LCNS, Vol 1767, pp. 174–186, Springer 2003

Faster Exact Solutions for Max2Sat

Jens Gramm Rolf Niedermeier

Wilhelm-Schickard-Institut für Informatik, Universität Tübingen,
Sand 13, D-72076 Tübingen, Fed. Rep. of Germany
gramm,niedermr@informatik.uni-tuebingen.de

Abstract. Given a boolean 2CNF formula F, the Max2Sat problem is that of finding the maximum number of clauses satisfiable simultaneously. In the corresponding decision version, we are given an additional parameter k and the question is whether we can simultaneously satisfy at least k clauses. This problem is *NP*-complete. We improve on known upper bounds on the worst case running time of Max2Sat, implying also new upper bounds for Maximum Cut. In particular, we give experimental results, indicating the practical relevance of our algorithms.
Keywords: *NP*-complete problems, exact algorithms, parameterized complexity, Max2Sat, Maximum Cut.

1 Introduction

The (unweighted) *Maximum Satisfiability* problem (MaxSat) is to assign values to boolean variables in order to maximize the number of satisfied clauses in a CNF formula. Restricting the clause size to two, we obtain Max2Sat. When turned into "yes–no" problems by adding a goal k representing the number of clauses to be satisfied, MaxSat and Max2Sat are *NP*-complete [7]. Efficient algorithms for MaxSat, as well as Max2Sat, have received considerable interest over the years [2]. Furthermore, there are several papers which deal with Max2Sat in detail, e.g., [3, 4, 6]. These papers present approximation and heuristic algorithms for Max2Sat. In this paper, by way of contrast, we introduce algorithms that give optimal solutions within provable bounds on the running time. The arising solutions for Max2Sat are both fast and exact and show themselves to be interesting not only from a theoretical point of view, but also from a practical point of view due to the promising experimental results we have found.

The following complexity bounds are known for Max2Sat: There is a deterministic, polynomial time approximation algorithm with approximation factor 0.931 [6]. On the other hand, unless $P = NP$, the approximation factor cannot be better than 0.955 [9]. With regard to exact algorithms, research so far has concentrated on the general MaxSat

problem [1, 12]. As a rule, the algorithms which are presented there (as well as our own) are based on elaborate case distinctions. Taking the case distinctions in [12] further, Bansal and Raman [1] have recently presented the following results: Let $|F|$ be the length of the given input formula and K be the number of clauses in F. Then MaxSat can be solved in times $O(1.3413^K|F|)$ and $O(1.1058^{|F|})$. The latter result implies that, using $|F| = 2K$, Max2Sat can be solved in time $O(1.2227^K)$, this being the best known result for Max2Sat so far. Moreover, Bansal and Raman have shown that, given the number k of clauses which are to be satisfied in advance, MaxSat can be solved in $O(1.3803^k k^2 + |F|)$ time.

Our main results are as follows: Max2Sat can be solved in times $O(1.0970^{|F|})$, $O(1.2035^K)$, and $O(1.2886^k k + |F|)$, respectively. In addition, we show that if each variable in the formula appears at most three times, then Max2Sat, still *NP*-complete, can be solved in time $O(1.2107^k|F|)$. In reference to modifications of our algorithms done in [8], we find that Maximum Cut in a graph with n vertices and m edges can be solved in time $O(1.3197^m)$. If restricted to graphs with vertex degree at most three, it can be solved in time $O(1.5160^n)$, and, if restricted to graphs with vertex degree at most four, in time $O(1.7417^n)$. In addition, the same algorithm computes a Maximum Cut of size at least k in time $O(m + n + 1.7445^k k)$, improving on the previous time bounds of $O(m + n + 4^k k)$ [11] and $O(m + n + 2.6196^k k)$ [12].

Aside from the theoretical improvements gained by the new algorithms we have developed, an important contribution of our work is also to show the practical significance of the results obtained. Although our algorithms are based on elaborate case distinctions which show themselves to be complicated upon analysis, they are relatively easy to apply when dealing with the number of cases the actual algorithm has to distinguish. Unlike what is known for the general MaxSat problem [1, 12], we thereby have for Max2Sat a comparatively small number of easy to check cases, making our implementation practical. Moreover, analyzing the frequency of how often different rules are applied, our experiments also indicate which rules might be the most valuable ones. Our algorithms can compete well with heuristic ones, such as the one described by Borchers and Furman [3].

Independent from our work, Hirsch [10] has simultaneously developed upper bounds for the Max2Sat problem. He presents an algorithm with bounds of $O(1.0905^{|F|})$ with respect to the formula length $|F|$ and $O(1.1893^K)$ with respect to the number of clauses K, which are better than the bounds shown for our algorithms. Moreover, he points out that his algorithm also works for weighted versions of Max2Sat. On the other

hand, however, he does not give any bound with respect to k, the number of satisfiable clauses. His analysis is simpler than ours, as he makes use of a result by Yannakakis [15]. The algorithm itself, however, seems much more complex and is not yet accompanied by an implementation. The reduction step of Hirsch's algorithm has a polynomial complexity, as a maximum flow computation has to be done, and it would be interesting to see whether this will turn out to be efficient in practice.

Due to the lack of space, we omitted several details and refer to [8] for more material.

2 Preliminaries and transformation rules

We use primarily the same notation as in [1, 12]. We study boolean formulas in 2CNF, represented as multisets of sets (clauses). A subformula, i.e., a subset of clauses, is denoted *closed* if it is a minimal subset of clauses allowing no variable within the subset to occur outside of this subset as well. A clause that contains the same variable positively and negatively, e.g., $\{x, \bar{x}\}$, is satisfied by every assignment. We will not allow for these clauses here, and assume that such clauses are always replaced by a special clause $\top$, denoting a clause that is always satisfied. We call a clause containing r literals simply an *r-clause*. Its *length* is therefore r. A formula in *2CNF* is one consisting of 1- and 2-clauses. We assume that 0-clauses do not appear in our formula, since they are clearly unsatisfiable. The *length of a clause* is its cardinality, and the *length of a formula* is the sum of the lengths of its clauses. Let l be a literal occurring in a formula F. We call it an *(i, j)-literal* if the variable corresponding to l occurs exactly i times as l and exactly j times as $\bar{l}$. Analogously, we obtain (i^+, j)-, (i, j^+)-, and *(i^+, j^+)-literals* by replacing "exactly" with "at least" at the appropriate positions, and get (i^-, j)-, (i, j^-)- and *(i^-, j^-)-literals* by replacing "exactly" with "at most". Following Bansal and Raman [1], we call an (i, j)-literal an $(i, j)[p_1, \ldots, p_i][n_1, \ldots, n_j]$-literal if the clauses containing l are of length $p_1 \leq \ldots \leq p_i$ and those containing $\bar{l}$ are of length $n_1 \leq \ldots \leq n_j$. For a literal l and a formula F, let $F[l]$ be the formula originating from F by replacing all clauses containing l with $\top$ and removing $\bar{l}$ from all clauses where it occurs. We say $\tilde{x}$ occurs in a clause C if $x \in C$ or $\bar{x} \in C$. We write $\#_{\tilde{x}}$ for the number of occurrences of $\tilde{x}$ in the formula. Should variable $\tilde{x}$ and variable $\tilde{y}$ occur in the same clause, we call this instance a *common occurrence* and write $\#_{\tilde{x}\tilde{y}}$ for the number of their common occurrences in the formula. In the same way, we write $\#_{xy}$ for the number of common occurrences of literals x and y.

As with earlier exact algorithms for MaxSat [1, 12], our algorithms are recursive. They go through a number of transformations and branching rules, where the given formula is simplified by assigning boolean values to some carefully selected variables. The fundamental difference between transformation and branching rules is that when the former has been given a formula, it is replaced by *one* simpler formula, whereas in the latter a formula is replaced by *at least two* simpler formulas. The asymptotic complexity of the algorithm is governed by the branching rules. We will use recurrences to describe the size of the corresponding *branching trees* created by our algorithms. Therefore, we will apply one of the transformation rules whenever possible, as they avoid a branching of recursion.

In the rest of this section, we turn our attention to the transformation rules. Our work here follows that of [12] closely, as the first 4 rules have also been used there. Their correctness is easy to check.

1. *Pure Literal Rule:* Replace F with $F[l]$ if l is a $(1^+, 0)$-literal.
2. *Dominating 1-Clause Rule:* If $\bar{l}$ occurs in i clauses and l occurs in at least i 1-clauses of F, then replace F with $F[l]$.
3. *Complementary 1-Clause Rule:* If $F = \{\{x\}, \{\bar{x}\}\} \cup G$, then replace F with G, increasing the number of satisfied clauses by one.
4. *Resolution Rule:* If $F = \{\{x, l_1\}, \{\bar{x}, l_2\}\} \cup G$ and G does not contain $\tilde{x}$, then replace F with $\{\{l_1, l_2\}\} \cup G$, increasing the number of satisfied clauses by one.
5. *Almost Common Clauses Rule:* If $F = \{\{x, y\}, \{x, \bar{y}\}\} \cup G$, then replace F with $\{x\} \cup G$, increasing the number of satisfied clauses by one.
6. *Three Occurrence Rules:* We consider two subcases:
   (a) If x is a $(2, 1)$-literal, $F = \{\{x, y\}, \{x, y\}, \{\bar{x}, \bar{y}\}\} \cup G$, and G does not contain $\tilde{x}$, then replace F with G, increasing the number of satisfied clauses by three.
   (b) If x is $(2, 1)$-literal, and either $F = \{\{x, y\}, \{x, y\}, \{\bar{x}, l_1\}\} \cup G$ or $F = \{\{x, y\}, \{x, l_1\}, \{\bar{x}, \bar{y}\}\} \cup G$, then replace F with $\{\{y, l_1\}\} \cup G$ or $\{\{\bar{y}, l_1\}\} \cup G$, respectively, increasing the number of satisfied clauses by two.

The Almost Common Clauses Rule was introduced by Bansal and Raman [1]. In the rest of this paper, we will call a formula *reduced* if none of the above transformation rules can be applied to it. The correctness of many of the branching rules that we will present relies heavily on the fact that we are dealing with reduced formulas.

3 A bound in the number of satisfiable clauses

Theorem 1 *For a 2CNF formula F, it can be computed in time $O(|F| + 1.2886^k k)$ whether or not at least k clauses are simultaneously satisfiable.*

Theorem 1 is of special interest in so-called parameterized complexity theory [5]. The corresponding bound for formulas in CNF is $O(|F| + 1.3803^k k^2)$ [1]. In this expression 1.3803^k gives an estimation of the branching tree size. The time spent in each node of the tree is $O(|F|)$, which for CNF formulas is shown to be bounded by k^2[11]. For 2CNF formulas, however, we can improve this factor for every node of the tree from k^2 to k: Note that the case where $k \leq \lceil \frac{K}{2} \rceil$ with K as the number of clauses is trivial, since for a random assignment, either this assignment or its inverse satisfy $\lceil \frac{K}{2} \rceil$ clauses. For $k > \lceil \frac{K}{2} \rceil$, however, Max2Sat formulas have $|F| = O(k)$.

Before sketching the remaining proof of Theorem 1, we give a corollary. Consider a 2CNF input formula in which every variable occurs at most three times. This problem is also *NP*-complete [13], but we can improve our upper bounds by excluding some of the cases necessary for general 2CNF formulas, thus obtaining a better branching than in Theorem 1. We omit details.

Corollary 2 *For a 2CNF formula F where every variable occurs at most three times, it can be computed in time $O(|F| + 1.2107^k k)$ whether or not at least k clauses are simultaneously satisfiable.*

We now sketch the proof of Theorem 1. We present algorithm A with the given running time. As an invariant of our algorithm, observe that the subsequently described branching rules are only applied if the formula is reduced, that is, there is no transformation rule to apply. The idea of branching is based on dividing the search space, i.e. the set of all possible assignments, into several parts, finding an optimal assignment within each part, and then taking the best of them. Carefully selected branchings enable us to simplify the formula in some of the branches. Observe that the subsequent order of the steps is important. In each step, the algorithm always executes the applicable branching rule with the lowest possible number:

RULE 1: If there is a $(9^+, 1)$-, $(6^+, 2)$-, or $(4^+, 3^+)$-literal x, then we branch into $F[x]$ and $F[\bar{x}]$. The correctness of this rule is clear. In the worst case, a $(4, 3)$-literal, by branching into $F[x]$, we may satisfy 4 clauses and by branching into $F[\bar{x}]$, we may satisfy 3 clauses. We describe

this situation by saying that we have a *branching vector* $(4,3)$, which expresses the corresponding recurrence for the search tree size, solvable by standard methods (cf. [1, 12]). Solving the corresponding recurrence for the branching tree size, we obtain here the *branching number* 1.2208. This means that were we always to branch according to a $(4,3)$-literal, the branching tree size would be bounded by 1.2208^k. It is easy to check that branching vectors $(9,1)$ and $(6,2)$ yield better (i.e., smaller) branching numbers.

RULE 2: If there is a $(2,1)$-literal x, such that $F = \{\{x,y\},\{x,z\}\} \cup G$ and y occurs at least as often in F as z, then branch as follows: If both y and z are $(2,1)$-literals, branch into $F[x]$ and $F[\bar{x}]$. We can show a worst case branching vector of $(4,5)$ in these situations. Otherwise, i.e., if one of y and z is not $(2,1)$, then branch into $F[y]$ and $F[\bar{y}]$. The correctness is again obvious. However, the complexity analysis (i.e., analysis of the branching vectors) is significantly harder in this case. Keep in mind that the formula is reduced, meaning that we may exclude all cases where a transformation rule would apply.

First, we distinguish according to the number of common occurrences of $\tilde{x}$ and $\tilde{y}$: Assuming that there are three common occurrences we either have clauses $\{x,y\}$, $\{x,\bar{y}\}$, clauses $\{x,y\}$, $\{\bar{x},y\}$, or clauses $\{x,y\}$, $\{x,y\}$, $\{\bar{x},\bar{y}\}$. In the first two cases, the Almost Common Clause Rule applies (cf. Section 2), and in the latter case, the first of the Three Occurrence Rules applies. Analogously, assuming two common occurrences, either the Almost Common Clause Rule or the second of the Three Occurrence Rules applies. Hence, because the formula is reduced, we can neglect these cases.

It remains to consider only one common occurrence of $\tilde{x}$ and $\tilde{y}$. We make the following observation: By satisfying y, we reduce literal x to occurrence two and the Resolution Rule applies, eliminating $\tilde{x}$ and satisfying one additional clause. On the other hand, satisfying $\bar{y}$ leaves a unit occurrence of x and the Dominating 1-Clause Rule applies, eliminating $\tilde{x}$ from the formula and satisfying the two x-clauses. Now we consider each possible occurrence pattern for literal y. If y occurs at least four times, it is a $(3^+,1)$-, $(1,3^+)$-, or a $(2^+,2^+)$-literal, and using the given observation, in the worst cases we obtain branching vectors $(4,3)$, $(2,5)$, or $(3,4)$. If y occurs only three times, it is a $(2,1)$- or $(1,2)$-literal. We then take a literal z into consideration as well. We know from the way in which y was chosen that the literal z is also of occurrence three. We consider all combinations of y and z, which are either $(2,1)$- or $(1,2)$, and also cover a possible common occurrence of $\tilde{y}$ and $\tilde{z}$ in one clause. Branching

as specified, we in the worst case obtain a branching vector $(2, 6)$, namely when both y and z are $(1, 2)$ and there is no common clause of y and z. We omit the details here.

Summarizing, for RULE 2, the worst observed branching vector is $(2, 5)$, which corresponds to the branching number 1.2365.

RULE 3: If there is a $(3^+, 3^+)$- or $(4^+, 2)$-literal x, then branch into $F[x]$ and $F[\bar{x}]$. Trivially we get the branching vectors $(3, 3)$ and $(4, 2)$, implying the branching numbers 1.2600 and 1.2721.

RULE 4: If there is a $(c, 1)$-literal x with $c \in \{3, 4, 5, 6, 7, 8\}$, then choose a literal y occurring in a clause $\{x, y\}$ and branch into $F[y]$ and $F[\bar{y}]$. Again, this is clearly correct.

With regard to the complexity analysis, we observe that by satisfying $\bar{y}$, a unit occurrence of x arises and the Dominating 1-Clause Rule applies, satisfying all x-clauses. Having reached RULE 4, we know that all literals in the formula occur at least four times, as the 3-occurrences are eliminated by RULE 2. We consider different possible cases for y, namely y being a $(3^+, 1)$-, $(1, 3^+)$-, or $(2^+, 2^+)$-literal, and we consider all possible numbers of common occurrences of $\tilde{x}$ and $\tilde{y}$. Using the given observation, we can show a branching vector of $(1, 6)$ in the worst case, namely for a $(3, 1)$-literal x, a $(1, 3)$-literal y, and $\#_{\tilde{x}\tilde{y}} = 1$. This corresponds to the branching number 1.2852. Again, we omit the details.

RULE 5: By this stage, there remain only $(2, 2)$-, $(3, 2)$-, or $(2, 3)$-literals in the formula. RULE 5 deals with the case that there is a $(2, 2)$-literal x. Our branching rule now is more involved. We choose a literal y occurring in a clause $\{x, y\}$ and a literal z occurring in a clause $\{\bar{x}, z\}$. For $\tilde{x}$ having at least two common occurrences with $\tilde{y}$ or $\tilde{z}$, we branch into $F[x]$ and $F[\bar{x}]$. If this is not the case but $\tilde{y}$ and $\tilde{z}$ have at least two common occurrences, we branch into $F[y]$ and $F[\bar{y}]$. It remains that $\#_{\tilde{x}\tilde{y}} = 1$, $\#_{\tilde{x}\tilde{z}} = 1$, and $\#_{\tilde{y}\tilde{z}} \leq 1$. If $\tilde{y}$ and $\tilde{z}$ have a common occurrence in a clause $\{y, z\}$, we branch into $F[y]$, $F[\bar{y}z]$, and $F[\bar{y}\bar{z}]$. If not, i.e. there is no clause $\{y, z\}$, we branch into $F[yz]$, $F[\bar{y}z]$, $F[y\bar{z}]$, and $F[\bar{y}\bar{z}]$. It is easy to verify that we have covered all possible cases.

Regarding the complexity analysis, we first make use of the following: Whenever two literals being $(2, 2)$ or $(3, 2)$ have at least two common occurrences, we can take one of them and branch setting it true and false. In the worst case, this results in the branching vector $(2, 5)$ with branching number 1.2366.

Thus, we are only left with situations in which $\#_{\tilde{x}\tilde{y}} = 1$, $\#_{\tilde{x}\tilde{z}} = 1$, and $\#_{\tilde{y}\tilde{z}} \leq 1$. For these cases, we consider all arrangements of $\tilde{x}$, $\tilde{y}$ and $\tilde{z}$ possible, with x being $(2, 2)$, y being $(2^+, 2^+)$ and z being $(2^+, 2^+)$.

We obtain "good" branching numbers of 1.2886 for vectors as, such as $(5,6,5,6)$ in most cases by branching into $F[yz]$, $F[\bar{y}z]$, $F[y\bar{z}]$, and $F[\bar{y}\bar{z}]$. Only for a possible common occurrence of $\tilde{y}$ and $\tilde{z}$ in a clause $\{y,z\}$ would the branching number be worse. We avoid this by branching into $F[y]$, $F[\bar{y}z]$, and $F[\bar{y}\bar{z}]$ instead. Here, we study in more detail what happens in the single subcases: Setting y true in the first subcase of the branching, we satisfy two y-clauses. By setting y false and z true in the second subcase, we directly eliminate two $\bar{y}$- and two z-clauses. Consequently, the Dominating Unit Clause Rule now applies for x and satisfies two additional clauses. In total, we satisfy six clauses in the second subcase. Setting y and z false in the third subcase, we satisfy two $\bar{y}$- and two $\bar{z}$-clauses. In addition, there arise unit clauses for x and $\bar{x}$ such that the Complementary 1-Clause Rule and then the Resolution Rule apply, satisfying two additional clauses. Summarizing these considerations, the resulting branching vector is $(2,6,6)$ with branching number 1.3022.

For our purpose, this vector is still not good enough. However, we observe that in the first branch $\tilde{x}$, is reduced to occurrence three, meaning that in this branch the next rule that will be applied will undoubtly be RULE 2. We recall that RULE 2 yields the branching vector $(2,5)$, and possibly even a better one. Combining these two steps, we obtain the branching vector $(4,7,6,6)$ and the branching number 1.2812.

Note that in RULE 5, we have the real worst case of the algorithm, namely for the situation of $\#_{\tilde{x}\tilde{y}} = \#_{\tilde{x}\tilde{z}} = \#_{\tilde{y}\tilde{z}} = 1$ and $\tilde{y}$ and $\tilde{z}$ having their common occurrence in a clause $\{\bar{y},\bar{z}\}$. For this situation, we can find no branching rule improving the branching number 1.2886.

RULE 6: When this rule applies, all literals in the formula are either $(3,2)$ or $(2,3)$. We choose a $(3,2)$-literal x. The branching instruction is now primarily the same as in RULE 5 above. However, it is now possible that there is no literal z occurring in a clause $\{\bar{x},z\}$, as the two $\bar{x}$-occurrences may be in unit clauses. In this case, i.e. for two $\bar{x}$-unit clauses, we branch into $F[y]$ and $F[\bar{y}]$. Having two or more common occurrences for a pair of $\tilde{x}$, $\tilde{y}$, and $\tilde{z}$, we branch as in RULE 5. For the remaining cases, i.e. $\#_{\tilde{x}\tilde{y}} = 1$, $\#_{\tilde{x}\tilde{z}} = 1$, and $\#_{\tilde{y}\tilde{z}} \leq 1$, we branch into $F[y]$, $F[\bar{y}z]$, and $F[\bar{y}\bar{z}]$.

The complexity analysis works analogously to RULE 5. For $\#_{\tilde{x}\tilde{y}} = 1$, $\#_{\tilde{x}\tilde{z}} = 1$, and $\#_{\tilde{y}\tilde{z}} \leq 1$ we test all possible arrangements of $\tilde{x}$, $\tilde{y}$, and $\tilde{z}$ with x being $(3,2)$ and y and z being either $(3,2)$ or $(2,3)$. The worst case branching vector in these situations, when branching into $F[y]$, $F[\bar{y}z]$, and $F[\bar{y}\bar{z}]$, is $(2,9,5)$ and yields the branching number 1.2835. Again, we omit the details.

4 A bound in the formula length

Compare Theorem 3 with the $O(1.1058^{|F|})$ time bound for MaxSat [1]. Observe that when the exponential bases are close to 1, even small improvements in the exponential base can mean significant progress.

Theorem 3 Max2Sat *can be solved in time* $O(1.0970^{|F|})$.

We sketch the proof of Theorem 3, presenting Algorithm B with the given running time. For the most part, it is equal to Algorithm A, sharing the branching instructions of RULEs 1 to 4. Taking up ideas given in [1], we replace RULE 5 and 6 with new branching instructions RULE $5'$, $6'$, $7'$, and $8'$.

For the rules known from Algorithm A, it remains to examine their branching vectors with respect to formula length. As the analysis is in essence the same as that of the proof for Theorem 1, we omit the details once again, while only stating that the worst case branching vector with respect to formula length for RULEs 1 to 4 is $(7, 8)$ (branching number 1.0970), and continue with the new instructions:

RULE $5'$**:** Upon reaching this rule, all literals in the formula are of type $(2, 2)$, $(3, 2)$, or $(2, 3)$. RULE $5'$ deals with the case that there is a $(3, 2)$-literal x, which is not $(3, 2)[2, 2, 2][1, 2]$.

If x is a $(3, 2)[2, 2, 2][2, 2]$-literal, we branch into $F[x]$ and $F[\bar{x}]$. Counting the literals eliminated in either branch, we easily obtain a branching vector of $(8, 7)$.

If x is $(3, 2)[2, 2, 2][1, 1]$-literal with clauses $\{x, y_1\}$, $\{x, y_2\}$, and $\{x, y_3\}$ in which some of y_1, y_2, and y_3 may be equal, we branch into $F[\bar{x}]$ and $F[x\bar{y_1}\bar{y_2}\bar{y_3}]$. This is correct, as should we want to satisfy more clauses by setting x to true than by setting x to false, all y_1, y_2 and y_3 must be falsified. We easily check that if all y_1, y_2, and y_3 are equal, we obtain a branching vector of $(10, 10)$. For at least two literals of y_1, y_2, and y_3 being distinct, we eliminate in the first subcase eight literals, namely the literals in the satisfied x-clauses and the falsified $\bar{x}$-literals. In the second subcase, we eliminate $\tilde{x}$, having five occurrences and two variables having at least four occurrences. This gives a branching vector of $(8, 13)$, corresponding to the branching number 1.0866.

If x is ultimately a $(3, 2)[1, 2, 2][2, 2]$-literal with clauses $\{\bar{x}, z_1\}$, $\{\bar{x}, z_2\}$ in which z_1 and z_2 may be equal, we branch into $F[x]$ and $F[\bar{x}\bar{z_1}\bar{z_2}]$. The correctness is shown as in the previous case. In the first branch, we directly eliminate eight literals. In the second branch, we eliminate literal x having five occurrences and at least one literal having four or five occurrences.

This gives a branching vector of $(7, 9)$, corresponding to the branching number 1.0910.

By using these branching instructions we obtain for RULE $5'$ the worst case branching vector $(8, 7)$ in terms of formula length, namely for a $(3, 2)[2, 2, 2][2, 2]$-literal x. This corresponds to the branching number 1.0970 and will turn out to be the overall worst case in our analysis of the algorithm.

RULE $6'$**:** Upon reaching this rule, all remaining literals in the formula are either $(2, 2)$, $(3, 2)[2, 2, 2][1, 2]$, or $(2, 3)[1, 2][2, 2, 2]$. RULE $6'$ deals with the case that there is a $(2, 2)[2, 2][1, 2]$-literal x, i.e. a $(2, 2)$-literal having a unit occurrence of $\bar{x}$. As this rule is similar to RULE $5'$, we omit the details here and claim a worst case branching vector of $(5, 12)$ corresponding to the number 1.0908.

RULE $7'$ eliminates all remaining $(3, 2)$-literals, namely those of type $(3, 2)[2, 2, 2][1, 2]$. We select literals y_1, y_2, y_3, and z from clauses $\{x, y_1\}$, $\{x, y_2\}$, $\{x, y_3\}$, and $\{\bar{x}, z\}$. If there is a variable $\tilde{y}$ which equals at least two of the variables $\tilde{y_1}$, $\tilde{y_2}$, $\tilde{y_3}$, and $\tilde{z}$, we branch into subcases $F[y]$ and $F[\bar{y}]$. Otherwise, i.e. all variables $\tilde{y_1}$, $\tilde{y_2}$, $\tilde{y_3}$, and $\tilde{z}$ are distinct, we branch into subcases $F[y_1\bar{x}]$, $F[y_1 x \bar{y_2} \bar{y_3} z]$, and $F[\bar{y_1}]$. The analysis of this rule is omitted here, as it is in large extent analogous to the final RULE $8'$, which we will study in more detail.

RULE $8'$ applies to the $(2, 2)[2, 2][2, 2]$-literals, which are the only literals remaining in the formula. Consider clauses $\{x, y_1\}$, $\{x, y_2\}$, $\{\bar{x}, z_1\}$, and $\{\bar{x}, z_2\}$. In the case where there is a variable $\tilde{y}$ which equals two of the variables $\tilde{y_1}$, $\tilde{y_2}$, $\tilde{z_1}$, or $\tilde{z_2}$, i.e. $\tilde{y}$ has two or more common occurrences with $\tilde{x}$, we branch into $F[y]$ and $F[\bar{y}]$. We can easily see how to obtain a branching vector of $(8, 8)$ and the branching number 1.0906, as setting a value for $\tilde{y}$ implies a value for $\tilde{x}$. Therefore, we proceed to the case of distinct variables $\tilde{y_1}$, $\tilde{y_2}$, $\tilde{z_1}$, and $\tilde{z_2}$.

First, we discuss the correctness of the subcases. The correctness of the subcases $F[y_1\bar{x}]$, $F[y_1 x]$, $F[\bar{y_1} x]$ and $F[\bar{y_1}\bar{x}]$ is obvious. Now assume in the second branch that a partner of $\bar{x}$, e.g. z_1, would be falsified. Then, in comparison to the first branch, we would lose the now falsified clause $\{\bar{x}, z_1\}$, but could, in the best case, gain one additional x-clause. On the other hand, assume that y_2 would be satisfied. Then in the second branch, as compared with the first one, we can not gain any additional x-clause, but could lose some $\bar{x}$-clauses. This shows that in the second branch, we can neglect the considered assignments, as they do not improve the result obtained in the first branch. Analogously, we obtain the additional assignments in the fourth branch and, therefore, branch into subcases

$F[y_1\bar{x}]$, $F[y_1x\bar{y}_2z_1z_2]$, $F[\bar{y_1}x]$, and $F[\bar{y_1}\bar{x}y_2\bar{z}_1\bar{z}_2]$. Knowing that all literals in the formula are $(2,2)[2,2][2,2]$, we obtain the vector $(11,20,11,20)$.

As this vector does not satisfy our purpose, we further observe that in branch $F[y_1\bar{x}]$ and in branch $F[\bar{y_1}x]$, there are undoubtly literals reduced to an occurrence of three or two. These literals are either eliminated due further reduction, or give rise to a RULE 2 branching in the next step. We check that the worst case branching vector in RULE 2 is $(7,10)$. Combining these steps, we are now able to give a worst case branching vector for RULE $8'$ of $(18,21,20,18,21,20)$, corresponding to the branching number 1.0958.

This completes our algorithm and its analysis in terms of formula length. Omitting some details, we have shown a worst case branching number of 1.0970 in all branching subcases, which justifies the claimed time bound.

For MaxSat in terms of the number of clauses, the upper time bound $O(1.3413^K|F|)$ is known [1]. Setting $|F| = 2K$ in Theorem 3, we obtain:

Corollary 4 Max2Sat *can be solved in time* $O(1.2035^K)$.

Using this algorithm we can also solve the Maximum Cut problem, as we can translate instances of the Maximum Cut problem into 2CNF formulas [11]. In fact, these formulas exhibit a special structure and we can modify and even simplify the shown algorithm, in order to obtain better bounds on formulas having this special structure. As shown in [8] on 2CNF formulas generated from Maximum Cut instances, Max2Sat can be solved in time $O(1.0718^{|F|})$ and $O(|F| + 1.2038^k)$, where k is the maximum number of satisfiable clauses in the formula. This implies the bounds for Maximum Cut shown in Theorem 5. Observe for part (2) that Maximum Cut, when restricted to graphs of vertex degree at most three, is *NP*-complete [7].

Theorem 5 *1. For a graph with n vertices and m edges the Maximum Cut problem is solvable in $O(1.3197^m)$ time.*

2. *If the graph has vertex degree at most three, then Maximum Cut can be solved in time $O(1.5160^n)$. If the graph has vertex degree at most four, then Maximum Cut can be solved in time in $O(1.7417^n)$.*
3. *We can compute in time $O(m + n + k \cdot 1.7445^k k)$ whether there is a maximum cut of size k.*

5 Experimental results

Here we indicate the performance of our algorithms A (Section 3) and B (Section 4), and compare them to the two-phase heuristic algorithm for MaxSat presented by Borchers and Furman [3]. The tests were run on a Linux PC with an AMD K6 processor (233 MHz) and 32 MByte of main memory. All experiments are performed on random 2CNF-formulas generated using the MWFF package from Bart Selman [14]. We take different numbers of variables and clauses into consideration and, for each such pair, generate a set of 50 formulas. As results, we give the average for these sets of formulas. If at least one of the formulas in a set takes longer than 48 hours, we do not process the set and indicate this in the table by "not run". Our algorithms are implemented in JAVA. This gives credit to the growing importance of JAVA as a convenient and powerful programming language. Furthermore, our aim is to show how the algorithms limit the exponential growth in running time, being effective independent of the programming language. The algorithm of Borchers and Furman is coded in C. Coding a simple program for Fibonacci recursion in C and JAVA and running it in the given environment, we found the C program to be faster by a factor of about nine. Due to the different performance of the programming languages, it is difficult to only compare running times. As a fair measure of performance we, therefore, also provide the size of the scanned branching tree, as it is responsible for the exponential growth of the running time. More precisely, for the branching tree size we count all inner nodes, where we branch towards at least two subcases.

There is almost no difference in the performance between algorithms A and B; therefore they are not listed separately. This is plausible, as in the processing of random formulas, the "bad" case situations in whose handling our algorithms differ, are rare. On problems of small size, the 2-phase-EPDL (Extended Davis-Putnam-Loveland) algorithm of Borchers and Furman [3] has smaller running times despite its larger branching trees. One reason may also be the difference in performance of JAVA and C. Nevertheless, with a growing problem size our algorithm does a better job in keeping the exponential growth of the branching tree small, which also results in significantly better running times, see Table 1.

In order to gain insight into the performance of our rules, we collected some statistics on the application of the transformation and branching rules. For algorithm B, we examine which rules apply how often during a run on random formulas. First, we consider the transformation rules. Note that at one point, several transformation rules could be applicable

		Algorithm B		2-Phase-EDPL	
n	m	Tree	Time	Tree	Time
25	100	16	0.77	961	0.27
	200	108	1.78	37 092	1.93
	400	385	5.41	514 231	43.72
	800	752	12.59	2 498 559	9:16.51
50	100	6	0.70	69	0.48
	200	320	4.48	611 258	27.11
	400	18 411	3:45.80	not run	–
100	200	36	1.14	10 872	2.14
	400	91 039	23:50.09	not run	–
200	400	1 269	21.87	not run	–

Table 1. Comparison of average branching tree sizes (Tree) and average running times (Time), given in minutes:seconds, of our Algorithm B and the 2-phase-EDPL by Borchers and Furman. Tests are performed on 2CNF formulas with different numbers of variables (n) and clauses (m).

to a formula. Therefore, for judging the results, it is important to know the sequence in which the application of transformation rules is tested. We show the results in Table 2, with the rules being in the order in which they are applied. Considering the shown and additional data, we

Variables	25				50		
Clauses	100	200	400	800	100	200	400
Search Tree Size	16	108	385	752	6	320	18 411
Almost Common Cl.	10	38	102	235	4	76	1 421
Pure Literals	26	82	111	99	34	658	11 476
Dominating 1-Clause	123	704	1 844	2 839	42	3387	134 425
Complementary 1-Clause	40	571	2 831	6 775	4	1173	128 030
Resolution	22	57	54	30	25	726	10 821
Three Occurrences 1	0	1	4	7	0	1	35
Three Occurrences 2	6	25	51	47	2	86	2 810

Table 2. Statistics about the application of transformation rules in algorithm B on random formulas.

find application profiles being characteristic for variable/clause ratios. We observe for formulas having a higher ratio, i.e. with fewer clauses for a fixed number of variables, that the Dominating 1-Clause Rule is the rule which is applied most often. With lower ratio, i.e. when we have more clauses for the same number of variables, the Complementary 1-Clause Rule gains in importance.

Variables	25				50		
Clauses	100	200	400	800	100	200	400
Tree Size	15.26	107.4	384.4	751.18	5.26	319.36	18 410.38
RULE 1	10.62	102.32	381.76	749.42	0.88	217.36	18 300.02
RULE 2	4.02	3.54	1.18	0.32	4.34	100.12	97.54
RULE 3	0.5	0.9	0.94	0.64	0	1.44	11.14
RULE 4	0.08	0.44	0.26	0.36	0.04	0.34	1.22
RULE 5′	0.04	0.12	0.16	0.26	0	0.08	0.3
RULE 6′	0	0.06	0.1	0.16	0	0.02	0.16
RULE 7′	0	0.02	0	0	0	0	0
RULE 8′	0	0	0	0.02	0	0	0

Table 3. Statistics on the application of branching rules in algorithm B on random formulas having n variables and m clauses. Recall that each result is the average on 50 formulas to understand that we give non-integer values. Thereby we even see the application of very rare rules.

Besides the transformation rules, we also study the frequency in which the single branching rules are applied. Recall that algorithm B has a list of eight different cases with corresponding branching rules. We show the results collected during runs on random formulas in Table 3. We observe that the most branching steps occur with RULE 1 or RULE 2. The other rules are used in less than one percent of the branchings. It is reasonable that in formulas with a high variable/clause ratio, i.e. fewer clauses, we have more variables with an occurrence of three. Therefore, the rule applied most while processing these formulas is RULE 2. As the variable/clause ratio shifts down, i.e. when we have more clauses for the same number of variables, there necessarily are more variables with a large number of occurrence in the formula. Consequently, RULE 1 becomes dominating.

Considering our statistics, we can roughly conclude: Some of the transformation rules are, in great part, responsible for the good practical performance of our algorithms, as they help to decrease the search tree size. The less frequent transformation rules and the rather complex set of branching rules, on the other hand, are mainly important for guaranteeing good theoretical upper bounds.

6 Open questions

There remains the option of investigating exact algorithms for other versions of Max2Sat, for example, Max3Sat. Furthermore, n being the number of variables, can Max2Sat be solved in less than 2^n steps? Re-

garding Hirsch's recent theoretical results [10], it seems a promising idea to combine our algorithm with his, in order to improve the upper bounds for Max2Sat even further.

References

1. N. Bansal and V. Raman. Upper bounds for MaxSat: Further improved. In *Proceedings of the 10th International Symposium on Algorithms and Computation*, Lecture Notes in Computer Science, Chennai, India, Dec. 1999. Springer-Verlag.
2. R. Battiti and M. Protasi. Approximate algorithms and heuristics for MAX-SAT. In D.-Z. Du and P. M. Pardalos, editors, *Handbook of Combinatorial Optimization*, volume 1, pages 77–148. Kluwer Academic Publishers, 1998.
3. B. Borchers and J. Furman. A two-phase exact algorithm for MAX-SAT and weighted MAX-SAT problems. *Journal of Combinatorial Optimization*, 2(4):299–306, 1999.
4. J. Cheriyan, W. H. Cunningham, L. Tunçel, and Y. Wang. A linear programming and rounding approach to Max 2-Sat. *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, 26:395–414, 1996.
5. R. G. Downey and M. R. Fellows. *Parameterized Complexity.* Springer-Verlag, 1999.
6. U. Feige and M. X. Goemans. Approximating the value of two prover proof systems, with applications to MAX 2SAT and MAX DICUT. In *3d IEEE Israel Symposium on the Theory of Computing and Systems*, pages 182–189, 1995.
7. M. Garey and D. Johnson. *Computers and Intractability: A Guide to the Theory of NP-completeness.* Freeman, San Francisco, 1979.
8. J. Gramm. Exact algorithms for Max2Sat: and their applications. Diplomarbeit, Universität Tübingen, 1999. Available through http://www-fs.informatik.uni-tuebingen.de/~niedermr/publications/index.html.
9. J. Håstad. Some optimal inapproximability results. In *Proceedings of the 29th ACM Symposium on Theory of Computing*, pages 1–10, 1997.
10. E. A. Hirsch. A new algorithm for MAX-2-SAT. Technical Report TR99-036, ECCC Trier, 1999. To appear at *STACS 2000*.
11. M. Mahajan and V. Raman. Parameterizing above guaranteed values: MaxSat and MaxCut. *Journal of Algorithms*, 31:335–354, 1999.
12. R. Niedermeier and P. Rossmanith. New upper bounds for MaxSat. In *Proceedings of the 26th International Conference on Automata, Languages, and Programming*, number 1644 in Lecture Notes in Computer Science, pages 575–584. Springer-Verlag, July 1999. Long version to appear in *Journal of Algorithms*.
13. V. Raman, B. Ravikumar, and S. S. Rao. A simplified NP-complete MAXSAT problem. *Information Processing Letters*, 65:1–6, 1998.
14. B. Selman. MWFF: Program for generating random Max$k$-Sat instances. Available from DIMACS, 1992.
15. M. Yannakakis. On the approximation of maximum satisfiability. *Journal of Algorithms*, 17:475–502, 1994.