us consider the following greedy algorithm that it is described informally and divided in two phases:

First phase (initialization). Assuming (as in algorithm RANDOM) that every variable u_i is **true** with probability 1/2, for every clause C_i compute the probability d_i that C_i is not satisfied. According to Lemma 4.8 this probability is $\frac{1}{2^k}$ where k is the number of literals occurring in C_i.

Second phase Given a variable u_j that has not been assigned a value, let **CT** be the set of remaining clauses that contain u_j and let **CF** be the set of remaining clauses that contain $\overline{u_j}$.

If $\sum_{C_i \in \mathbf{CT}} w_i d_i \geq \sum_{C_i \in \mathbf{CF}} w_i d_i$ then assign to u_j **true**, otherwise assign false. In the first case remove all clauses of **CT** from the formula and double d_i for all clauses C_i of **CF**; in the second case remove all clauses of **CF** from the formula and double d_i for all clauses C_i of **CT**.

It is immediate to note that the truth assignment computed by such an algorithm has weight at least equal to the expected value W of the solution found by algorithm RANDOM. Moreover the algorithm is deterministic.

On the other hand, the two phases correspond to what is made in algorithm GREEDYJOHNSON2.

The approach shown for derandomizing RANDOM can be applied to many randomized algorithms. For the sake of simplicity, let us assume that the input is given by n Boolean variables. Now let E be the expected value of the solutions achieved by a randomized algorithm A. One is now interested in finding a deterministic algorithm B that achieves a solution of value E in polynomial time. In such a way A has been efficiently derandomized.

The algorithm B consists of n iterations and, at each iteration, the value of a variable is determined. The Boolean value of the i-th variable is found in the following way: given the values of variables $u_1, \ldots, u_{i-1}$, we set $u_i = 1$ and we compute the expected weight of clauses satisfied by that truth assignment; then we compute the expected weight of clauses satisfied by the truth assignment in which one has $u_i = 0$, given the current assignment to $u_1, \ldots, u_{i-1}$. We assign u_i the value that maximizes the conditional expectation.

After n iterations, a truth assignment is found deterministically. If we are able to compute each conditional expectation in polynomial time, the algorithm runs in polynomial time and has found an approximate solution whose value is at least E.

Coming back to algorithm RANDOM, one has that, for $k = 1$, the algo-

rithm achieves an expected performance ratio 1/2. The performance of the algorithm improves if we increase the number of literals. In particular, for $k = 2$, that is for formulae which do not contain unit clauses, one obtains an expected value which is at least 3/4 of the optimal value. Therefore if one could discard unit clauses, one would already have a 3/4-approximate algorithm for *MAX W–SAT*, after applying the derandomization. This observation will reveal its usefulness in the following.

4.3.2 A randomized 3/4-approximate algorithm for *MAX W–SAT*

This subsection presents an algorithm that considerably improves the performance of algorithm RANDOM, and obtains a performance ratio 3/4.

First of all we consider a generalization of algorithm RANDOM. In the previous case the value of every variable was chosen randomly and uniformly, that is with probability 1/2; now the value of variable u_i is chosen with probability p_i, obtaining algorithm GENRANDOM.

GENRANDOM
Input: Set $\mathbf{C}$ of weighted clauses in conjunctive normal form
Output: Truth assignment U, $\mathbf{C}'$, $\sum_{C_j \in \mathbf{C}'} w_j$
1 Independently set each variable u_i to **true** with probability p_i
2 Compute $\mathbf{C}' = \{C_j \in \mathbf{C} : C_j \text{ is satisfied } \}$
3 Compute $\sum_{C_j \in \mathbf{C}'} w_j$

Figure 10: The GENRANDOM algorithm.

The expected number of clauses satisfied by algorithm GENRANDOM can be immediately computed as a function of p_i.

Lemma 4.10 *The expected weight W of the set of clauses $\mathbf{C}$ is:*

$$W = \sum_{C_j \in C} w_j (1 - \prod_{i \in U_j^+} (1 - p_i) \prod_{i \in U_j^-} p_i)$$

where $U_j^+ (U_j^-)$ denotes the set of indices of the variables appearing unnegated (negated) in the clause C_j.

Proof. It is an obvious generalization of the proof given in the particular case $p_i = 1/2$.
■

Now, if one manages to find suitable values p_i such that $W \geq 3/4\ m^*(\mathbf{C})$ for every formula $\mathbf{C}$, one would obtain a 3/4-approximate randomized algorithm.

To aim at this result, let us consider the representation of the instances of *MAX W–SAT* as instances of an integer linear programming problem (*ILP*) already presented in Section 2:

$$max \sum_{C_j \in \mathbf{C}} w_j z_j$$

subject to :

$$\sum_{i \in U_j^+} y_i + \sum_{i \in U_j^-} (1 - y_i) \geq z_j, \forall C_j \in \mathbf{C}$$

$$y_i \in \{0,1\}, 1 \leq i \leq n$$

$$z_j \in \{0,1\}, \forall C_j \in \mathbf{C}$$

Let $u_1, \ldots, u_n$ be the Boolean variables appearing in the formula. An instance of *MAX W–SAT* is equivalent to an instance of *ILP* if we choose the following conditions:

- $y_i = 1$ iff variable u_i is **true**;
- $y_i = 0$ iff variable u_i is **false**;
- $z_j = 1$ iff clause C_j is satisfied;
- $z_j = 0$ iff clause C_j is not satisfied.

The linear inequality states the fact that a clause can be satisfied ($z_j = 1$) only if at least one of its literals is matched.

One cannot compute the optimal value in polynomial time because *ILP* is $\mathcal{NP}$–complete. However let us consider the *LP* relaxation (by *relaxation* one means that the set of admissible solution increases with respect to that of the original problem) in which one relaxes the conditions $y_i, z_j \in \{0,1\}$ with the new constraints $0 \leq y_i, z_j \leq 1$. It is known that *LP* can be solved in polynomial time finding a solution

$$(y^* = (y_1^*, \ldots, y_n^*), z^* = (z_1^*, \ldots, z_m^*))$$

with value $m^*_{LP}(x) \geq m^*_{ILP}(x)$, for every instance x, where $m^*_{LP}(x)$ and $m^*_{ILP}(x)$ denote the optimal value of the *LP* and *ILP* instances,

respectively. The upper bound is obvious given that the set of admissible solutions is enlarged by the relaxation.

Let us consider algorithm GenApprox, see Fig. 11, that works as follows: first it solves the linear programming relaxation and so computes the optimal values (y^*, z^*); then, given a function g to be specified later, it computes, for each i, $i = 1, \ldots, n$, the probabilities $p_i = g(y^*{}_i)$. By Lemma 4.10 we know that a solution of weight:

$$W = \sum_{C_j \in \mathbf{C}} w_j (1 - \prod_{i \in U_j^+} (1 - p_i) \prod_{i \in U_j^-} p_i)$$

must exist; by applying the method of conditional probabilities, such solution can be deterministically found.

GenApprox
Input: Set $\mathbf{C}$ of clauses in disjunctive normal form
Output: Set $\mathbf{C}'$ of clauses, $W = \sum_{C_j \in \mathbf{C}'} w_j$
1 Express the input $\mathbf{C}$ as an equivalent instance x of *ILP*
2 Find the optimum value y^*, z^* of x in the linear relaxation
3 Choose $p_i \leftarrow g(y^*{}_i)$, $i = 1, 2, \ldots, n$, for a suitable function g
4 $W \leftarrow \sum_{C_j \in \mathbf{C}} w_j (1 - \prod_{i \in X_j^+} (1 - p_i) \prod_{i \in X_j^-} p_i)$
5 Apply the method of conditional probabilities to find
6 a feasible solution $\mathbf{C}' = \{C_j \in \mathbf{C} : C_j \text{ is satisfied}\}$ of value W

Figure 11: The GenApprox algorithm, deterministic version.

If the function g can be computed in polynomial time then algorithm GenApprox runs in polynomial time. In fact the linear relaxation can be solved efficiently and the computation of the feasible solution can be computed in polynomial time with the method of conditional probabilities explained before.

The quality of approximation naturally depends on the choice of the function g. Let us suppose that this function finds suitable values such that:

$$(1 - \prod_{i \in U_j^+} (1 - p_i) \prod_{i \in U_j^-} p_i) \geq \frac{3}{4} z_j^*.$$

If this inequality is satisfied, then the algorithm is a 3/4-approximate algo-

rithm for *MAX W–SAT*. In fact one has :

$$\begin{aligned} W &= \sum_{C_j \in \mathbf{C}} w_j (1 - \prod_{i \in U_j^+} (1 - p_i) \prod_{i \in U_j^-} p_i) \geq \frac{3}{4} \sum_{C_j \in \mathbf{C}} w_j z_j^* = \\ &= \frac{3}{4} m_{LP}^* (x) \geq \frac{3}{4} m_{ILP}^* (x) \end{aligned}$$

More generally if one has :

$$(1 - \prod_{i \in U_j^+} (1 - p_i) \prod_{i \in U_j^-} p_i) \geq \alpha z_j^*$$

one obtains a α-approximate algorithm.

A first interesting way of choosing the function g consists of applying the following technique, called *Randomized Rounding*, to get an integral solution from a linear programming relaxation. In order to get integer values one rounds the fractional values, that is each variable y_i is independently set to 1 (corresponding to the Boolean variable u_i being set to **true**) with probability y_i^*, for each $i = 1, 2, \ldots, n$. Hence the use of the randomized rounding technique is equivalent to choosing $p_i = g(y_i^*) = y_i^*, i = 1, 2, \ldots, n$.

Lemma 4.11 *Given the optimal values (y^*, z^*) to LP and given any clause C_j with k literals, one has*

$$(1 - \prod_{i \in U_j^+} (1 - y_i^*) \prod_{i \in U_j^-} y_i^*) \geq \alpha_k z_j^*$$

where

$$\alpha_k = 1 - \left(1 - \frac{1}{k}\right)^k .$$

Proof. Let us consider a clause C_j and, for the sake of simplicity, let us assume that every variable is unnegated. If a variable u_i would appear negated in C_j, one could substitute u_i by its negation $\overline{u}_i$ in every clause and also replace y_i by $1 - y_i$. So we can assume $C_j = u_1 \vee \cdots \vee u_k$ with the associated condition $y_1^* + \cdots + y_k^* \geq z_j^*$. The Lemma is proved by showing that:

$$1 - \prod_{i=1}^{k} (1 - y_i^*) \geq \alpha_k z_j^*.$$

In the proof we exploit the geometric inequality based on the properties of the arithmetic mean: given a finite set of nonnegative numbers $\{a_1, \ldots, a_k\}$,

$$\frac{a_1 + \cdots + a_k}{k} \geq \sqrt[k]{a_1 a_2 \cdots a_k}.$$

Now we apply the geometric inequality to the set $\{1 - y_1^*, \ldots, 1 - y_k^*\}$. Because $\sum_{i=1}^{k} \frac{1-y_i^*}{k} = 1 - \frac{\sum_{i=1}^{k} y_i^*}{k}$, one has

$$1 - \prod_{i=1}^{k}(1 - y_i^*) \geq 1 - (1 - \frac{\sum_{i=1}^{k} y_i^*}{k})^k \geq 1 - (1 - \frac{z_j^*}{k})^k.$$

We note that the function $g(z_j^*) = 1 - (1 - \frac{z_j^*}{k})^k$ is concave in the interval $[0, 1]$; hence it is sufficient to prove that $g(z_j^*) \geq \alpha_k z_j^*$ at the extremal points of the interval. Because one has

$$g(0) = 0 \text{ and } g(1) = \alpha_k$$

the Lemma is shown.

■

One can conclude that algorithm GenApprox with the choice $p_i = y_i^*$ reaches an approximation ratio equal to α_k. In particular for $k = 2$, the ratio is 3/4. Note that, because α_k is decreasing with k, algorithm GenApprox is an α_k–approximation algorithm for formulae with *at most* k literals per clause.

Moreover, it is well known that $\lim_{k\to\infty}(1 - \frac{1}{k})^k = \frac{1}{e}$; hence for arbitrary formulae one finds approximate solutions whose value is at least $1 - \frac{1}{e}$ times the optimal value. Because $1 - \frac{1}{e} = 0.632\ldots$, the randomized rounding obtains a better performance than Random, but it looks as if one is far from achieving a 3/4-approximation ratio.

Luckily, with a suitable merging of the above algorithm with Random one obtains the desired performance ratio. Firstly let us recall that Random is a 3/4-approximation algorithm if all clauses have *at least* two literals. On the other hand, GenApprox is a 3/4-approximation algorithm if we work with clauses with *at most* two literals. One algorithm is good for large clauses, the other for short ones. A simple combination consists of running both algorithm and choosing the best truth assignment obtained. Let us now consider the expected value obtained from the combination.

Theorem 4.12 *Let W_1 be the expected weight corresponding to $p_i = 1/2$ and let W_2 be the expected weight corresponding to $p_i = y_i^*$, $i = 1, 2, \ldots, n$. Then one has :*

$$\max(W_1, W_2) \geq \frac{3}{4} m_{LP}^*(x), \text{ for any instance } x.$$

Proof. Because $\max(W_1, W_2) \geq \frac{W_1+W_2}{2}$, it is sufficient to show that $\frac{W_1+W_2}{2} \geq \frac{3}{4} m^*_{LP}(x)$ for any x. Let us denote by $\mathbf{C}^k$ the set of clauses with exactly k literals. By Lemma 4.8, because $0 \leq z^*_j \leq 1$ one has

$$W_1 = \sum_{k\geq 1} \sum_{C_j \in \mathbf{C}^k} \gamma_k w_j \geq \sum_{k\geq 1} \sum_{C_j \in \mathbf{C}^k} \gamma_k w_j z^*_j \tag{7}$$

where $\gamma_k = (1 - \frac{1}{2^k})$.

Moreover, by applying Lemma 4.11, one obtains:

$$W_2 \geq \sum_{k\geq 1} \sum_{C_j \in \mathbf{C}^k} \alpha_k w_j z^*_j. \tag{8}$$

Summing 7 and 8 one has :

$$\frac{W_1 + W_2}{2} \geq \sum_{k\geq 1} \sum_{C_j \in \mathbf{C}^k} \frac{\gamma_k + \alpha_k}{2} w_j z^*_j.$$

We note that $\gamma_1 + \alpha_1 = \gamma_2 + \alpha_2 = 3/2$ and for $k \geq 3$ one has that $\gamma_k + \alpha_k \geq 7/8 + 1 - \frac{1}{e} \geq 3/2$; Therefore:

$$\frac{W_1 + W_2}{2} \geq \sum_{k\geq 1} \sum_{C_j \in \mathbf{C}^k} \frac{3}{4} w_j z^*_j = \frac{3}{4} m^*_{LP}(x).$$

■

Note that it is not necessary to separately apply the two algorithms but it is sufficient to randomly choose one of the two algorithms with probability 1/2, as it is done in algorithm 3/4–Approximate Sat.

Corollary 4.13 *Algorithm* 3/4–Approximate Sat *is a* 3/4*-approximation algorithm for MAX W–SAT.*

Proof. The proof derives from the above theorem and from the use of the method of conditional probabilities.

■

4.3.3 A variant of the randomized rounding technique

This subsection shows that it possible to directly design a 3/4-approximate algorithm for *MAX W–SAT* based on randomized rounding. However, in order to reach this aim, one needs to apply some modifications to the standard technique.

3/4–Approximate Sat
Input: Set $\mathbf{C}$ of clauses in conjunctive normal form
Output: Set $\mathbf{C}'$ of clauses, $W = \sum_{C_j \in \mathbf{C}'} w_j$
1 Express the input $\mathbf{C}$ as an equivalent instance x of *ILP*
2 Find the optimum value (y^*, z^*) of x in the linear relaxation
3 With probability 1/2 choose $p_i = 1/2$ or $p_i = y_i^*, i = 1, 2, \ldots, n$
4 $W \leftarrow \sum_{C_j \in \mathbf{C}} w_j (1 - \prod_{i \in U_j^+} (1 - p_i) \prod_{i \in U_j^-} p_i)$
5 Apply the method of conditional probabilities to find a feasible
6 solution $\mathbf{C}' = \{C_j \in \mathbf{C} : C_j \text{ is satisfied}\}$ of value W

Figure 12: The 3/4–Approximate Sat algorithm: deterministic with performance ratio 3/4.

Let us start again from algorithm GenApprox. One has already seen that, by choosing $g(y_i^*) = y_i^*$, one cannot obtain a performance ratio 3/4. Therefore a different choice of g is necessary.

Let us consider the following definition:

Definition 4.14 *A function $g : [0,1] \longrightarrow [0,1]$ has property 3/4 if*

$$1 - \prod_{i=1}^{l} (1 - g(y_i)) \prod_{i=l+1}^{k} g(y_i) \geq \frac{3}{4} \min(1, \sum_{i=1}^{l} y_i + \sum_{i=l+1}^{k} (1 - y_i)).$$

for any integers k,l with $k \geq l$ and any $y_1, \ldots, y_k \in [0,1]$.

By Lemma 4.11 if a function g with property 3/4 is found, then algorithm GenApprox becomes a 3/4-approximate algorithm. In order to prove the existence of functions with property 3/4 one needs the following lemma:

Lemma 4.14 *A function $g : [0,1] \longrightarrow [0,1]$ has property 3/4 if satisfies the following conditions:*

i) $1 - \prod_{i=1}^{k} (1 - g(y_i)) \geq \frac{3}{4} \min(1, \sum_{i=1}^{k} y_i)$, $\forall k$ *and* $\forall y_i \in [0,1]$
for any integer k and $y_i \in [0,1], i = 1, 2, \ldots, n$.

ii) $g(y) \leq 1 - g(1 - y)$.

Proof. Given integers k, l with $k \geq l$, let $y_i' = y_i$ for $i = 1 \ldots, l$ and $y_i' = 1 - y_i$

for $i = l+1, \ldots, k$. one has

$$\begin{aligned} 1 - \prod_{i=1}^{l}(1-g(y_i)) \prod_{i=l+1}^{k} g(y_i) &\geq 1 - \prod_{i=1}^{l}(1-g(y_i)) \prod_{i=l+1}^{k} (1-g(1-y_i)) \\ &= 1 - \prod_{i=1}^{k}(1-g(y'_i)) \geq \frac{3}{4}\min(1, \sum_{i=1}^{k} y'_i) \\ &= \frac{3}{4}\min(1, \sum_{i=1}^{l} y_i + \sum_{i=l+1}^{k}(1-y_i)). \end{aligned}$$

■

Lemma 4.15 *The following function g_α verifies property 3/4:*

$$g_\alpha(y) = \alpha + (1-2\alpha)y$$

where $2 - \frac{3}{\sqrt[3]{4}} \leq \alpha \leq \frac{1}{4}$

Proof. It is immediate to verify that g_α satisfies (ii). In order to prove that also the condition (i) of the lemma 4.14 is verified, one has

$$\begin{aligned} 1 - \prod_{i=1}^{k}(1-g_\alpha(y_i)) &= 1 - \prod_{i=1}^{k}(1-\alpha-(1-2\alpha)y_i) \\ &\geq 1 - \left(1-\alpha-(1-2\alpha)\frac{\sum_{i=1}^{k} y_i}{k}\right)^k \end{aligned}$$

where one exploits the fact that the arithmetic mean is greater or equal than the geometric mean of k numbers. Let us define $Y = \dfrac{\sum_{i=1}^{k} y_i}{k}$. It is sufficient to prove that:

$$h_k(Y) = 1 - (1-\alpha-(1-2\alpha)Y)^k \geq \frac{3}{4}\min(1, kY) \ \forall Y \in [0,1].$$

Let us first prove the result in the interval $[0, 1/k]$. Because the function h_k is concave, the minimum value is reached at one of the extremal points of the interval. This means that it is sufficient to check that the inequality holds for $Y = 0$ and $Y = 1/k$. This fact is immediately true for $Y = 0$. In the other case it is sufficient to prove that:

$$1 - (1-\alpha-(1-2\alpha)\frac{1}{k})^k \geq 3/4, \ \forall k \geq 1.$$

For $k = 1$ the inequality is satisfied if $\alpha \leq 1/4$. On the other side for $k = 2$ the inequality becomes an identity and therefore it is always satisfied. For $k \geq 3$, by easy algebraic steps, one needs to show that

$$\alpha \geq \frac{k - 1 - k4^{-1/k}}{k - 2}. \tag{9}$$

Note that the right-hand-side of the inequality is a function decreasing in k. Moreover, for $k = 3$, inequality 9 holds for $\alpha \geq 2 - \frac{3}{\sqrt[3]{4}}$. Therefore the proof is completed for the interval $[0, \frac{1}{k}]$.

To finish the proof of the lemma it is sufficient to observe that, for any given k, the function $h_k(Y)$ is increasing in the interval $(\frac{1}{k}, 1]$.
■

Lemmata 4.11 and 4.15 imply the following theorem:

Theorem 4.16 *Given α such that $2 - \frac{3}{\sqrt[3]{4}} \leq \alpha \leq \frac{1}{4}$, algorithm* GenApprox *with the choice $p_i = \alpha + (1 - 2\alpha)y_i^*$ is a 3/4-approximate algorithm for MAX W–SAT.*

4.4 Another $\frac{3}{4}$–approximate algorithm by Yannakakis

It is possible to achieve a performance ratio $\frac{3}{4}$ also by using a very different approach. In fact Yannakakis [84] introduced an algorithm that exploits network flow techniques and again obtains the performance bound $\frac{3}{4}$.

Because of the complexity of the proofs, we will limit ourselves to consider the case of *MAX W–2–SAT*, by showing that *MAX W–2–SAT* can be approximated with a performance ratio $\frac{3}{4}$. After generalizing the techniques used for *MAX W–2–SAT* to an arbitrary formula, it is still possible to obtain the same bound. About the notation: in this section we consider the *MAX W–2–SAT* problem with clauses containing *one or two* literals. As already shown, if every clause has at least two literals, a simple greedy algorithm finds the ratio $\frac{3}{4}$. Therefore, if one can *eliminate the unit clauses* from instances of *MAX W–2–SAT*, one obtains the desired bound.

More precisely, given a set $\mathbf{C}$ of clauses with at most two literals per clause, one fixes the truth value for a subset of the variables and builds a set $\mathbf{C}'$ of clauses with exactly two literals per clause in which the remaining variables occur. $\mathbf{C}'$ is constructed in such a way that a truth assignment for $\mathbf{C}'$ with approximation ratio R gives a truth assignment for $\mathbf{C}$ (when combined with the truth values of fixed variables) with an approximation ratio at least R.

Formally, one has the following theorem in which we use this notation: $w(\mathbf{C}, \rho)$ is the weight of the formula $\mathbf{C}$ with respect the truth assignment ρ. In this Subsection, for the sake of clarity, we will use different symbols for the set of Boolean variables and the truth assignments.

Theorem 4.17 *Let $\mathbf{C}$ be an instance of MAX W–2–SAT defined over a set U of Boolean variables. It is possible to find in polynomial time a subset V of variables, a truth assignment σ for V, a nonnegative constant h, and a set $\mathbf{C}'$ of clauses with exactly two literals per clause in which only variables belonging to the set $U - V$ occur such that:*

1. *For every truth assignment θ to the set $U - V$ of variables one has $w(\mathbf{C}, \sigma \cup \theta) = w(\mathbf{C}', \theta) + h$ where $\sigma \cup \theta$ is the global truth assignment obtained applying σ to V and θ to $U - V$.*

2. *For every truth assignment ρ to U with restriction θ to $U - V$ one has $w(\mathbf{C}, \rho) \leq w(\mathbf{C}', \theta) + h$*

This theorem says that one does not lose in the level of approximation by choosing the truth assignment σ for the variables in V; an optimal (or near-optimal) truth assignment for $U - V$ together with σ for V gives an optimal (near-optimal) assignment for the entire set U of variables.

Because one knows how to approximate formulae with exactly two literals per clause with a performance ratio 3/4, the above theorem implies the following:

Corollary 4.18 *If MAX W–2–SAT with exactly two literals per clause can be approximated with performance ratio R, then the general MAX W–2–SAT problem can be approximated with performance ratio R.*
In particular, MAX W–2–SAT can be approximated with performance ratio 3/4.

Proof. Assume that it is possible to achieve an approximation ratio R for the formula $\mathbf{C}'$, that is $w(\mathbf{C}', \theta) \geq Rm^*(\mathbf{C}')$. Then the same ratio holds for $\mathbf{C}$ according to the following calculation. By Part 2 of the above Theorem, one has $m^*(\mathbf{C}) \leq m^*(\mathbf{C}') + h$. Moreover, applying Part 1 and because $h \geq 0$ and $R \leq 1$ one obtains $w(\mathbf{C}, \sigma \cup \theta) = w(\mathbf{C}', \theta) + h \geq Rm^*(\mathbf{C}') + h \geq R(m^*(\mathbf{C}') + h) \geq Rm^*(\mathbf{C})$.

Intuitively, the proof of theorem 4.17 is based on the idea of finding a correspondence between formulae and networks so that the weight of a formula is evaluated by computing the maximum flow of a network.

In the sequel we will assume that some basic notions of network flow theory are known. For a clear introduction to this area see, for instance, [73].

Given a formula $\mathbf{C}$, a network $N(\mathbf{C})$ is built in the following way: Every literal in $\mathbf{C}$ becomes a node in $N(\mathbf{C})$; moreover two other nodes are introduced, that is, s (which is the source of the network) and t (which is the sink of the network).

The arcs are defined as follows. First of all, two arcs (u, v) and (w, z) are said to *correspond* to each other if $u = \overline{z}$ and $v = \overline{w}$. In this approach $\overline{\overline{a}} = a$ for an arbitrary literal a and $\overline{s} = t$. Now, given a clause C_i of $\mathbf{C}$ with weight w_i, C_i is associated to two corresponding arcs of $N(\mathbf{C})$, each having capacity $w_i/2$. If $C_i = a$ is a unit clause then its associated arcs are (s, a) and $(\overline{a}, t)$; if $C_i = a \vee b$ is a clause of length two, its associated arcs are $(\overline{a}, b)$ and $(\overline{b}, a)$. Finally note that the source node stands for the constant **true** and the sink node for **false**.

Some other definition is needed.

Definition 4.15
- *A network is symmetric if corresponding arcs have the same capacity.*
- *A flow is symmetric if corresponding arcs have the same flow.*

According to such definition $N(\mathbf{C})$ is symmetric.

Let f^* be the maximum flow in $N(\mathbf{C})$. Now let us consider a new flow f in $N(\mathbf{C})$. If e_{i_1} and e_{i_2} are the two arcs associated to a clause C_i, then f is defined in the following way: $f(e_{i_1}) = f(e_{i_2}) = \frac{f^*(e_{i_1}) + f^*(e_{i_2})}{2}$.

Two introductory lemmata are now presented. The proofs can be found in [84].

Lemma 4.19 *The flow f satisfies the capacity and flow conservation constraints and has maximum value.*

Definition 4.16 *Let G and G' be two formulae defined over the same set of variables. G and G' are said to be equivalent if every truth assignment gives the same weight to the two formulae.*

In the following lemma one will assume that all the considered clauses have the same weight.

Lemma 4.20 *1. Let us consider the following two formulae: $G = \{\overline{u}_i \vee u_{i+1} | i = 0, \ldots, k\}$ and $G' = \{\overline{u}_{i+1} \vee u_i | i = 0, \ldots, k\}$. Then G and G' are equivalent.*

2. *Let us consider the following two formulae:*
   $H = \{u_1\} \cup \{\overline{u}_i \vee u_{i+1} | i = 1, \ldots, k-1\}$ *and* $H' = \{u_k\} \cup \{\overline{u}_{i+1} \vee u_i | i = 1, \ldots, k-1\}$. *Then also* H *and* H' *are equivalent.*

Let us define the *residual network* M with respect to the flow f. Given any arc $e = (u, v)$ of $N(\mathbf{C})$, M contains e with capacity $c(e) - f(e)$ if e is not saturated, where c denotes the capacity in $N(\mathbf{C})$; moreover M contains the reverse arc (v, u) with capacity $f(e)$, again if the capacity is not saturated. No arc going into source s or going out of t is included.

The reversal of two corresponding arcs gives two arcs that are still corresponding to each other. Furthermore M is symmetric because the network $N(\mathbf{C})$ and the flow f are symmetric. Finally note that M is the network $N(\tilde{\mathbf{C}})$ of a formula $\tilde{\mathbf{C}}$ on the same set of variables. If M has an arc (s, l) and hence an arc $(\overline{l}, t)$ of weight w, then $\tilde{\mathbf{C}}$ has a unit clause l with weight $2w$. If M contains the corresponding arcs (a, b) and $(\overline{b}, \overline{a})$ of weight w, then $\tilde{\mathbf{C}}$ contains the clause $\overline{a} \vee b$ with weight $2w$.

Now one is ready to state the following important Lemma:

Lemma 4.21 *Let* f_{opt} *be the value of the maximum flow* f. *For any truth assignment* θ, *one has*

$$w(\mathbf{C}, \theta) = w(\tilde{\mathbf{C}}, \theta) + f_{opt}.$$

Proof. A flow can be decomposed into a set of simple paths $P_1, \ldots, P_l$ from the source to the sink and into a set of cycles $K_1, \ldots, K_m$. Given an arbitrary arc e, the flow $f(e)$ through e is equal to the sum of the weights associated with the paths and cycles containing e. Moreover f_{opt} is equal to the sum of the weights of the paths. For this decomposition, in the case of the residual network M, one has to reverse every path and cycle from $N(\mathbf{C})$. This operation is performed by subtracting the associated weight from the capacities of all the arcs of the path or cycle and summing the weight to the capacities of the arcs in the reverse path or cycle (except for the arcs going into the source or out of the sink).

Let us consider what happens to the clauses in $\mathbf{C}$ after applying this reversal operation. By reversing a cycle K_j with weight w_j, w_j is subtracted from all the clauses that correspond to arcs of the cycle and w_j is added to the clauses corresponding to arcs of the reverse cycle. Part 1 of Lemma 4.20 guarantees that a set of equivalent clauses is obtained. Considering the paths, assume that an arbitrary path P_j consists of arcs $(s, u_1), \ldots, (u_{k-1}, u_k), (u_k, t)$. By Part 2 of Lemma 4.20, the corresponding set of clauses

$\{u_1\} \cup \{\overline{u_i} \vee u_{i+1} | i = 1, \ldots, k-1\} \vee \{\overline{u_k}\}$ is equivalent to the set $\{u_k, \overline{u_k}\} \cup \{\overline{u}_{i+1} \vee u_i | i = 1, \ldots, k-1\}$. $\{u_k, \overline{u_k}\}$ is equivalent to the constant clause **true** while $\{\overline{u}_{i+1} \vee u_i | i = 1, \ldots, k-1\}$ corresponds to the reverse path of P_j. Hence, by reversing a path with weight w_j, the weight is subtracted from the corresponding clauses and added to the clauses of the reverse path in M; moreover, the weight w_j is given to constant clause **true**, so preserving the equivalence.

Globally speaking, one obtains an equivalent set of clauses that consists of $\tilde{\mathbf{C}}$ and the clause **true** with weight equal to the sum of the weights of the paths, that is, f_{opt}.

■

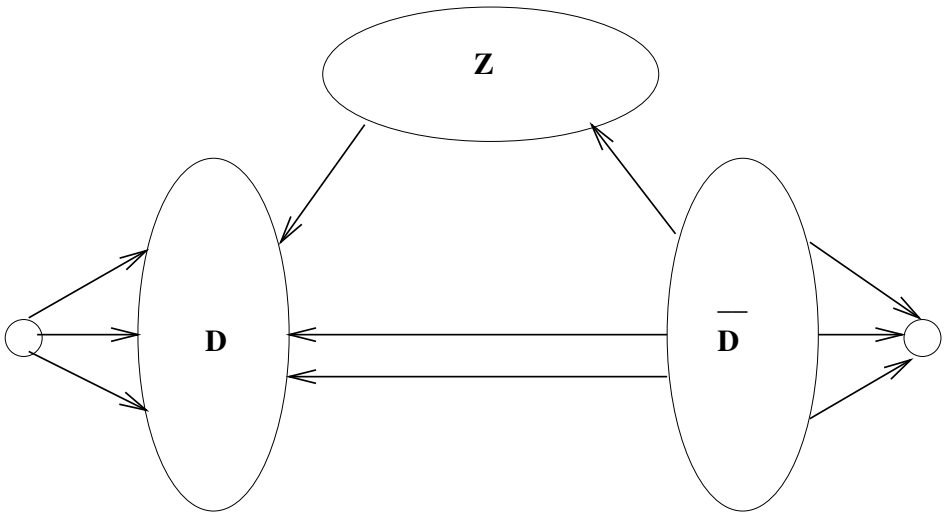

Figure 13: The residual network M.

Note that the residual network M does not contain any path from s to t, because f is a maximum flow (see [73]). Let D be the set of nodes reachable from s in M. The symmetry of M implies that there exists a path from s to a node a if and only if there exists a path from $\overline{a}$ to t. One can conclude that D does not contain any complementary literals because otherwise M would contain a path from s to t. Again, by symmetry, the set of nodes that can reach t is given by the set $\overline{D} = \{\overline{a} | a \in D\}$. The set Z is given by the the remaining nodes of M, that is, by the nodes that do not belong to D or $\overline{D}$. By construction, there are no arcs coming out of D and, by symmetry of M, no arcs going into $\overline{D}$, see Fig. 13.

So one has obtained the following Lemma.

Lemma 4.22 *The set D does not contain any complementary literals. Every clause of $\tilde{\mathbf{C}}$ that contains the negation $\overline{a}$ of a literal a in D, also contains (positively) a literal b in D.*

Exploiting this lemma, if one sets every literal in D to **true**, all the clauses in $\tilde{\mathbf{C}}$ involving such literals and their negations are satisfied. Let d^* be the total weight of these clauses. Lemma 4.21 implies Theorem 4.17 if one does these choices: V is given by those variables with a literal in D, σ is the above truth assignment for V, $\mathbf{C}'$ is the set of remaining clauses of $\tilde{\mathbf{C}}$ that involve only literals from Z and $h = f_{opt} + d^*$. As desired, $\mathbf{C}'$ does not have unit clauses because M does not contain any arc from s to nodes outside D.

Finally it is worthwhile to note that, because the weights of the clauses of $\mathbf{C}$ are integers, it follows that, in general, the weights of $\tilde{\mathbf{C}}$ and $\mathbf{C}'$, are half-integers. Then these weights can become integers if they are doubled. This multiplication by two does not change the problem.

The proof of Theorem 4.17 is therefore completed.

A generalization of this approach allows to eliminate the unit clauses in the case of formulae in which there are clauses of length 3 or more.

4.5 Approximate solution of *MAX W–SAT*: improvements

The approximation ratio 3/4 can be slightly improved by applying different relation techniques. More precisely, Goemans and Williamson [40] introduce a new way to approximate another classical optimization problem: *MAX–CUT* .

Definition 4.17 *MAX–CUT is the following $\mathcal{NPO}$ problem: Given a graph $G = (V, E)$ and a weight $w(e)$ for each $e \in E$, one looks for a subset V' of V that maximizes the sum of the weights of the edges from E that have one endpoint in V' and one endpoint in $V - V'$.*

MAX–CUT can be easily approximated with a performance ratio 1/2 (see, for instance, [84]). For many years it was impossible to improve this bound. Goemans and Williamson devised a new approach. The problem can be represented as a quadratic programming problem. As in the case of *MAX W–SAT*, one can look for some relaxation in order to find a good approximate algorithm. Actually, in the case of *MAX–CUT* , a relaxation

based on the so called *semidefined programming* allows to design an approximation ratio .87856, therefore strongly improving the bound 1/2. In fact, by applying similar techniques, it is possible to reach better results also for *MAX W–SAT*, although the improvement is very small in this case:

Theorem 4.23 *There exists a polynomial time approximate algorithm for MAX W–SAT with a performance ratio* .7584.

For the restricted case of *MAX 2–SAT* , one can obtain a more substantial improvement with the technique of Feige and Goemans [32]. Actually they have obtained a performance ratio 0.931. Coming back to the general problem some other small improvements have been given. Asano [5] (following [6]) has improved the bound to .77. If one considers only *satisfiable MAX W–SAT* instances, Trevisan [83] obtains a 0.8 approximation factor, while Karloff and Zwick [58] claim a 0.875 performance ratio for satisfiable instances of *MAX W–3–SAT*.

4.6 Negative results about approximability

Until now one has shown how to approximate *MAX W–SAT*, obtaining better and better approximation ratios. It is natural, in this framework, to wonder whether it is possible to further improve the approximation properties of *MAX W–SAT*, by showing that the problem belongs to $\mathcal{PTAS}$. We recall that if *MAX W–SAT* belongs to $\mathcal{PTAS}$, one would be able to introduce a polynomial time ε–approximate algorithm for every ε between 0 and 1. Unfortunately it is possible to prove that, unless $\mathcal{P} = \mathcal{NP}$, this is not true.

The negative results for *MAX W–SAT* and, more generally, for $\mathcal{NPO}$ optimization problems are based on the theory of probabilistic checkable proof (for short, *PCP*). This theory was developed in a different context but, surprisingly, can be applied to the field of approximation algorithms allowing to find very interesting negative results. Because of its complexity it is impossible to present this theory. The reader interested in understanding this nice relationship can read the pioneering papers that introduced *PCP*, for instance [3] and [4]. Exploiting this theory many negative results for *MAX W–SAT* have been given. We limit ourselves to present the strongest one which is due to Håstad [50].

Theorem 4.24 *Unless $\mathcal{P} = \mathcal{NP}$ MAX W–SAT cannot be approximated in polynomial time within a performance ratio greater than* 7/8.

More precisely, this result has been obtained for the *MAX W–SAT* problem in which each clause is of length exactly three. Since this version is a particular case of *MAX W–SAT*, of course the result holds in general.

5 A different *MAX–SAT* problem and completeness results

In this section we present another optimization problem again having *SAT* as associated recognition problem. While in *MAX W–SAT* we associate a weight to each clause, now we associate a weight to each variable. More formally we introduce *MAX–VAR SAT* .

MAX–VAR SAT is an $\mathcal{NPO}$ problem in which (I, sol, m, opt) are defined in the following way:

1. I = sets $U = u_1, \ldots, u_n$ of Boolean variables and a collection $\mathbf{C}$ of clauses over U, a set $W = w_1, \ldots, w_n$ of integers (weights) associated to the variables.
2. Given an instance x of I, $sol(x)$ = set of truth assignments to the variables in U that satisfy all clauses in $\mathbf{C}$.
3. Given an instance x of I and a feasible solution τ of x, $m(x, \tau)$ =max(1, sum of weights associated to the variables that are **true** in τ).
4. opt= max.

We note that, in the case of this new problem, the feasible solutions are restricted to those truth assignments that satisfy the formula completely. Formally, in the definition, the measure is found by determining a maximum between 1 and the sum of the weights associated to the variables that are **true** in the truth assignment, because the formula could be not satisfiable. In this case we directly assume that the optimum value is 1, to define the optimization problem for every instance.

A first important result due to [9] and independently to [71] is the following:

Theorem 5.1 *MAX–VAR SAT is $\mathcal{NPO}$–complete.*

Proof. Let P be an $\mathcal{NPO}$ problem and let us consider the corresponding non deterministic Turing machine M associated to P and that was presented in Section 4.1.

According to Cook's Theorem, for any instance x, one can find a Boolean formula whose satisfying truth assignments are in one-to-one correspondence with the halting computation paths of $M(x)$. Let $y_1, y_2, \ldots, y_r$ be the Boolean variables describing the feasible solution y of x and let $m_1, \ldots, m_s$ be the Boolean variables that correspond to the tape cells on which M prints the value $m_P(x, y)$. Then a zero weight is assigned to every variable except the m_i's which are given weight 2^{s-i}.

Given a satisfying truth assignment, one is able to find a solution for P just by looking at the values of the y_i's. From the construction $m_P(x, y)$ is equal to the sum of the weights of the **true** variables. Therefore it has been proved that $P \leq_{\mathcal{PTAS}}$ *MAX–VAR SAT* with, in this case $c(\epsilon) = \epsilon$.
■

Considering a particular version of *MAX–VAR SAT* one can exhibit an example of a problem which is $\mathcal{APX}$–complete. Let us consider the problem *MAX–VAR BOUNDED SAT* in which the total sum of the weights is between Z and $2Z$, where Z is an integer given in input. Consequently the measure is changed in the following way: $\max(Z, \sum_{i=1}^{n} w_i \tau(u_i))$ if the formula is satisfied, $m(x, \tau) = Z$ otherwise

Theorem 5.2 *MAX–VAR BOUNDED SAT is $\mathcal{APX}$–complete.*

The proof of the theorem can be found in [27].

Historically *MAX–VAR BOUNDED SAT* is the first example of a problem that is $\mathcal{APX}$-complete. However, by combining together different techniques (including *PCP*) it would be possible to prove the following:

Theorem 5.3 *MAX W–SAT is $\mathcal{APX}$–complete.*

A presentation of the proof can be found in [8]. Let us note that this theorem is another way of stating that *MAX W–SAT* does not belong to $\mathcal{PTAS}$.

6 Local search

According to [73] "local search is based on what is perhaps the oldest optimization method – trial and error." The idea is simple and natural and it is surprising to see how successful local search has been on a variety of difficult problems. *MAX–SAT* is among the problems for which local search has been very effective: different variations of local search with randomness techniques have been proposed for *SAT* and *MAX–SAT* starting from the

late eighties, see for example [42, 81], motivated by previous applications of "min-conflicts" heuristics in the area of Artificial Intelligence [66].

The general scheme is based on generating a starting point in the set of admissible solution and trying to improve it through the application of simple *basic* moves. If a move ("trial") is successful one accepts it, otherwise ("error") one keeps the current point. Of course, the successfulness of a local search technique depends on the neighborhood chosen and there are often trade-offs between the size of the neighborhood (and the related computational requirements to calculate it) and the quality of the obtained local optima.

In addition, as it will be demonstrated in Sec. 6.2, the use of a guiding function different from the original one can in some cases guarantee local optima of better quality.

Because this presentation is dedicated to the *MAX–SAT* problem, the search space that we consider is given by all possible truth assignments. Of course, a truth assignment can be represented by a binary string. For this presentation, let us consider the elementary changes to the current assignment obtained by changing a single truth value. The definitions are as follows.

Let $\mathcal{U}$ be the discrete search space: $\mathcal{U} = \{0,1\}^n$, and let $f : \mathcal{U} \longrightarrow R$ (R are the real numbers) be the function to be maximized, i.e., in our case, the number of satisfied clauses. In addition, let $U^{(t)} \in \mathcal{U}$ be the current configuration along the *search trajectory* at iteration t, and $N(U^{(t)})$ the neighborhood of point $U^{(t)}$, obtained by applying a set of basic moves μ_i $(1 \leq i \leq n)$, where μ_i complements the i-th bit u_i of the string: $\mu_i\ (u_1, u_2, ..., u_i, ..., u_n) = (u_1, u_2, ..., 1 - u_i, ..., u_n)$. Clearly, these moves are idempotent ($\mu_i^{-1} = \mu_i$).

$$N(U^{(t)}) = \{U \in \mathcal{U} \ \text{ such that } \ U = \mu_i\, U^{(t)}, i = 1, ..., n\}$$

The version of *local search* (LS) that we consider starts from a random initial configuration $U^{(0)} \in \mathcal{U}$ and generates a search trajectory as follows:

$$V = \textsc{Best-Neighbor}(\ N(U^{(t)})\ ) \tag{10}$$

$$U^{(t+1)} = \begin{cases} V & \textbf{if} \quad f(V) > f(U^{(t)}) \\ U^{(t)} & \textbf{if} \quad f(V) \leq f(U^{(t)}) \end{cases} \tag{11}$$

where Best-Neighbor selects $V \in N(U^{(t)})$ with the best f value and ties are broken randomly. V in turn becomes the new current configuration if f improves. Other versions are satisfied with an improving (or non-worsening) neighbor, not necessarily the best one. Clearly, local search stops as soon as

the first local optimum point is encountered, when no improving moves are available, see eqn. 11. Let us define as LS^+ a modification of LS where a specified number of iterations are executed and the candidate move obtained by BEST-NEIGHBOR is *always* accepted even if the f value remains equal or worsens.

6.1 Quality of local optima

Let m^* be the optimum value and k the minimum number of literals contained in the problem clauses.

For the following discussion it is useful to consider the different degree of *coverage* of the various clause for a given assignment. Precisely, let us define as COV_s the subset of clauses that have exactly s literals matched by the current assignment, and by $\text{COV}_s(l)$ the number of clauses in COV_s that contain literal l.

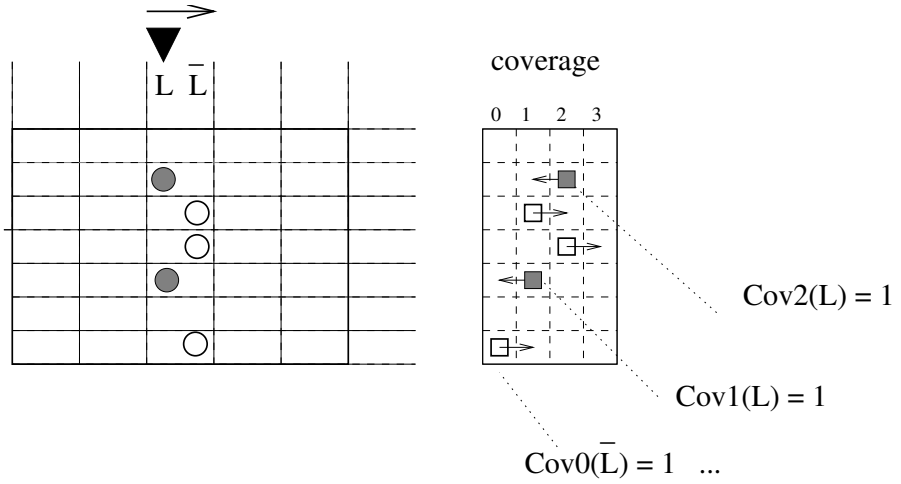

Figure 14: Literal L is changed from **true** to **false**.

One has the following theorem [48]:

Theorem 6.1 *Let m_{loc} be the number of satisfied clauses at a local optimum of any instance of MAX–SAT with at least k literals per clause. m_{loc} satisfies the following bound*

$$m_{loc} \geq \frac{k}{k+1} m$$

and the bound is sharp.

Proof. By definition, if the assignment U is a local optimum, one cannot flip the truth value of a variable (from **true** to **false** or *vice versa*) and obtain a net increase in the number of satisfied clauses f. Now, let $(\Delta f)_i$ by the increase in f if variable u_i is flipped. By using the above introduced quantities one verifies that:

$$(\Delta f)_i = -\textsc{Cov}_1(u_i) + \textsc{Cov}_0(\overline{u_i}) \leq 0 \tag{12}$$

In fact, when u_i is flipped one looses the clauses that contain u_i as the single matched literal, i.e., $\textsc{Cov}_1(u_i)$ and gains the clauses that have no matched literal and that contain $\overline{u_i}$, i.e., $\textsc{Cov}_0(\overline{u_i})$.

After summing over all variables:

$$\sum_{i=1}^{n} \textsc{Cov}_0(\overline{u_i}) \leq \sum_{i=1}^{n} \textsc{Cov}_1(u_i) \tag{13}$$

$$k|\textsc{Cov}_0| \leq |\textsc{Cov}_1| \leq m_{loc} \tag{14}$$

where the equality $\sum_{i=1}^{n} \textsc{Cov}_0(\overline{u_i}) = k|\textsc{Cov}_0|$ and $\sum_{i=1}^{n} \textsc{Cov}_1(u_i) = |\textsc{Cov}_1|$ have been used. The equality are demonstrated by counting how many times a clause in $\textsc{Cov}_0$ (or $\textsc{Cov}_1$) is uncountered during the sum. For example, because all literals are unmatched for the clauses in $\textsc{Cov}_0$, each of them will be encountered k times during the sum.

The conclusion is immediate:

$$m = m_{loc} + |\textsc{Cov}_0| \leq (1 + \frac{1}{k}) m_{loc} = \frac{k+1}{k} m_{loc} \tag{15}$$

■

The intuitive explanation is as follows: if there are too many clauses in $\textsc{Cov}_0$, because each of them has k unmatched literals, there will be at least one variable whose flipping will satisfy so many of these clauses to lead to a net increase in the number of satisfied clauses.

There is therefore a very simple local search algorithm that reaches the same bound as the GreedyJohnson1 algorithm. One starts from a truth assignments and keeps flipping variables that cause a net increase of satisfied clauses, until a local optimum is encountered. Of course, because one gains at least one clause at each step, there is an upper bound of m on the total number of steps executed before reaching the local optimum.

The following corollary is immediate:

Corollary 6.2 *If m_{loc} is the number of satisfied clauses at a local optimum, then:*

$$m_{loc} \geq \frac{k}{k+1} m^* \tag{16}$$

Besides *MAX–SAT* , many important optimization problems share the property that the ratio between the value of the local optimum and the optimal value is bounded by a constant. It is possible to define a class $\mathcal{GLO}$ composed of these problems. It is of interest to note that the closure of $\mathcal{GLO}$ coincides with $\mathcal{APX}$ [10].

6.2 Non-oblivious local optima

In the design of efficient approximation algorithms for *MAX–SAT* a recent approach of interest is based on the use of *non-oblivious functions* independently introduced in [2] and in [59].

Let us consider the classical local search algorithm LS for *MAX–SAT* , here redefined as *oblivious* local search (LS-OB). Clearly, the feasible solution found by LS-OB typically is only a *local* and not a *global* optimum.

Now, a different type of local search can be obtained by using a *different* objective function to direct the search, i.e., to select the best neighbor at each iteration. Local optima of the standard objective function f are not necessarily local optima of the different objective function. In this event, the second function causes an *escape* from a given local optimum. Interestingly enough, suitable *non-oblivious* functions f_{NOB} improve the performance of LS if one considers both the worst-case performance ratio and, as it has been shown in [13], the actual average results obtained on benchmark instances.

Let us mention a theoretical result for *MAX 2–SAT* . The d-neighborhood of a given truth assignment is defined as the set of all assignment where the values of at most d variables are changed. The theoretically-derived non–oblivious function for *MAX 2–SAT* is:

$$f_{NOB}(U) = \frac{3}{2}|\textsc{Cov}_1| + 2|\textsc{Cov}_2|$$

Theorems 7-8 of [59] state that:

Theorem 6.3 *The performance ratio for any oblivious local search algorithm with a d-neighborhood for MAX 2–SAT is* $2/3$ *for any* $d = o(n)$. *Non-oblivious local search with an 1-neighborhood achieves a performance ratio* $3/4$ *for MAX 2–SAT .*

Proof. While one is referred to the cited papers for the complete details, let us only demonstrate the second part of the theorem. The proof is a generalization of that for Theorem 6.1. Let the non-oblivious function be a weighted linear combination of the number of clauses with one and two matched literals:

$$f_{NOB} = a|\textsc{Cov}_1| + b|\textsc{Cov}_2|$$

Let $(\Delta f)_i$ by the increase in f if variable u_i is flipped. By using the definition of local optimum and the quantities introduced in Sec. 6.1 one has that $(\Delta f)_i \leq 0$ for each possible flip of a variable u_i. After expressing $(\Delta f)_i$ by using the above introduced quantities, one obtains:

$$-a|\textsc{Cov}_1(u_i)|-(b-a)|\textsc{Cov}_2(u_i)|+a|\textsc{Cov}_0(\overline{u_i})|+(b-a)|\textsc{Cov}_1(\overline{u_i})| \leq 0 \quad (17)$$

In fact, when u_i is flipped, all clauses that contain it decrease their coverage by one, while the clauses that contain $\overline{u_i}$ increase it by one, see also Fig. 14. As usual, let us assume that no clause contains both a literal and its negation.

After summing over all variables and collecting the sizes of the sets $\textsc{Cov}_i$ one obtains:

$$\sum_{i=1}^{n} (\Delta f)_i \leq 0 \quad (18)$$

$$\frac{b-a}{a}|\textsc{Cov}_2| + \frac{2a-b}{2a}|\textsc{Cov}_1| \geq |\textsc{Cov}_0| \quad (19)$$

Now one can fix the relative size of the values a and b in order to get the best possible bound. This occurs when the coefficients of the terms $|\textsc{Cov}_2|$ and $|\textsc{Cov}_1|$ in equation 19 are equal, that is, for $b = \frac{4}{3}a$.

For these values one obtains the following bound:

$$|\textsc{Cov}_2| + |\textsc{Cov}_1| \geq 3\,|\textsc{Cov}_0| \quad (20)$$

The number of satisfied clauses must be larger than three times the number of unsatisfied ones, which implies that $|\textsc{Cov}_0| \leq \frac{1}{4}m$, or $m_{loc} \geq \frac{3}{4}m$.

■

Therefore LS-NOB, by using a function that weights in different ways the satisfied clauses according to the number of matched literals, improves considerably the performance ratio, even if the search is restricted to a much smaller neighborhood. In particular the "standard" neighborhood where all possible flips are tried is sufficient.

With a suitable generalization the above result can be extended: LS-NOB achieves a performance ratio $1 - \frac{1}{2^k}$ for *MAX–k–SAT* . The oblivious function for *MAX–k–SAT* is of the form:

$$f_{NOB}(U) = \sum_{i=1}^{k} c_i |\mathrm{Cov}_i|$$

and the above given performance ratio is obtained if the quantities $\Delta_i = c_{i+1} - c_i$ satisfy:

$$\Delta_i = \frac{1}{(k-i+1)\begin{pmatrix} k \\ i-1 \end{pmatrix}} \left[\sum_{j=0}^{k-i} \begin{pmatrix} k \\ j \end{pmatrix} \right]$$

Because the positive factors c_i that multiply $|\mathrm{Cov}_i|$ in the function f_{NOB} are strictly increasing with i, the approximations obtained through f_{NOB} tend to be characterized by a "redundant" satisfaction of many clauses. Better approximations, at the price of a limited number of additional iterations, can be obtained by a two-phase local search algorithm (NOB&OB): after a random start f_{NOB} guides the search until a local optimum is encountered [13]. As soon as this happens a second phase of LS is started where the move evaluation is based on f. A further reduction in the number of unsatisfied clauses can be obtained by a "plateau search" phase following NOB&OB: the search is continued for a certain number of iterations after the local optimum of OB is encountered, by using LS^+, with f as guiding function [13].

6.2.1 An example of non-oblivious search

Let us consider the following task with number of variables $n = 5$, and clauses $m = m^* = 4$, see also Fig. 15:

$$(\overline{u}_1 \vee \overline{u}_2 \vee u_3) \wedge (\overline{u}_1 \vee \overline{u}_2 \vee u_4) \wedge (\overline{u}_1 \vee \overline{u}_2 \vee u_5) \wedge (\overline{u}_3 \vee \overline{u}_4 \vee \overline{u}_5)$$

Let us assume that the assignment $U = (11111)$ is reached by OB local search. It is immediate to check that $U = (11111)$ is an oblivious local optimum with one unsatisfied clause (clause-4). While OB stops here, a possible sequence to reach the global optimum starting from U is the following: i) u_1 is set to **false**, ii) u_3 is set to **false**. Now, the first move does not change the number of satisfied clauses, but it changes the "amount of redundancy"

(in clause-1 two literals are now satisfied, i.e., clause-1 enters Cov_2) and the move *is* a possible choice for a selection based on the *non-oblivious* function. The *oblivious* plateau has been eliminated and the search can continue toward the globally optimal point $U = (01011)$.

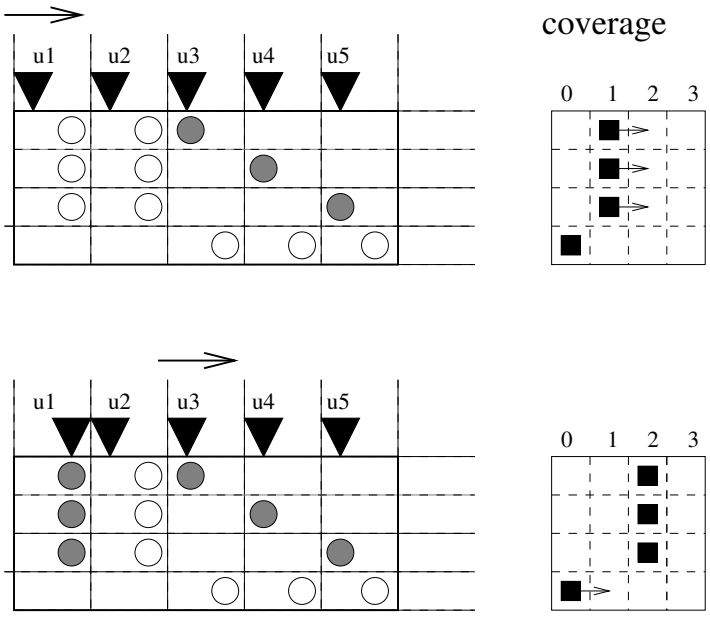

Figure 15: Non-oblivious search takes the different coverage into account.

6.3 Local search satisfies most *3–SAT* formulae

An intriguing result by Koutsoupias and Papadimitriou [63] shows that, for the vast majority of satisfiable *3–SAT* formulae, the local search heuristic that starts at a random truth assignments and repeatedly flips a variable that improves the number of satisfied clauses, almost always succeeds in discovering a satisfying truth assignment.

Let us consider all clauses that are satisfied by a given truth assignment $\hat{U}$ and let us pick each of them with probability $p = 1/2$ to build a *3–SAT* formula. The following theorem [63] is demonstrated:

Theorem 6.4 *Let* $0 < \varepsilon < 1/2$. *Then there exists* c, $c \approx \left(1 - \sqrt{1 - (1/2 - \varepsilon)^2}\right)^2 /6$, *such that for all but a fraction of at most* $n2^n e^{-cn^2/2}$ *satisfiable 3–SAT formulae with* n *variables, the probability that*

local search succeeds in discovering a truth assignment in each independent trial from a random start is at least $1 - e^{-\varepsilon^2 n}$.

Proof. Let us focus on the structure of the proof, without giving the technical details. One assumes that there is an assignment $\hat{U}$ that satisfies all clauses and shows that, if one starts from a *good* initial assignment, i.e., one that agrees with $\hat{U}$ in at least $(1/2 - \varepsilon)$ variables, the probability that the local search is ever *mislead* is small. By "*mislead*" one means that, when a variable is flipped, the Hamming distance between $U^{(t)}$ and $\hat{U}$ increases. The Hamming distance between two binary strings is given by the number of differing bits.

In detail, the quantity $1 - e^{-\varepsilon^2 n}$ in the theorem is the probability that the initial random truth assignment is *good* (use Chernoff bound). Then one demonstrates that, if the initial assignment is good, the probability that one does not reduce the Hamming distance between $U^{(t)}$ and $\hat{U}$ when an improving neighbor is chosen is at most $2e^{-cpn^2}$, the probability being measured with respect to the random choice of the clauses to build the original formula ($p = 1/2$ for the above theorem).

Finally, the probability that local search starting from a good assignment will ever be misled by flipping a variable during the entire search trajectory is at most $n2^n e^{-cpn^2}$, since there are at most $n2^{n-1}$ such possible flippings – the number of edges of the n-hypercube.

■

The original formulation of the above theorem is for a *greedy* version of local search, using the function BEST-NEIGHBOR described in eqn. 10, but the authors note that greediness is not required for the theorem to hold, although it may be important in practice.

Let us finally note that the result, while of theoretical interest, is valid for formulae with many clauses (p must be such that the expected number of clauses is $\Omega(n^2)$), while the most difficult formulae have a number of clauses that is linear in n, see also Sec 9.2.

6.4 Randomized search for *2–SAT* (Markov processes)

A "natural" polynomial-time *randomized* search algorithm for *2–SAT* is presented in [72]. While it has long been known that *2–SAT* is a polynomially solvable problem, the algorithm is of interest because of its simplicity and is summarized here also because it motivated the GSAT-WITH-WALK algorithm of [78], see also Sec. 7.2.

In its "standard" form, local search is guided by the number of satisfied clauses and the basic criterion is that of accepting a neighbor only if more clauses are satisfied. The paper by Papadimitriou [72] changes the perspective by concentrating the attention to the *unsatisfied* clauses.

The algorithm for *2–SAT* , is extremely simple:

```
MarkovSearch
1     Start with any truth assignment
2     while   there are unsatisfied clauses  do
3           pick one of them and flip a random literal in it
```

Figure 16: The MarkovSearch randomized algorithm for *2–SAT* .

Let us note that worsening moves, leading to a lower number of satisfied clause, can be accepted during the search.

One can prove that:

Theorem 6.5 *The* MarkovSearch *randomized algorithm for 2–SAT , if the instance is satisfiable, finds a satisfying assignment in* $O(n^2)$ *expected number of steps.*

Proof. The proof involves an aggregation of the states of the Markov chain so that the chain is mapped to the *gambler's ruin* chain. A sketch of the proof is derived from [68]. Given an instance with a satisfying assignment $\hat{U}$, and the current assignment $U^{(t)}$, the progress of the algorithm can be represented by a particle moving between the integers $\{0, 1, ..., n\}$ on the real line. The position of the particle indicates how many variables in $U^{(t)}$ agree with those of $\hat{U}$. At each iteration the particle's position can change only by one, from the current position i to $i + 1$ or $i - 1$ for $0 < i < n$. A particle at 0 can move only to 1, and the algorithm terminates when the particle reaches position n, although it may terminate at some other position with a satisfying assignment different from $\hat{U}$. The crucial fact is that, in an unsatisfied clause, at least one of the *two* literals has an incorrect value and therefore, with probability at least $1/2$, the number of correct variables increases by one when a randomized step is executed.

The random walk on the line is one of the most extensively studied stochastic processes. In particular, the above process is a version of the "gambler's ruin" chain with reflecting barrier (that is, the house cannot lose its last dollar). Average number of steps for the gambler to be ruined is $O(n^2)$.

■

7 Memory-less Local Search Heuristics

State-of-the-art heuristics for *MAX–SAT* are obtained by complementing local search with schemes that are capable of producing better approximations beyond the locally optimal points. In some cases, these schemes generate a sequence of points in the set of admissible solutions in a way that is fixed before the search starts. An example is given by *multiple runs* of local search starting from different random points. The algorithm does not take into account the *history* of the previous phase of the search when the next points are generated. The term *memory-less* denotes this lack of feedback from the search history.

In addition to the cited *multiple-run* local search, these techniques are based on Markov processes (Simulated Annealing), see Sec. 7.1, "plateau" search and "random noise" strategies, see Sec. 7.2, or combinations of randomized constructions and local search, see Sec. 7.3.

7.1 Simulated Annealing

The use of a Markov process (Simulated Annealing or SA for short) to generate a stochastic search trajectory is adopted for example in [82].

```
SA
1   for tries ← 1 to MAX-TRIES
2       U ← random truth assignment ; iter ← 0
3       forever
4           if U satisfies all clause then return U
5           temperature ← MAX-TEMP × e^{−iter×decay_rate}
6           if temperature < MIN-TEMP then exit loop
7           for i ← 1 to n
8               δ ← increase of satisfied clauses if u_i is flipped
9               FLIP(u_i)with probability1/(1 + e^{−δ/temperature})
10          iter ← iter + 1
```

Figure 17: The Simulated Annealing algorithm for *SAT* .

The main structure of the algorithm is illustrated in Fig. 17, adapted from [82]. For a certain number of tries, a random truth assignment is generated (line 2) and the *temperature* parameter is set to MAX-TEMP. In the inner loop, new assignments are generated by probabilistically flipping each variable based on the improvement δ in the number of satisfied clauses that would occur after the flip. Of course, the improvement can be negative.

The probability to flip is given by a logistic function that penalizes smaller or negative improvements (line 9). The inner loop controls the *annealing schedule*: when *iter* increases the *temperature* slowly decreases (line 5) until a minimum of MIN-TEMP is reached and the control exits the loop (line 6) Let us note that, when the *temperature* is large, the moves are similar to those produced by a random walk, while, when the *temperature* is low the acceptance criterion of the moves is that of local search and the algorithm resembles GSAT, that will be introduced in Sec. 7.2. Implementation details, the addition of a "random walk" modification inspired by [78], and experimental results are described in the cited paper.

7.2 GSAT with "random noise" strategies

SAT is of special concern to Artificial Intelligence because of its connection to *reasoning*. In particular, deductive reasoning is the complement of satisfiability: from a collection of base facts A one should deduce a sentence F if and only if $A \cup \overline{F}$ is *not* satisfiable, see also Sec. 2.2. The popular and effective algorithm GSAT was proposed in [81] as a *model-finding* procedure, i.e., to find an interpretation of the variables under which the formula comes out **true**. GSAT consists of multiple runs of LS^+, each run consisting of a number of iterations that is typically proportional to the problem dimension n. The experiments in [81] show that GSAT can be used to solve hard (see sec. 9.2) randomly generated problems that are an order of magnitude larger than those that can be solved by more traditional approaches like Davis-Putnam or resolution. Of course, GSAT is an incomplete procedure: it could fail to find an optimal assignment. An extensive empirical analysis of GSAT is presented in [37, 36].

Different "noise" strategies to escape from attraction basins are added to GSAT in [78, 80]. In particular, the GSAT-WITH-WALK algorithm has been tested in [80] on the Hansen-Jaumard benchmark of [48], where a better performance with respect to SAMD is demonstrated, although requiring much longer CPU times. See Sec. 8.1 for the definition of SAMD.

The algorithm is briefly summarized in Fig. 18. A certain number of tries (MAX-TRIES) is executed, where each try consists of a number of iterations (MAX-FLIPS). At each iteration a variable is chosen by two possible criteria and then flipped by the function FLIP, i.e., U_i becomes equal to $(1 - U_i)$. One criterion, active with "noise" probability p, selects a variable occurring in some unsatisfied clause with uniform probability over these variables, the other one is the standard method based on the function f given by the number of satisfied clauses. The first criterion was motivated

```
GSAT-WITH-WALK
1     for i ← 1 to MAX-TRIES
2         U ← random truth assignment
3         for j ← 1 to MAX-FLIPS
4             if RANDOMNUMBER < p then
5                 u ← any variable occurring in some unsat. clause
6             else
7                 u ← any variable with largestΔf
8             FLIP(u)
```

Figure 18: The GSAT-WITH-WALK algorithm. RANDOMNUMBER generates random numbers in the range $[0, 1]$.

by [72], see also Sec. 6.4. For a generic move μ, the quantity $\Delta_\mu f$ (or Δf for short) is defined as $f(\mu\ U^{(t)}) - f(U^{(t)})$. The straightforward book-keeping part of the algorithm is not shown. In particular, the best assignment found during all trials is saved and reported at the end of the run. In addition, the run is terminated immediately if an assignment is found that satisfies all clauses. The original GSAT algorithm can be obtained by setting $p = 0$ in the GSAT-WITH-WALK algorithm of Fig. 18.

7.3 Randomized Greedy and Local Search (GRASP)

A hybrid algorithm that combines a randomized greedy construction phase to generate initial candidate solutions, followed be a local improvement phase is the GRASP scheme proposed in [75] for the *SAT* and generalized for the *MAX W–SAT* problem in [76], a work that is briefly summarized in this section.

GRASP is an iterative process, with each iteration consisting of two phases, a construction phase and a local search phase.

During each construction, all possible choices are ordered in a candidate list with respect to a greedy function measuring the (myopic) benefit of selecting it. The algorithm is randomized because one picks in a random way one of the best candidates in the list, not necessarily the top candidate. In this way different solutions are obtained at the end of the construction phase.

Because these solutions are not guaranteed to be locally optimal with respect to simple neighborhoods, it is usually beneficial to apply a local search to attempt to improve each constructed solution.

A high-level description of the GRASP algorithm is presented in Fig. 19,

```
GRASP(RCLSize, MaxIter, RandomSeed)
1    △ Input instance and initialize data structures
2    for i ← 1 to MaxIter
3        [ U ← CONSTRUCTGREEDYRAND(RCLSize, RandomSeed)
4        [ U ← LOCALSEARCH(U)

CONSTRUCTGREEDYRAND(RCLSize, RandomSeed)
1    for k ← 1 to n
2        [ MAKERCL(RCLSize)
3        [ s ← SELECTINDEX(RandomSeed)
4        [ ASSIGNVARIABLE(s)
5        [ ADAPTGREEDYFUNCTION(s)
```

Figure 19: The GRASP algorithm (above) and the randomized greedy construction (below).

a summarized version of the more detailed description in [76]. After reading the instance and initializing the data structures one repeats for $MaxIter$ iterations the construction of an assignment U and the application of local search starting from U to produce a possibly better assignment (lines 2–4). Of course, the best assignment found during all iterations is saved and reported at the end. In addition to $MaxIter$, the parameters are $RCLSize$, the size of the restricted candidate list of moves out of which a random selection is executed, and a random seed used by the random number generator. In detail, see function CONSTRUCTGREEDYRAND in Fig. 19, the restricted candidate list of assignments is created by MAKERCL, the index of the next variable to be assigned a truth value is chosen by SELECTINDEX, the truth value is assigned by ASSIGNVARIABLE and the greedy function that guides the construction is changed by ADAPTGREEDYFUNCTION to reflect the assignment just made.

The remaining details about the greedy function (designed to maximize the total weight of yet-unsatisfied clauses that become satisfied after a given assignment), the creation of the restricted candidate list, and local search (based on the *1-flip* neighborhood) are presented in [76], together with experimental results.

8 History-sensitive Heuristics

Different history-sensitive heuristics have been proposed to continue local search schemes beyond local optimality. These schemes aim at intensifying the search in promising regions and at diversifying the search into uncharted territories by using the information collected from the previous phase (the *history*) of the search. The *history* at iteration t is formally defined as the set of ordered couples (U, s) such that $0 \leq s \leq t$ and $U = U^{(s)}$.

Because of the internal feedback mechanism, some algorithm parameters can be modified and tuned in an *on-line* manner, to reflect the characteristics of the *task* to be solved and the *local* properties of the configuration space in the neighborhood of the current point. This tuning has to be contrasted with the *off-line* tuning of an algorithm, where some parameters or choices are determined for a given problem in a preliminary phase and they remain fixed when the algorithm runs on a specific instance.

8.1 Prohibition-based Search: TS and SAMD

Tabu Search (TS) is a *history-sensitive* heuristic proposed by F. Glover [38] and, independently, by Hansen and Jaumard, that used the term SAMD ("steepest ascent mildest descent") and applied it to the *MAX–SAT* problem in [48]. The main mechanism by which the history influences the search in TS is that, at a given iteration, some neighbors are *prohibited*, only a non-empty subset $N_A(U^{(t)}) \subset N(U^{(t)})$ of them is *allowed*. The general way of generating the search trajectory that we consider is given by:

$$N_A(U^{(t)}) = \textsc{Allow}(N(U^{(t)}), U^{(0)}, ..., U^{(t)}) \quad (21)$$

$$U^{(t+1)} = \textsc{Best-Neighbor}(\ N_A(U^{(t)})\ ) \quad (22)$$

The set-valued function ALLOW selects a non-empty subset of $N(U^{(t)})$ in a manner that depends on the entire previous history of the search $U^{(0)}, ..., U^{(t)}$. Let us note that worsening moves *can* be produced by eqn. 22, as it must be in order to exit local optima.

The introduction of algorithm SAMD is motivated in [48] by contrasting the technique with Simulated Annealing (SA) [60] for maximization. The directions of local changes are little explored by SA: for example, if the objective function increases, the change is always accepted however small it may be. On the contrary, it is desirable to exploit the information on the direction of *steepest* ascent and yet to retain the property of not being blocked at the first local optimum found. SAMD performs local changes in the direction of steepest ascent until a local optimum is encountered, then a

local change along the direction of mildest descent takes place and the reverse move is *forbidden* for a given number of iterations to avoid cycling with a high probability. The details of the SAMD technique as well as additional specialized devices for detecting and breaking cycles are outlined in [48]. A computational comparison with SA and with Johnson's two algorithms is also presented. A specialized Tabu Search heuristic is used in [51] to speed up the search for a solution (if the problem is satisfiable) as part of a branch-and-bound algorithm for SAT , that adopts both a relaxation and a decomposition scheme by using polynomial instances, i.e., 2–SAT and Horn SAT .

8.2 HSAT and "clause weighting"

In addition to the already cited SAMD [48] heuristic that uses the temporary prohibitions of recently executed moves, let us mention two variations of GSAT that make use of the previous history.

HSAT [37] introduces a tie-breaking rule into GSAT: if more moves produce the same (best) Δf, the preferred move is the one that has not been applied for the longest span. HSAT can be seen as a "soft" version of Tabu Search: while TS prohibits recently-applied moves, HSAT discourages recent moves if the same Δf can be obtained with moves that have been "inactive" for a longer time. It is remarkable to see how this innocent variation of GSAT can increase its performance on some SAT benchmark tasks [37].

Clause–weighting has been proposed in [79] in order to increase the effectiveness of GSAT for problems characterized by strong asymmetries. In this algorithm a positive weight is associated to each clause to determine how often the clause should be counted when determining which variable to flip. The weights are dynamically modified during problem solving and the qualitative effect is that of "filling in" local optima while the search proceeds. Clause–weighting can be considered as a "reactive" technique where a repulsion from a given local optimum is generated in order to induce an escape from a given attraction basin.

8.3 Reactive Search

Different methods to generate prohibitions produce qualitatively different *search trajectories*, i.e., sequences of visited configurations $U^{(t)}$. In particular, prohibitions based on a list of *moves* lead to a faster escape from a locally optimal point than prohibitions based on a list of visited *configurations* [11]. In this method prohibitions are determined by the last moves

applied. In detail, the ALLOW function can be specified by introducing a *prohibition parameter* T (also called *list size*) that determines how long a move will remain prohibited after its execution. The FIXED-TS algorithm is obtained by fixing T throughout the search [38]. A neighbor is allowed if and only if it is obtained from the current point by applying a move that has not been used during the last T iterations. In detail, if LASTUSED(μ) is the last usage time of move μ (LASTUSED(μ) $= -\infty$ at the beginning):

$$N_A(U^{(t)}) = \{U = \mu\, U^{(t)} \text{ such that } \text{LASTUSED}(\mu) < (t - T)\} \quad (23)$$

The Reactive Tabu Search algorithm [14], REACTIVE-TS for short, defines simple rules to determine the prohibition parameter by reacting to the repetition of previously-visited configurations. One has a repetition if $U^{(t+R)} = U^{(t)}$, for $R \geq 1$. The prohibition period T depends on the iteration t (therefore the notation is $T^{(t)}$), and the discrete dynamical system that generates the search trajectory comprises an additional evolution equation for $T^{(t)}$, that is specified through the function REACT, see eqn. 24 below. The dynamical system becomes:

$$T^{(t)} = \text{REACT}(T^{(t-1)}, U^{(0)}, ..., U^{(t)}) \quad (24)$$

$$N_A(U^{(t)}) = \{U = \mu\, U^{(t)} \text{ such that } \text{LASTUSED}(\mu) < (t - T^{(t)})\} \quad (25)$$

$$U^{(t+1)} = \text{BEST-NEIGHBOR}(N_A(U^{(t)})) \quad (26)$$

While the reader is referred to [14] for the details, the design principles of REACTIVE-TS are that $T^{(t)}$ (in the range $1 \leq T^{(t)} \leq n - 2$) increases when repetitions happen, and decreases when repetitions disappear for a sufficiently long search period. For convenience, let us introduce a "fractional prohibition" T_f, such that the prohibition is obtained by setting $T = \lfloor T_f\, n \rfloor$. T_f ranges between zero and one, with bounds inherited from those on T. Larger T values imply larger diversification, in particular the relationship between T and the diversification is as follows:

- The Hamming distance H between a starting point and successive point along the trajectory is strictly increasing for $T + 1$ steps.

$$H(U^{(t+\tau)}, U^{(t)}) = \tau \quad \text{for} \quad \tau \leq T + 1$$

- The minimum repetition interval R along the trajectory is $2(T + 1)$.

$$U^{(t+R)} = U^{(t)} \;\Rightarrow\; R \geq 2(T + 1)$$

REACTIVE-TS has been applied to various problems with competitive performance with respect to alternative heuristics like FIXED-TS, SA, Neural Networks, and Genetic Algorithms, see the review in [11].

8.3.1 The Hamming-Reactive Tabu Search (H-RTS) algorithm

An algorithm that combines the previously described techniques of local search (oblivious and non-oblivious), the use of prohibitions (see TS and SAMD), and a reactive scheme to determine the prohibition parameter is presented in [12]. The algorithm is called HAMMING-REACTIVE-TS algorithm, and its core is illustrated in Fig. 20.

```
HAMMING-REACTIVE-TS
1     repeat
2         ⌈ t_r ← t
3         | U ← random truth assignment
4         | T ← ⌊T_f n⌋
          |
5         | repeat                                  { NOB local search }
6         |     [ U ← BEST-MOVE(LS, f_NOB)
7         | until largest Δf_NOB = 0
          |
8         | repeat
9         |     ⌈ repeat                            { local search }
10        |     |     [ U ← BEST-MOVE(LS, f_OB)
11        |     | until largest Δf_OB = 0
12        |     | U_I ← U
          |     |
13        |     | for 2(T + 1) iterations           { reactive tabu search }
14        |     |     [ U ← BEST-MOVE(TS, f_OB)
15        |     | U_F ← U
          |     |
16        |     ⌊ T ← REACT(T_f, U_F, U_I)
17        ⌊ until (t − t_r) > 10 n
18    until solution is acceptable or max. number of iterations reached
```

Figure 20: The *H-RTS* algorithm.

The initial truth assignment is generated in a random way, and non-oblivious local search (LS-NOB) is applied until the first local optimum of f_{NOB} is encountered. LS-NOB obtains local minima of better average quality than LS-OB, but then the guiding function becomes the standard oblivious one. This choice was motivated by the success of the NOB & OB combination [13] and by the poor diversification properties of NOB alone, see [12].

The search proceeds by repeating phases of local search followed by phases of TS (lines 8–17 in Fig. 20), until a suitable number of iterations are accumulated after starting from the random initial truth assignment (see line 17 in Fig. 20). A single elementary move is applied at each iteration. The variable t, initialized to zero, identifies the current iteration and increases after a local move is applied, while t_r identifies the iteration when the last random assignment was generated. Some trivial bookkeeping details (like the increase of t) are not shown in the figure.

During each combined phase, first the local optimum of f is reached, then $2(T + 1)$ moves of Tabu Search are executed. The design principle underlying this choice is that prohibitions are necessary for diversifying the search only after LS reaches a local optimum. The fractional prohibition T_f is changed during the run by the function REACT to obtain a proper balance of diversification and bias [12].

The random restart executed after 10 n moves guarantees that the search trajectory is not confined in a localized portion of the search space.

Being an heuristic algorithm, there is not a natural termination criterion. In its practical application, the algorithm is therefore run until either the solution is acceptable, or a maximum number of moves (and therefore CPU time) has elapsed. What is demonstrated in the computational experiments in [12] is that, given a fixed number of iterations, HAMMING-REACTIVE-TS achieves much better average results with respect to competitive algorithms (GSAT and GSAT-WITH-WALK). Because, to a good approximation, the actual running time is proportional to the number of iterations, HAMMING-REACTIVE-TS should therefore be used to obtain better approximations in a given allotted number of iterations, or equivalent approximations in a much smaller number of iterations.

9 Experimental analysis and threshold effects

Given the hardness of the problem and the relevancy for applications in different fields, the emphasis on the experimental analysis of algorithms for the *MAX–SAT* problem has been growing in recent years.

In some cases the experimental comparisons have been executed in the framework of "challenges," with support of electronic collection and distribution of software, problem generators and test instances. An example is the the Second DIMACS Algorithm Implementation Challenge on Cliques, Coloring and Satisfiability, whose results have been published in [53]. The archive is currently available from:

`http://dimacs.rutgers.edu/Challenges/`.
Practical and industrial *MAX–SAT* problems and benchmarks, with significant case studies are also presented in [30], see also the contained review [45].

9.1 Models

Let us describe some basic problem models that are considered both in theoretical and in experimental studies of *MAX–SAT* algorithms [45].

- **k-SAT model**, also called **fixed length clause model**. A randomly generated CNF formula consists of independently generated random clauses, where each clause contains exactly k literals. Each literal is chosen uniformly from $U = \{u_1, ..., u_n\}$ without replacement, and negated with probability p. The default value for p is $1/2$.

- **average k-SAT model**, also called **random clause model**. A randomly generated CNF formula consists of independently generated random clauses. Each literal has ha probability p of being part of a clause. In detail, each of the n variables occurs positively with probability $p(1-p)$, negatively with probability $p(1-p)$, both positively and negatively with probability p^2, and is absent with probability $(1-p)^2$.

Both models have many variations depending on whether the clauses are required to be different, whether a variable and its negation can be present in the same clause, etc.

Although superficially similar, the two models differ in the *difficulty* to solve the obtained formulae and in the mathematical analysis. In particular, when the initial formula comes from the average k-SAT model, a step that fixes the value of a variable produces a set of clauses from the same model, while if the same step is executed in the k-SAT model, the resulting clauses do not necessarily have the same length and therefore do not come from the k-SAT model.

Other *structured* problem models are derived from the mapping of instances of different problems, like *coloring*, n-queens, etc. [43]. The performance of algorithms on these more structured models tend to have little correlation with the performance tested on the above introduced random problems. Unfortunately, the theoretical analysis of these more structured problems is very hard. The situation worsens if one considers "real-world" practical applications, where one is typically confronted with a few instances and little can be derived about the "average" performance, both because the

probability distribution is not known and because the number of instances tends to be very small.

A compromise can be reached by having parametrized generators that capture some of the relevant structure of the "real-world" problems of interest.

9.2 Hardness and threshold effects

Different algorithms demonstrate a different degree of effort, measured by number of elementary steps or CPU time, when solving different kinds of instances. For example, Mitchell et al. [67] found that some distributions used in past experiments are of little interest because the generated formulae are almost always very easy to satisfy. They also reported that one can generate very hard instances of k-SAT, for $k \geq 3$. In addition, they report the following observed behavior for random fixed length *3–SAT* formulae: if r is the ratio r of clauses to variables ($r = m/n$), almost all formulae are satisfiable if $r < 4$, almost all formulae are unsatisfiable if $r > 4.5$. A rapid transition seems to appear for $r \approx 4.2$, the same point where the computational complexity for solving the generated instances is maximized, see [61, 26] for reviews of experimental results.

A series of theoretical analyses aim at approximating the unsatisfiability threshold of random formulae. Let us define the notation and summarize some results obtained.

Let $\mathbf{C}$ be a random k-SAT formula. The research problem that has been considered, see for example [62], is to compute the least real number κ such that, if r is larger than κ, then the probability of $\mathbf{C}$ being satisfiable converges to 0 as n tends to infinity. In this case one says that $\mathbf{C}$ is asymptotically almost certainly satisfiable. Experimentally, κ is a threshold value marking a "sudden" change from probabilistically certain satisfiability to probabilistically certain unsatisfiability. More precisely [1], given a sequence of events $\mathcal{E}_i$, one says that $\mathcal{E}_n$ occurs almost surely (a.s.) if $\lim_{n\to\infty} \mathbf{Pr}[\mathcal{E}_n] = 1$, where $\mathbf{Pr}[event]$ denotes the probability of an event. The behavior observed in experiments with random k-SAT leads to the following conjecture:
For every $k \geq 2$, there exist r_k such that for any $\varepsilon > 0$, random instances of k-SAT with $(r_k - \varepsilon)n$ clauses are a.s. satisfiable and random instances with $(r_k + \varepsilon)n$ clauses are a.s. unsatisfiable

For $k = 2$ (i.e., for the polynomially solvable *2–SAT*) the conjecture was proved [23, 41], in fact showing that $r_2 = 1$. For $k = 3$ much less progress has been made: neither the existence of r_3 nor its value has been

determined.

In the fixed-length *3–SAT* model, the total number of all possible clauses is $8\binom{n}{3}$ and the probability that a random clause is satisfied by a truth assignment U is 7/8.

Let $\mathcal{U}_n$ be the set of all truth assignments on n variables, and let $\mathcal{S}_n$ be the set of assignments that satisfy the random formula $\mathbf{C}$. Therefore the cardinality $|\mathcal{S}_n|$ is a random variable. Given $\mathbf{C}$, let $|\mathcal{S}_n(\mathbf{C})|$ be the number of assignments satisfying $\mathbf{C}$.

The expected value of the number of satisfying truth assignments of a random formula, $\mathbf{E}[|\mathcal{S}_n|]$, is defined as:

$$\mathbf{E}[|\mathcal{S}_n|] = \sum_{\mathbf{C}} (\mathbf{Pr}[\mathbf{C}]\,|\mathcal{S}_n(\mathbf{C})|) \tag{27}$$

The probability that a random formula is satisfiable is:

$$\mathbf{Pr}[\text{the random formula is satisfiable}] = \sum_{\mathbf{C}} (\mathbf{Pr}[\mathbf{C}]\, I_{\mathbf{C}}) \tag{28}$$

where $I_{\mathbf{C}}$ is 1 if $\mathbf{C}$ is satisfiable, 0 otherwise.

From equations (27) and (28) the following Markov's inequality follows:

$$\mathbf{Pr}[\text{the random formula is satisfiable}] \leq \mathbf{E}[|\mathcal{S}_n|] \tag{29}$$

Let us now consider the "first moment" argument to obtain an upper bound for κ in the *3–SAT* model. First one observes that the expected number of truth assignments that satisfy $\mathbf{C}$ is $2^n(7/8)^{rn}$, then one lets this expected value converge to zero and uses the above Markov's inequality. From this one obtains

$$\kappa \leq \log_{8/7} 2 = 5.191$$

This result has been found independently by many people, including [33] and [24].

The weakness of the above technique is that, in the right-hand side of equation (27) one can have small probabilities multiplied by large cardinalities, therefore the condition may be unnecessarily strong to ensure only that $\mathbf{C}$ is almost certainly satisfiable. In [62], instead of considering the random class $\mathcal{S}_n$ that may have a large cardinality for a formula with small probability, one considers a *subset* of it obtained by considering truth assignments that satisfy a local maximality condition. In particular, one considers the subset $\mathcal{S}_n^{\#}$ defined as the random class of assignments U satisfying $\mathbf{C}$ such

that any assignment obtained from U by changing exactly one **false** value of U to **true** does not satisfy **C**.

It is demonstrated in the cited paper that the expected value $\mathbf{E}\left[\mathcal{S}_n^{\#}\right]$ is at most $(7/8)^{rn}(2 - e^{-3r/7} + o(1))^n$. It follows that the unique positive solution of the equation

$$(7/8)^{rn}(2 - e^{-3r/7})^n = 1$$

is an upper bound for κ. This solution is less than 4.667. Better bounds can be obtained by increasing the range of locality when selecting the local maxima that represent $\mathcal{S}_n$. A previous best bound of 4.758 had been obtained in [55] by non-elementary means. Independently, Dubois and Boufkhad [31] obtained an upper bound of 4.64.

Unlike upper bounds, which are based on probabilistic counting arguments, all known lower bounds for r_3 are algorithmic. The UNIT CLAUSE algorithm for *3–SAT* is considered in [20], where it is shown that, for $r < 2.9$ or $r < 8/3$, depending on the presence or absence of a "majority rule," it finds a satisfying assignment with positive probability instead of a.s. (therefore this does not imply that $r_3 \geq 2.9$). The PURE LITERAL algorithm succeeds a.s. for $r < 1.63$, see [17]. A generalization of the analysis [34] shows that the GUC algorithm succeeds a.s. for $r < 3.003$, giving the best known lower bound for r_3. Additional recent developments are presented in [1].

References

[1] D. Achlioptas, L. M. Kirousis, E. Kranakis, and D. Krinzac, *Rigorous results for random (2+p)-SAT*, Proc. Work. on Randomized Algorithms in Sequential, Parallel and Distributed Computing, Santorini, Greece, 1997.

[2] P. Alimonti, *New local search approximation techniques for maximum generalized satisfiability problems*, Proc. Second Italian Conf. on Algorithms and Complexity, Rome, 1994, pp. 40–53.

[3] S. Arora, C. Lund, R. Motwani, M. Sudan, and M. Szegedy, *Proof verification and hardness of approximation problems*, Proc. 33rd Annual IEEE Symp. on Foundations of Computer Science, IEEE Computer Society, 1992, pp. 14–23.

[4] S. Arora and S. Safra, *Probabilistic checking of proofs: a new characterization of NP*, Proc. 33rd Annual IEEE Symp. on Foundations of Computer Science, IEEE Computer Society, 1992, pp. 2–13.

[5] T. Asano, *Approximation algorithms for MAX-SAT: Yannakakis vs. Goemans-Williamson*, Proc. 3rd Israel Symp. on the Theory of Computing and Systems, Ramat Gan, Israel, 1997, pp. 24–37.

[6] T. Asano, T. Ono, and T. Hirata, *Approximation algorithms for the maximum satisfiability problem*, Proc. 5th Scandinavian Work. on Algorithms Theory, 1996, pp. 110–111.

[7] P. Asirelli, M. de Santis, and A. Martelli, *Integrity constraints in logic databases*, Journal of Logic Programming **3** (1985), 221–232.

[8] G. Ausiello, P. Crescenzi, and M. Protasi, *Approximate solution of NP optimization problems*, Theoretical Computer Science **150** (1995), 1–55.

[9] G. Ausiello, A. D'Atri, and M. Protasi, *Lattice theoretic properties of NP-complete problems*, Fundamenta Informaticae **4** (1981), 83–94.

[10] G. Ausiello and M. Protasi, *Local search, reducibility and approximability of NP-optimization problems*, Information Processing Letters **54** (1995), 73–79.

[11] R. Battiti, *Reactive search: Toward self-tuning heuristics*, Modern Heuristic Search Methods (V. J. Rayward-Smith, I. H. Osman, C. R. Reeves, and G. D. Smith, eds.), John Wiley and Sons, 1996, pp. 61–83.

[12] R. Battiti and M. Protasi, *Reactive search, a history-sensitive heuristic for MAX-SAT*, ACM Journal of Experimental Algorithmics **2** (1997), no. 2, http://www.jea.acm.org/.

[13] ———, *Solving MAX-SAT with non-oblivious functions and history-based heuristics*, Satisfiability Problem: Theory and Applications, DIMACS: Series in Discrete Mathematics and Theoretical Computer Science, no. 35, AMS and ACM Press, 1997.

[14] R. Battiti and G. Tecchiolli, *The reactive tabu search*, ORSA Journal on Computing **6** (1994), no. 2, 126–140.

[15] C.E. Blair, R.G. Jeroslow, and J.K. Lowe, *Some results and experiments in programming for propositional logic*, Computers and Operations Research **13** (1986), no. 5, 633–645.

[16] M. Boehm and E. Speckenmeyer, *A fast parallel sat solver - efficient workload balancing*, Annals of Mathematics and Artificial Intelligence **17** (1996), 381–400.

[17] A. Broder, A. Frieze, and E. Upfal, *On the satisfiability and maximum satisfiability of random 3-CNF formulas*, Proc. of the 4th Annual ACM-SIAM Symp. on Discrete Algorithms, 1993.

[18] M. Buro and H. Kleine Buening, *Report on a SAT competition*, EATCS Bulletin **49** (1993), 143–151.

[19] S. Chakradar, V. Agrawal, and M. Bushnell, *Neural net and boolean satisfiability model of logic circuits*, IEEE Design and Test of Computers (1990), 54–57.

[20] M.-T. Chao and J. Franco, *Probabilistic analysis of two heuristics for the 3-satisfiability problem*, SIAM Journal on Computing **15** (1986), 1106–1118.

[21] J. Chen, D. Friesen, and H. Zheng, *Tight bound on Johnson's algorithm for MAX-SAT*, Proc. 12th Annual IEEE Conf. on Computational Complexity, Ulm, Germany, 1997, pp. 274–281.

[22] J. Cheriyan, W. H. Cunningham, T. Tuncel, and Y. Wang, *A linear programming and rounding approach to MAX 2-SAT*, Proc. of the Second DIMACS Algorithm Implementation Challenge on Cliques, Coloring and Satisfiability (M. Trick and D. S. Johson, eds.), DIMACS Series on Discrete Mathematics and Theoretical Computer Science, no. 26, 1996, pp. 395–414.

[23] V. Chvatal and B. Reed, *Mick gets some (the odds are on his side)*, Proc. 33th Ann. IEEE Symp. on Foundations of Computer Science, IEEE Computer Society, 1992, pp. 620–627.

[24] V. Chvátal and E. Szemerédi, *Many hard examples for resolution*, Journal of the ACM **35** (1988), 759–768.

[25] S.A. Cook, *The complexity of theorem-proving procedures*, Proc. of the Third Annual ACM Symp. on the Theory of Computing, 1971, pp. 151–158.

[26] S.A. Cook and D.G. Mitchell, *Finding hard instances of the satisfiability problem: a survey*, Satisfiability Problem: Theory and Applications

(D.-Z. Du, J. Gu, and P.M. Pardalos, eds.), DIMACS Series in Discrete Mathematics and Theoretical Computer Science, vol. 35, AMS and ACM Press, 1997.

[27] P. Crescenzi and A. Panconesi, *Completeness in approximation classes*, Information and Computation **93** (1991), 241–262.

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

[29] M. Davis and H. Putnam, *A computing procedure for quantification theory*, Journal of the ACM **7** (1960), 201–215.

[30] D. Du, J. Gu, and P.M. Pardalos (Eds.), *Satisfiability problem: Theory and applications*, DIMACS Series in Discrete Mathematics and Theoretical Computer Science, vol. 35, AMS and ACM Press, 1997.

[31] O. Dubois and Y. Boufkhad, *A general upper bound for the satisfiability threshold of random r-SAT formulas*, Tech. report, LAFORIA, CNRS-Univ. Paris 6, 1996.

[32] U. Feige and M.X. Goemans, *Approximating the value of two proper proof systems, with applications to MAX-2SAT and MAX-DICUT*, Proc. of the Third Israel Symp. on Theory of Computing and Systems, 1995, pp. 182–189.

[33] J. Franco and M. Paull, *Probabilistic analysis of the davis-putnam procedure for solving the satisfiability problem*, Discrete Applied Mathematics **5** (1983), 77–87.

[34] A. Frieze and S. Suen, *Analysis of two simple heuristics on a random instance of k-SAT*, Journal of Algorithms **20** (1996), 312–355.

[35] H. Gallaire, J. Minker, and J. M. Nicolas, *Logic and databases: a deductive approach*, Computing Surveys **16** (1984), no. 2, 153–185.

[36] I.P. Gent and T. Walsh, *An empirical analysis of search in gsat*, Journal of Artificial Intelligence Research **1** (1993), 47–59.

[37] ———, *Towards an understanding of hill-climbing procedures for SAT*, Proc. of the Eleventh National Conf. on Artificial Intelligence, AAAI Press / The MIT Press, 1993, pp. 28–33.

[38] F. Glover, *Tabu search - part I*, ORSA Journal on Computing **1** (1989), no. 3, 190–260.

[39] M.X. Goemans and D.P. Williamson, *New 3/4-approximation algorithms for the maximum satisfiability problem*, SIAM Journal on Discrete Mathematics **7** (1994), no. 4, 656–666.

[40] ———, *Improved approximation algorithms for maximum cut and satisfiability problems using semidefinite programming*, Journal of the ACM **42** (1995), no. 6, 1115–1145.

[41] A. Goerdt, *A threshold for unsatisfiability*, Journal of Computer and System Sciences **53** (1996), 469–486.

[42] J. Gu, *Efficient local search for very large-scale satisfiability problem*, ACM SIGART Bulletin **3** (1992), no. 1, 8–12.

[43] ———, *Global optimization for satisfiability (SAT) problem*, IEEE Transactions on Data and Knowledge Engineering **6** (1994), no. 3, 361–381.

[44] J. Gu, Q.-P. Gu, and D.-Z.Du, *Convergence properties of optimization algorithms for the SAT problem*, IEEE Transactions on Computers **45** (1996), no. 2, 209–219.

[45] J. Gu, P.W. Purdom, J. Franco, and B.W. Wah, *Algorithms for the satisfiability (SAT) problem: A survey*, Satisfiability Problem: Theory and Applications (D.-Z. Du, J. Gu, and P.M. Pardalos, eds.), DIMACS Series in Discrete Mathematics and Theoretical Computer Science, vol. 35, AMS and ACM Press, 1997.

[46] J. Gu and R. Puri, *Asynchronous circuit synthesis with boolean satisfiability*, IEEE Transactions on Computer-Aided Design of Integrated Circuits **14** (1995), no. 8, 961–973.

[47] P.L. Hammer, P. Hansen, and B. Simeone, *Roof duality, complementation and persistency in quadratic 0-1 optimization*, Mathematical Programming **28** (1984), 121–155.

[48] P. Hansen and B. Jaumard, *Algorithms for the maximum satisfiability problem*, Computing **44** (1990), 279–303.

[49] J.N. Hooker, *Resolution vs. cutting plane solution of inference problems: some computational experience*, Operations Research Letters **7** (1988), no. 1, 1–7.

[50] J. Håstad, *Some optimal inapproximability results*, Proc. 28th Annual ACM Symp. on Theory of Computing, El Paso, Texas, 1997, pp. 1–10.

[51] B. Jaumard, M. Stan, and J. Desrosiers, *Tabu search and a quadratic relaxation for the satisfiability problem*, Proc. of the Second DIMACS Algorithm Implementation Challenge on Cliques, Coloring and Satisfiability (M. Trick and D. S. Johson, eds.), DIMACS Series on Discrete Mathematics and Theoretical Computer Science, no. 26, 1996, pp. 457–477.

[52] D.S. Johnson, *Approximation algorithms for combinatorial problems*, Journal of Computer and System Sciences **9** (1974), 256–278.

[53] D.S. Johnson and M. Trick (Eds.), *Cliques, coloring, and satisfiability: Second DIMACS implementation challenge*, vol. 26, DIMACS Series in Discrete Mathematics and Theoretical Computer Science, no. 26, AMS, 1996.

[54] J.L. Johnson, *A neural network approach to the 3-satisfiability problem*, Journal of Parallel and Distributed Computing **6** (1989), 435–449.

[55] A. Kamath, R. Motwani, K. Palem, and P. Spirakis, *Tail bounds for occupancy and the satisfiability threshold conjecture*, Random Structures and Algorithms **7** (1995), 59–80.

[56] A.P. Kamath, N.K. Karmarkar, K.G. Ramakrishnan, and M.G. Resende, *Computational exprience with an interior point algorithm on the satisfiability problem*, Annals of Operations Research **25** (1990), 43–58.

[57] ———, *A continuous approach to inductive inference*, Mathematical programming **57** (1992), 215–238.

[58] H. Karloff and U. Zwick, *A 7/8-approximation algorithm for MAX 3SAT?*, Proc. of the 38th Annual IEEE Symp. on Foundations of Computer Science, IEEE Computer Society, 1997, in press.

[59] S. Khanna, R.Motwani, M.Sudan, and U.Vazirani, *On syntactic versus computational views of approximability*, Proc. 35th Ann. IEEE Symp. on Foundations of Computer Science, IEEE Computer Society, 1994, pp. 819–836.

[60] S. Kirkpatrick, C.D. Gelatt Jr., and M.P. Vecchi, *Optimization by simulated annealing*, Science **220** (1983), 671–680.

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

[62] L. M. Kirousis, E. Kranakis, and D. Krizanc, *Approximating the unsatisfiability threshold of random formulas*, Proc. of the Fourth Annual European Symp. on Algorithms (Barcelona), Springer-Verlag, September 1996, pp. 27–38.

[63] E. Koutsoupias and C.H. Papadimitriou, *On the greedy algorithm for satisfiability*, Information Processing Letters **43** (1992), 53–55.

[64] O. Kullmann and H. Luckhardt, *Deciding propositional tautologies: Algorithms and their complexity*, Tech. Report 1596, JohannWolfgang Goethe-Univ., Fachbereich Mathematik, Frankfurt, Germany, January 1997.

[65] D.W. Loveland, *Automated theorem proving: A logical basis*, North-Holland, 1978.

[66] S. Minton, M. D. Johnston, A. B. Philips, and P. Laird, *Solving large-scale constraint satisfaction and scheduling problems using a heuristic repair method*, Proc. of the 8th National Conf. on Artificial Intelligence (AAAI-90), 1990, pp. 17–24.

[67] D. Mitchell, B. Selman, and H. Levesque, *Hard and easy distributions of SAT problems*, Proc. of the 10th National Conf. on Artificial Intelligence (AAAI-92) (San Jose, Ca), July 1992, pp. 459–465.

[68] R. Motwani and P. Raghavan, *Randomized algorithms*, Cambridge University Press, New York, 1995.

[69] T.A. Nguyen, W.A. Perkins, T.J. Laffrey, and D. Pecora, *Checking an expert system knowledge base for consistency and completeness*, Proc. of the International Joint Conf. on Artificial Intelligence (Los Altos, CA), 1985, pp. 375–378.

[70] P. Nobili and A. Sassano, *Strengthening lagrangian bounds for the MAX-SAT problem*, Tech. Report 96-230, Institut fuer Informatik, Koln Univ., Germany, 1996, Proc. of the Work. on the Satisfiability Problem, Siena, Italy (J. Franco and G. Gallo and H. Kleine Buening, Eds.).

[71] P. Orponen and H. Mannila, *On approximation preserving reductions: complete problems and robust measures*, Tech. Report C-1987-28, Dept. of Computer Science, Univ. of Helsinki, 1987.

[72] C. H. Papadimitriou, *On selecting a satisfying truth assignment (extended abstract)*, Proc. of the 32th Annual Symp. on Foundations of Computer Science, 1991, pp. 163–169.

[73] C.H. Papadimitriou and K. Steiglitz, *Combinatorial optimization, algorithms and complexity*, Prentice-Hall, NJ, 1982.

[74] R. Puri and J. Gu, *A BDD SAT solver for satisfiability testing: an industrial case study*, Annals of Mathematics and Artificial Intelligence **17** (1996), no. 3-4, 315–337.

[75] M.G.C. Resende and T. A. Feo, *A grasp for satisfiability*, Proc. of the Second DIMACS Algorithm Implementation Challenge on Cliques, Coloring and Satisfiability (M. Trick and D. S. Johson, eds.), DIMACS Series on Discrete Mathematics and Theoretical Computer Science, no. 26, 1996, pp. 499–520.

[76] M.G.C. Resende, L.S. Pitsoulis, and P.M. Pardalos, *Approximate solution of weighted MAX-SAT problems using GRASP*, Satisfiability Problem: Theory and Applications, DIMACS: Series in Discrete Mathematics and Theoretical Computer Sc ience, no. 35, 1997.

[77] J. A. Robinson, *A machine-oriented logic based on the resolution principle*, Journal of the ACM **12** (1965), 23–41.

[78] B. Selman and H. Kautz, *Domain-independent extensions to GSAT: Solving large structured satisfiability problems*, Proc. of the International Joint Conf. on Artificial Intelligence, 1993, pp. 290–295.

[79] B. Selman and H.A. Kautz, *An empirical study of greedy local search for satisfiability testing*, Proc. of the 11th National Conf. on Artificial Intelligence (AAAI-93) (Washington, D. C.), 1993.

[80] B. Selman, H.A. Kautz, and B. Cohen, *Local search strategies for satisfiability testing*, Proc. of the Second DIMACS Algorithm Implementation Challenge on Cliques, Coloring and Satisfiability (M. Trick and D. S. Johson, eds.), DIMACS Series on Discrete Mathematics and Theoretical Computer Science, no. 26, 1996, pp. 521–531.

[81] B. Selman, H. Levesque, and D. Mitchell, *A new method for solving hard satisfiability problems*, Proc. of the 10th National Conf. on Artificial Intelligence (AAAI-92) (San Jose, Ca), July 1992, pp. 440–446.

[82] W.M. Spears, *Simulated annealing for hard satisfiability problems*, Proc. of the Second DIMACS Algorithm Implementation Challenge on Cliques, Coloring and Satisfiability (M. Trick and D. S. Johnson, eds.), DIMACS Series on Discrete Mathematics and Theoretical Computer Science, no. 26, 1996, pp. 533–555.

[83] L. Trevisan, *Approximating satisfiable satisfiability problems*, Proc. of the 5th Annual European Symp. on Algorithms, Graz, Springer Verlag, 1997, pp. 472–485.

[84] M. Yannakakis, *On the approximation of maximum satisfiability*, Journal of Algorithms **17** (1994), 475–502.