Algorithms and Experiments on Finding Minimal Models

Paolo Liberatore
Department of Computer and System Sciences
University of Rome "La Sapienza"
Via Salaria 113, I-00198 Rome Italy
Email: `liberato@dis.uniroma1.it`

December 29, 2000

Abstract

In this paper we consider the problem of finding *minimal models* of a set of propositional clauses. This problem has many applications: for instance, finding minimal-length plans, minimal diagnosis, and solving many other minimization and maximization problems. We have adapted the Davis-Putnam algorithm (that finds generic models of a set of clauses) in order to find minimal models. Experiments have been performed to determine which heuristics can be useful, and to decide in which cases the problem is hard and in which cases it is easy. A comparison with a local search algorithm is also shown.

1 Introduction

The problem of finding models of logical formulas (SAT) has a long history. Recent algorithms such as GSAT [SLM92] and WALKSAT [SKC94], as well as optimizations and variants of the old Davis-Putnam algorithm [DLL62, CA93, LA97] makes it feasible to solve the problem of propositional satisfiability even for large sets of clauses.

In some cases, this allows for solving problems of areas different from logics. It has been proved, for instance, that plans can be found efficiently by translating the problem into the problem of finding models of sets of propositional clauses. Planning systems based on this translation exist [EMW97, KS98].

This approach, while successful in some cases, has theoretical limitations. Indeed, many relevant problems cannot be solved by translating them to the problem of satisfiability. The aim of our work is to build a system for solving a problem more expressive than satisfiability, that is, a problem to which we can translate some relevant problems that cannot be translated to SAT.

The problem we consider in this paper is: given a set of propositional clauses, find a model with the least possible number of variables assigned to true. This problem is of theoretical interest, as it is "prototypical" of a large class of optimization problems. It is also relevant in practice, for at least two reasons.

1. Many optimization problems cannot be efficiently translated to SAT. Indeed, only problems in the complexity class NP can be translated to SAT in polynomial time [Joh90], and there are many optimization problems that are not in NP. For instance, finding a minimal-length plan in the STRIPS formalism is not in NP, even in the assumption of polynomially bounded plans (on the contrary, finding non-minimal plans is in NP). Many other optimization problems cannot be translated to SAT in polynomial time, while they can be translated to the problem of minimal models. The same holds for some problems of reasoning, such as Default Logics and Belief Revision.

2. Some optimization problems can be translated to SAT (for example, planning in the blocksworld). However, a pure SAT procedure will in general find either an optimal solution for the given problem, or no solution at all. This means that, before the exact optimal solution is found, no approximate solution is available. Since the process of solving those problems is usually very long, an anytime algorithm would be useful. Note that an anytime algorithm can be developed only if there is a distinction between valid solutions and optimal solutions (what we want are non-optimal but valid solutions found during the search). The structure of the SAT problem itself does not allow such a distinction.

A solver for the problem of minimal models is useful in first cases, since many optimization problems that are not in NP are in $\Delta_2^p[\log n]$, and such problems can be translated into the problem of finding minimal models of a set of propositional clauses.

Our algorithms for finding minimal models are also useful regarding the second point. Indeed, the algorithms are based on the idea of finding a model of the formula, and then trying to find models with less true variables, then another one with still less true variables and so on, until some termination condition is met. This means that, during the search, some approximate solutions may be available.

Consider for instance the problem of finding minimal plans in the blocksworld. We can translate the instance into a propositional formula, in such a way that:

1. models correspond to possible plans;

2. models with less true variables correspond to shorter plans.

Since our algorithms start with a model, and then finds better and better models, we have an initial (non-optimal) plan, and then a sequence of shorter and shorter plans. This means that we can stop the procedure at any time and, if at least one model was found, this is actually a plan (maybe non-minimal)[1]. Such a correspondence between models with less true variables and better solutions can also be established for many other optimization problems (for instance, minimal graph coloring, minimal abductive diagnosis, etc.)

In this paper we have considered the problem of finding minimal models only, that is, we did not try (yet) to solve real-world problems (planning, diagnosis, graph coloring) with our programs.

2 Preliminaries

In this section we define the problems we tried to solve, and give some theoretical considerations about them. The problem of finding minimal models can be formally defined as follows.

Definition 1 *The problem of cardinality-minimal models is the problem of finding, given a set of propositional clauses F, a model M of F such that there is no other model N of F whose number of variables assigned* `true` *is less than that of M.*

Of course, an inconsistent set of clauses has no minimal models. The above problem can be of theoretical interest. However, it is too simple to encode real-world problem into it in a simple way. For instance, if we translate a problem of planning into a propositional formula, minimal plans corresponds to models in which a *subset* of variables assigned `true` is minimal. While it is in theory possible to translate such a problem to that of finding cardinality-minimal models, this translation implies an increase in the number of clauses. For this reason, we extend the above definition assigning *weights* to variables.

Definition 2 *Let $\{x_1, \ldots, x_n\}$ be the set of the propositional variables. A set of weights associated to them is a set of nonnegative integers $\{w_1, \ldots, w_n\}$, each w_i being associated with x_i. The weight of a model M is:*

$$W(M) = \sum \{ \; w_i \mid x_i \textit{ is } \mathtt{true} \textit{ in } M \; \}$$

The problem we consider is that of finding models whose weight is minimum.

Definition 3 *The problem of weight-minimal models consists in finding, given a set of propositional clauses F over the variables $\{x_1, \ldots, x_n\}$ and a set of weights $w_1, \ldots, w_n$ associated to the variables, the model M of F having a minimum value of $W(M)$.*

It is easy to see that the latter definition is an extension of the former, as cardinality-minimal models are those with minimum weight, if $w_1 = \cdots = w_n = 1$.

Let us give an example of the above definitions.

[1]Algorithms like GSAT and WALKSAT actually find some partial models during the search. However, such models do not correspond to any valid plan.

Example 1 *Let $F = \{x_1 \vee x_2,\ x_1 \vee x_3\}$. This set has five models. To denote models we use the set of variables assigned to* true*. The models of F are thus:*

$$\begin{aligned} M_1 &= \{x_1\} \\ M_2 &= \{x_2, x_3\} \\ M_3 &= \{x_1, x_2\} \\ M_4 &= \{x_1, x_3\} \\ M_5 &= \{x_1, x_2, x_3\} \end{aligned}$$

In words, M_1 is the model that assigns true *to the variable x_1 and* false *to the others, M_2 assigns* true *to x_2 and x_3 and* false *to x_1, etc.*

The set of clauses F has only one cardinality-minimal model, M_1. Indeed, M_1 assigns true *to only one variable, while M_2, M_3 and M_4 assign* true *to two variables, and M_5 have three variables assigned* true*.*

Assuming $w_1 = w_2 = 1$ and $w_3 = 0$, the set of clauses F has three models with minimum weight, namely M_1, M_2, and M_4. There is exactly one set of weights such that M_5 has minimum weight. The solution of this little quiz is in the last section of this paper.

Let us give some theoretical results about these problems. We begin with the computational complexity of finding minimal models. Details about the complexity classes $\Delta_2^p[\log n]$ and Δ_2^p can be found in any textbook on computational complexity [Joh90, Pap94]. Short summary: $\Delta_2^p[\log n]$ is the complexity class that contains problems that can be solved using a polynomial-time algorithm that makes a logarithmic number of calls to a procedure that solves an NP problem, while Δ_2^p is the class of problems requiring a polynomial number of calls to the procedure for the NP problem.

Determining the complexity of a finding problem (such as the minimal model problem) is usually quite involved. Let us consider the subproblem of determining the minimum cardinality/weight of models of a set of clauses.

Theorem 1 *Determining the minimum cardinality of models of a set of clauses is a $\Delta_2^p[\log n]$-complete problem.*

Proof. Easy consequence of Theorem 13 of [LS97]. □

Note that, once known the minimum cardinality of models, determining a minimal model requires only one call to a procedure for solving a problem in NP. Thus, the hard part of the minimal model finding problem is that of determining the number of true variables in the minimal models.

The complexity of the problem of minimum-weight models is not so simple to determine. Indeed, it depends on whether weights are bounded or not. We report the proof of the following theorem, but it may have been appeared already in the literature (anyway, it is quite straightforward).

Theorem 2 *Determining the minimum weight of models of a set of clauses is:*

1. *Δ_2^p-complete if the weights can be arbitrarily large;*
2. *$\Delta_2^p[\log n]$-complete if the weights are bounded by a linear function of the number of variables.*

Proof. In the case of bounded weights, we can use the usual binary search technique: since deciding whether there is a model of weight less or equal than a number c is a problem in NP, we can use a logarithmic number of calls to a procedure for solving it to determine the minimum weight of models. Hardness immediately follows from the fact that determining the minimum cardinality of models is a subcase of finding the minimum weight.

If weights are not bounded, the binary search technique can be used as well. However, the number of calls to the procedure is not necessarily logarithmic. Polynomiality can be ensured by normalizing weights: assuming for example that $w_1 \leq w_2 \leq \cdots \leq w_n$, if for some j it holds

$$w_j > \sum_{i=1}^{j-1} w_i + 1$$

then we set w_j to the right-hand side of the above equation. Using the binary search technique, the number of calls to the procedure for the NP problem is polynomial.

Hardness is an immediate consequence of the fact that the problem of the lexicographic-minimal model (which is Δ_2^p complete [Kre88]) can be reduced to the problem of minimum weight: just use the weights $w_i = 2^{n-i}$: there is exactly one model of minimum weight (if any), and it is identical to the lexicographic minimal model. □

The above theorem shows that the complexity of the problem of minimum weight depends whether weights are bounded or not. This is not the pure theoretical distinction it may appear. For instance, if we want to use a program for finding models of minimum weight to determine the solution of a Δ_2^p complete problem (for instance, iterated belief revision [Lib97]), we need both very large and very small numbers, so we cannot use any fixed-size representation of numbers (such as `int`, `float`, etc.) Since we actually used `int` numbers in our programs, we are implicitly assuming a bound on the value of the weights.

The above two theorems proves the usefulness of solving the problem of finding minimal models. Indeed, any problem in $\Delta_2^p[\log n]$ can be polynomially reduced to the problem of determining the minimum cardinality/weight of models, which is an obvious subcase of finding a minimal model. This class includes problems of reasoning, such as Belief Revision [EG92], and Default Logics [BG99].

The problem of finding the minimum length of plans in the STRIPS formalism can also be proved to be in the complexity class $\Delta_2^p[\log n]$, under suitable hypothesis. Indeed, as shown for instance in [KS96], if we assume to be interested in polynomial-length plans, planning is in NP. Under the same assumption, finding the minimum length of plans is in $\Delta_2^p[\log n]$, and thus it can be reduced in polynomial time to the problem of minimal models. Moreover, we can reduce the problem of *finding* a minimum-length plan to the problem of minimal models (under the same assumption): we can for instance use the translation based on the idea of representing each operator and each condition in each time point as a propositional variable, and then writing a set of clauses expressing that the sequence of operators leads to the goal. Setting the weight of variables representing operators to 1, and all other ones to 0, this is equivalent to finding plans with the least possible number of operators.

3 The Algorithm

The algorithm we use for finding minimal-weight models is a variant of the algorithm for satisfiability developed by Davis and Putnam [DP60] and improved by Davis, Logemann, and Loveland [DLL62]. This is often known as the Davis-Putnam algorithm, or the Davis-Putnam-Logemann-Loveland algorithm (DPLL). We give an abstract description of the DPLL algorithm, and show how it can be used to find minimal models. We present two heuristics that are useful for finding minimal models. The results of experiments are reported in the next section.

3.1 The DPLL Algorithm (Davis-Putnam-Logemann-Loveland)

The DPLL algorithm has been proposed in the 60s to find arbitrary (i.e. not necessarily minimal) models of a set of clauses. The algorithm is basically a backtracking algorithm enhanced with some rules to reduce the search. It can be summarized as follows.

```
/*
Backtracking algorithm for satisfiability (core of DPLL).
M+{x=value} is the model obtained from M by setting the value of x.
*/

model find_model( set_of_clauses F, partial_model M ) {
  if all clauses are satisfied by M
    return M;
  else if all literals of a clause are false in M
    return unsatisfiable;
  else {
    choose variable x not assigned by M;
    if( find_model( F, M + { x=false } ) != unsatisfiable )
      return find_model( F, M + { x=false } );
    else if( find_model( F, M + { x=true } ) != unsatisfiable )
      return find_model( F, M + { x=true } );
    else
      return unsatisfiable;
  }
}
```

We start the algorithm with the partial model in which all variables are unassigned. At each step we check whether all clauses are satisfied, or there is a clause in which all literals are false. In these cases, we can conclude that the partial model satisfies the set of clauses, or falsifies it, respectively. If none of the above is true, we choose a variable that is not assigned a value in M, and try with the model obtained by setting this variable to false, and then with the one obtained by setting it to true. The set is satisfiable if either model is consistent with the set of clauses.

A concept that often clarifies how the algorithm works is that of *search tree*. Assume that we choose the variables to assign value in a lexicographic order, that is, at the first recursive call we choose x_1, then x_2 etc. The first call to the procedure calls itself recursively twice, assigning x_1 to true the first time and to false the second time. In each call:

$x_1 =$ false when the procedure is called with $x_1 =$ false, two other calls are made, assigning x_2 to false and true, respectively;

$x_1 =$ true the same two calls are made.

The structure of recursive calls can be seen as a tree. Each node represents a call of the procedure. In each call, an unassigned variable is chosen, and this is indicated as the label of the node. In each node, two recursive calls are made assigning this variable to false and true, and these are represented as the children of the node. The first three levels of a search tree are shown in Figure 1.

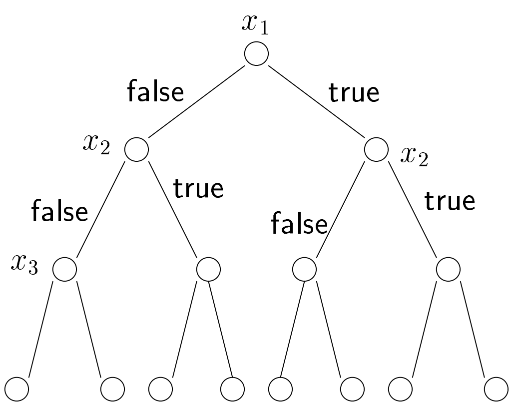

Figure 1: The first three levels of a search tree.

The DPLL algorithm consist, in principle, in a visit of this tree. Part of the tree, however, is not visited due to contradictions found during the search. For instance, if the procedure is in a node in which $x_1 =$ false and $x_2 =$ false, and there is a clause falsified by this assignment (for instance, $x_1 \vee x_2$), there is no need to visit the children of the node.

The original DPLL algorithm uses three rules to improve efficiency.

Unit Propagation: Whenever all literals of a clause but one are falsified by the partial model, we set the value of the unassigned one in such a way the clause is satisfied. For instance, if F contains a clause $x_1 \vee x_4 \vee x_8$, and both x_1 and x_4 are assigned false, we set x_8 to true. This function is called "unit propagation" because clauses in which only one literals is unassigned (and all others are false) can be seen as unit clauses, that is, clauses containing only one literal. It can be proved that unit propagation can be performed in linear time, by using a suitable index of the clauses.

Monotone Literals: If, neglecting the satisfied clauses, a literal appears while its negation does not, we can set the value of the literal to true.

Clause Subsumption: If the unassigned literals of an unsatisfied clause are a subset of the unassigned literals of another unsatisfied clause, we can neglect the second one. This rule is not used in modern implementations of the DPLL algorithm, and it is also not implemented in our programs.

What is described so far is the basic DPLL algorithm. Implementations differ from it in several aspects:

1. many optimizations can be done: for example, the procedure may be implemented in an iterative manner (instead of recursive), an index of the clauses can be used to implement a linear-time unit propagation function, etc;
2. choosing variables to assign in a lexicographic way has been proved to be not efficient; several other choices are possible (different implementations of the DPLL algorithm differ for the way this choice is made).

In our implementation, we used the linear-time unit propagation function, and the choice of unassigned variables is the same as in [CA93]: we use the binary clauses to select a set of candidates, and then we select the variable that produces the maximum number of literals by unit propagation. We do not describe the implementation of DPLL in details, but rather how it can be modified to find minimal models.

3.2 DPLL for Minimal Model Finding

The core of our algorithm is a function that finds a model with weight *bounded* by some number. For instance, this function may be called to find a model with weight less or equal than 10. The first version of this algorithm is very similar to the DPLL algorithm. The only difference is that whenever the weight of the partial model is greater then the bound, we act as if the model falsified the set of clauses. This way, models with weight greater than the bound are never accepted, even if they satisfy all clauses.

```
/*
Variant of Davis-Putnam for finding models of weight bounded by a constant.
```

```
*/

model find_model_bounded( set_of_clauses F, partial_model M, int bound ) {
  if weight of M > bound
    return not_exists;
  else if all clauses are satisfied by M
    return M;
  else if all literals of a clause are false in M
    return not_exists;
  else {
    choose unassigned variable x;
    if( find_model_bounded( F, M + { x=false } ) != not_exists )
      return find_model_bounded( F, M + { x=false } );
    else if( find_model_bounded( F, M + { x=true } ) != not_exists )
      return find_model_bounded( F, M + { x=true } );
    else
      return not_exists;
  }
}
```

The only difference is that the weight of the partial model is evaluated before doing any other step. The weight of the partial model is the sum of the weight of variables assigned true, that is, we do not consider the weight of unassigned variables and of variables assigned false.

It is easy to see that the weight of a partial model M is less than or equal to the weight of any model obtained by assigning a value to the unassigned variables of M. In other words, if the search is in a node whose corresponding partial model M has weight w, then all its successor nodes have weight greater than or equal to w. As a result, if w is greater than the bound, any successor node has weight greater than w, thus there is no need to visit the subtree of the successor nodes, since any model it contains has weight greater than the bound.

Let us consider the three rules used in DPLL for improving efficiency. The unit propagation rules can be safely applied. The clause subsumption rule can also be used, but (as for DPLL) it is not useful in practice, so we did not implement it. The monotone literal rule requires some care. For example, if the instance is $\{x_1 \vee x_2, \neg x_2 \vee \neg x_3, x_2 \vee x_3\}$, and the weights are $\{1, 0, 0\}$, following the monotone literal rule we could conclude that, since x_1 only appears positive, we can set it to true. This leads to models of weight 1. However, the interpretation $\{x_2\}$ is a model of the above set of clauses, and its weight is 0. Clearly, the first model is not minimal. As a result, the monotone literal rule must be modified as follows.

Modified Monotone Literal Rule: If a literals l appears in a currently not satisfied clause, while its negation $\neg l$ does not, then l can be set to `true` only if it is either negative ($l = \neg x_i$), or it is positive ($l = x_i$) and its weight is 0.

In addition of the unit propagation and the (modified) monotone literal rules, two heuristics are useful for the specific case of minimal model finding. The first uses the positive clauses (clauses whose unassigned literals are positive) to find a lower bound on the weight of models. The second uses the weight of partial models to infer the truth value of variables. In the next section we show the result of an experiment on the efficiency improvement obtained by using these heuristics.

3.3 Heuristics "Positive"

This heuristics is best shown with an example.

Example 2 *Suppose we are looking for a model of weight less or equal than 5, our partial model M has weight 5, and all weights are equal to 1. Among the clauses still not satisfied (that is, of which no literal is assigned to true), we have the following one: $x_2 \vee x_4$, where neither x_2 not x_4 is currently assigned. Does it makes sense to try to continue the search?*

Answer: no. In order to satisfy the clause, we need to assign `true` *to at least one variable between x_2 and x_4, which means that any model extending M has weight at least equal to 6.*

Given a partial model M, we use the clauses whose unassigned literals are positive to obtain a lower bound of the weight of models extending M. In other words, we select all clauses such that:

1. they contain no satisfied literals; and
2. all their unassigned literals are positive.

Since false literals of clauses can be neglected, this is actually a set of clauses containing only positive literals. For instance, if $x_3 =$ `true`, then $x_1 \vee \neg x_3 \vee x_5$ is equivalent to $x_1 \vee x_5$. Finding a model for a set of positive clauses is straightforward (just assign `true` to all variables). However, we are looking for a lower bound, that is, we want a number that is always *lower* than the weight of any model of this set.

Obtaining an exact bound is in general an NP-complete problem. We use an approximation, which is essentially the greedy algorithm for the set cover problem. Indeed, we want to select a number of variables to assign `true`, in such a way any clause contains at least a variable assigned to `true`, and the weight of the selected variables is minimum.

The heuristics works as follows. For each variable, we count the number of clauses in which it appears. Let o_i be the number of clauses in which x_i appears. Then we consider, for each clause, the minimum value of w_i/o_i.

int *lower_bound*(*set_of_positive_clauses* C) {
  for each *variable* x_i

o_i = *number of clauses in which* x_i *appears;*

$lb = 0$;

for each *clause* c_i {
  i=*index of the variable of* c_i *such that* w_i/o_i *is minimum;*
  $lb = lb + w_i/o_i$;
}

return *lb;*
}

This greedy algorithm returns a weight, which is always less than or equal to the minimum weight of models satisfying the clauses. This lower bound can be used to reduce the part of the search tree visited by the procedure for finding models with a bound on the weight.

The algorithm `find_model_bounded` is modified as follows. At each step, we build a set of positive clauses by taking the unassigned literals from the clauses whose unassigned literals are positive and no literal is satisfied by the current partial model. Then, we run the above algorithm that finds a lower bound on the weight of the variables that are needed to satisfy all positive clauses. If the weight of the partial model plus the lower bound is greater then the bound on models, we return `not_exists`, that is, we do not try assigning a value to other variables.

The algorithm above is not optimized. Its efficiency can be greatly improved by building incrementally an appropriate data structure for storing information about clauses in which all unassigned literals are positive and no literal is true. Moreover, clauses in which an unassigned literal has weight 0 can be neglected.

3.4 Heuristics "Over"

The second heuristics is based on the principle of setting to false the value of variables whenever their weight is too great w.r.t. the bound.

Example 3 *Let us assume that we are looking for a model of weight less or equal then 10, that our partial model M has weight 9, and x_2 is still not set and has weight 2. It is clear that we can set x_2 to* false*, since setting it to* true *will result in a weight of 11.*

Experiments show that this heuristics is useful when most of the variables have weight equal to 0. Consider what happens if x_2, in the above example, is not set to false. Since most of the variables have weight equal to 0, this means that the next variable chosen by the `find_model_bounded` procedure is likely to have weight 0. As a result, two nodes are visited. At this point, another variable is chosen. If it still has weight 0, other 4 nodes are visited. Assume that x_2 is assigned at this point. This means that 8 more nodes are visited, and in four of them x_2 is assigned to true, generating a contradiction.

Suppose, on the other hand, that x_2 is immediately set to false. It is easy to see that the number of visited node is reduced from 1+2+4+8=15 to 1+2+4=7. The improvement can be even better if making unit propagation on $x_2 =$ false generates contradiction or a model whose weight is greater than the bound: in this case the number of nodes is reduced from 15 to 1.

3.5 Minimal Models

The algorithm described above finds a model with weight less than or equal to a given number. Models with minimum weight can be found using this algorithm. We defined (and implemented) two algorithms. We begin describing the iterative search algorithm, since it is the easier to explain. It essentially a branch-and-bound algorithm, enhanced with the two DPLL heuristics, and possibly the two heuristics we developed.

```
/*
 Minimal model search: iterative algorithm.
*/

minimum_weight( set_of_clauses C ) {
  k=sum of the weight of all variables;
  klast=unsatisfiable;
  while( true ) {
    find model M with weight less or equal than k;
    if( such model does not exists )
      return klast;
    else {
      klast=weight of M;
      k=klast-1;
    }
  }
}
```

In words, we start searching for any model. If it does not exists, then the set of clauses is unsatisfiable. Otherwise, we store the weight of the model in the variable `klast`, and continue the search with `k=klast-1`, that is, we look for a model whose weight is less than that of the last model found.

The above algorithm can be easily modified to find a minimal model instead of returning its weight only. The fact that the `find_model_bounded` returns a partial model instead of a model is not a problem: by assigning any unassigned variable to false, we obtain a model of the set of clauses having the same weight of the partial model.

Note that each time we find a model, we do not need to restart the search from the beginning. In other words, we do not have to start the search from the root of the search tree. Instead, we can continue the search starting from the last node visited.

How many calls to the procedure that finds a model of bounded weight are performed? Consider the case in which all variables have weight equal to 1. If the number of variables is n, it may be that the first model found has weight exactly equal to n, the second has weight $n-1$, and so on. Conclusion: $n+1$ calls to the find procedure may be needed.

It is possible to reduce this number, using the binary search technique. This is done as follows: if w_t is the sum of the weight of all variables, we start by looking for a model whose weight is less than or equal to $w_t/2$. If it exists a model with weight $w < w_t/2$, we look for a model with weight less than or equal to $w/2$. If it does not exists, we search for a model with weight less than or equal to $w_t \cdot 3/4$. In the case in which all weights are 1, the number of calls to the search procedure is logarithmic in the number of variables.

The actual implementation of the binary search algorithm does not restart the search from the beginning at each step. Rather, it saves the complete state of the search whenever a model is found. Then, the search is continued from this point with a new value of k. If this search succeed, the new state of the search is saved (and the old one can be discarded). Otherwise, the search is restarted from the point in which we found the last model, using a new bound. This algorithm can be described as follows.

```
/*
  Minimal model search: binary search algorithm.
state is the state of the search
last_state is the state of the search at the last model found
*/

minimum_weight( set_of_clauses C ) {
  k=sum of the weight of all variables;
  find model M with weight less or equal than k;
  if( such model does not exist )
    return unsatisfiable;
  kmax=weight(M) - 1;
  kmin=0;
  k=(kmax-kmin)/2;
  last_state=state;
  while( true ) {
    if( kmax<0 )
      return 0;
    find model M with weight less or equal than k;
    if( such model exists ) {
      last_state=state;
      kmax=weight(M)-1;
```

```
      k=(kmax-kmin)/2;
    }
    else
      if( kmax==kmin )
        return kmax+1;
      else {
        kmin=k+1;
        k=(kmax-kmin)/2;
        state=last_state;
      }
  }
}
```

Counting the number of calls to the procedure for finding models, it may seem that binary search is better than iterative search, since it calls the procedure for finding models less times. Experiments done show that this does not necessarily produce a gain in performance.

3.6 A Note about the "Trivial" Algorithm

The proof that the minimum cardinality/weight problem is in $\Delta_2^p[\log n]$ shows an algorithm for solving the minimal model problem: indeed, the solution can be found with a logarithmic calls to a procedure for solving an NP-complete problem. The search for minimal plans in SAT-based planning systems is currently made in this way [EMW97]: minimal plans are found by solving a logarithmic number of SAT instances. Here we discuss the drawbacks of this approach.

First, in general the NP-complete problem to solve is not exactly SAT. Indeed, it is the problem of deciding whether a set of clauses has a model of weight less than or equal to a number c. In order to encode this problem into SAT, we need a large number of additional clauses. On the other hand, using a specific function for determining models whose weight is bounded, a simple loop can be used to determine the weight of the current partial model (actually, the weight of a partial model can be determined incrementally, so we do not even need to look at the whole model to determine its weight). Moreover, two heuristics can be used to reduce the part of the search tree to visit.

Second (and more important point), making independent calls to a complete solver for NP-complete problems implies that the part of the search tree visited in previous calls may be visited again. For example, if in the first call we conclude that x_1 is positive in all models of the instance, there is no need to branch on x_1 during the second call (which is done if the calls to the SAT procedure are independent.) On the contrary, both the iterative and the binary search algorithms use this kind of information.

4 Experiments

The experiments have been made using randomly generated clauses of three literals. This is customary in testing the efficiency of SAT solvers, but the programs we implemented can be used on sets of clauses of arbitrary length. Each literal is randomly chosen in uniform way among the set of all literals.

An instance of the minimal models problem is composed of a set of clauses and a set of weights. While sets of clauses can be generated in the standard way, there is no "standard" way to generate a set of weights. In order to determine how the running time changes depending on the weights, we used three different methods of generation:

1. all variables have weight equal to 1;
2. each variable has weight 1 with probability p and 0 with probability $1-p$.
3. each variable have weight 0.

In the second case, we have also to specify the probability p. This should give some information on how the complexity of the problem changes according to the number of variables to be minimized. Indeed, variables whose weight is 0 are "not to be minimized", that is, the weight of a model is independent from their truth value. In other words, we are looking for models in which the number of variables weighted 1 is minimum. When a set of clauses is obtained for instance from a translation from a planning problem instance, most of the variables have weight 0 (the variables encoding the operators of the planning problem are the only ones weighted 1).

We also decided to use a set of instances in which the weight are not only in the set $\{0,1\}$. In this experiment, we set the weight of each variable randomly in the set $\{0,\ldots,10\}$. There was no particular reason to choose this distribution, other than the fact that weights are easy to generate.

There are other choices to make, and these are about the algorithm to use. First, we have to decide whether to test binary or iterative search. We did both. In the last section, two heuristics have been proposed to improve the efficiency of the search: where not otherwise specified, experiments have been conducted using both heuristics.

All experiments have been made on a Pentium 100MHz running Linux. The results are reported in diagrams having time and number of clauses in the axis (the number of variables and the percentage of variables to be minimized are constant). Time is given as number of clock ticks, as reported by the function `times()` as the user time. Each clock tick corresponds, in our system, to 1/100 of second. Each point referring to c clauses is the average of 200 experiments made using a number of clauses ranging from $c-10$ to $c+9$.

4.1 Iterative vs. Binary Search

The aim of this experiment is to compare iterative search with binary search. All weights are set to 1, that is, we look for cardinality-minimal models. The result of this comparison is in Figure 2.

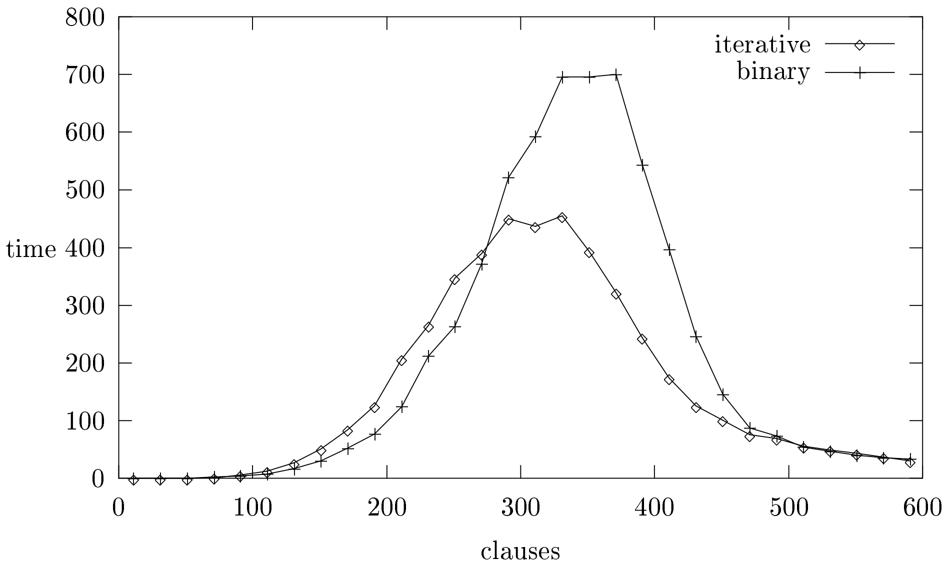

Figure 2: Iterative vs. Binary Search. Quite surprisingly, the running time of binary search is in most cases greater than that of linear search.

The easy-hard-easy pattern, which is a well-known phenomenon in satisfiability [MSL92], is also present in the diagram of both the iterative and the binary search running time: the running time is low when there are few clauses, then it increases with the number of clauses, reaches a maximum, and then it start decreasing when the number of clauses is further increased. In satisfiability, the peak is reached when the number of clauses is 4.2 times the number of variables. In the case of minimal models, the peak is reached for a lesser number of clauses, and differ slightly between iterative and binary search.

The most surprising result of this work is probably the outcome of the comparison between iterative and binary search: iterative search is almost always better than binary search. The difference is not very great; however, theoretical considerations indicate that binary search should be *much better* than linear search. Our explanation is that this phenomenon is due to two causes:

1. complete algorithms (like DP and our variant) are usually slower on unsatisfiable instances than they are on satisfiable ones, since in the former case all search tree have to be explored, while in the latter the tree is visited up to the first model;

2. the problem of finding models without bound is usually much simpler than finding models with a given bound. Namely, the complexity of finding models increases as we approach the optimum. The slowest phase of the iterative algorithm is the last one, that is, the search for a model with weight less than the optimum (this will result, of course, in a failure).

In case of iterative search, the only call to the `find_model_bounded` procedure that fails is the last one. On the converse, binary search does less calls to the find function, but each of them takes more. First, the `find_model_bounded` procedure may fail more than once. Moreover, the way we approach the optimum is not linear. Namely, we have several calls to the function with a bound close to the optimum. For these reason, binary search is slower even if the number of calls to the function for finding models is less.

Another possible explanation of the difference is the way the search tree is visited. In the iterative search, once a model is found in a node of the tree, we can continue the search from that node. If we use binary search, and we find a model with weight w, the next model searched is one having weight less than or equal to $w/2$. If this search fails, we have to restart from the node corresponding to the last model found. As a result, nodes visited in the last call to `find_model_bounded` may be visited again. On the other hand, iterative search never visit a node more than once.

4.2 Number of variables to be minimized

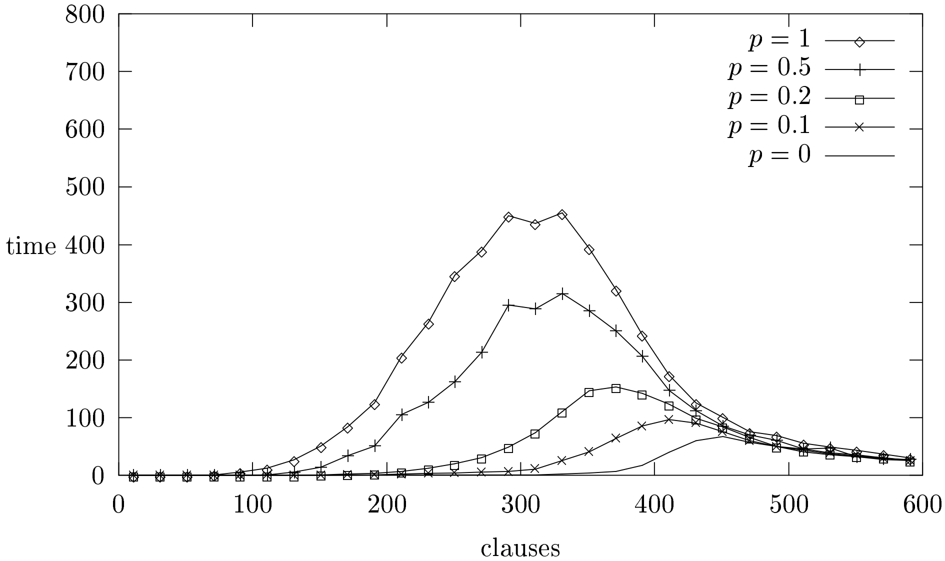

Figure 3: Iterative Search, 100 variables, p = probability of weight 1.

The second test made is about how the running time changes when the number of variables "to be minimized" changes. We consider iterative search only. We run this test with 100 variables, and a number of clauses ranging from 1 to 600. The results of this test are reported in Figure 3. The highest curve is that corresponding to 100% variables minimized, that is, all weights are equal to 1. The other curves are those in which the probability for a variable to have weight 1 is p = 0.5, 0.2, 0.1, and 0. As a rule, the fewer the variables to be minimized, the lesser the running time. In the case in which no variable is to be minimized (i.e. $p = 0$), only one call to the function for finding models is performed, so the search is exactly equivalent to that of the basic DP algorithm. This way, we can compare the search for minimal models and the search of arbitrary models *using the same procedure for finding models*. What is evident from Figure 3, is that finding minimal models is much more difficult than finding arbitrary models.

There is a clear easy-hard-easy pattern in all those curves: our experiment shows that the position of the maximum depends on the number of variables to be minimized. Namely, it is lower when all variables are minimized, and increases as the number of variables to minimize decreases (when no variable is minimized, it coincides with the phase transition point).

4.3 Size of the input

In this experiment we tried to discover how iterative search is affected by the number of considered variables. Since the hardest case is the one in which all variables have weight 1, we made the experiment with this assumption.

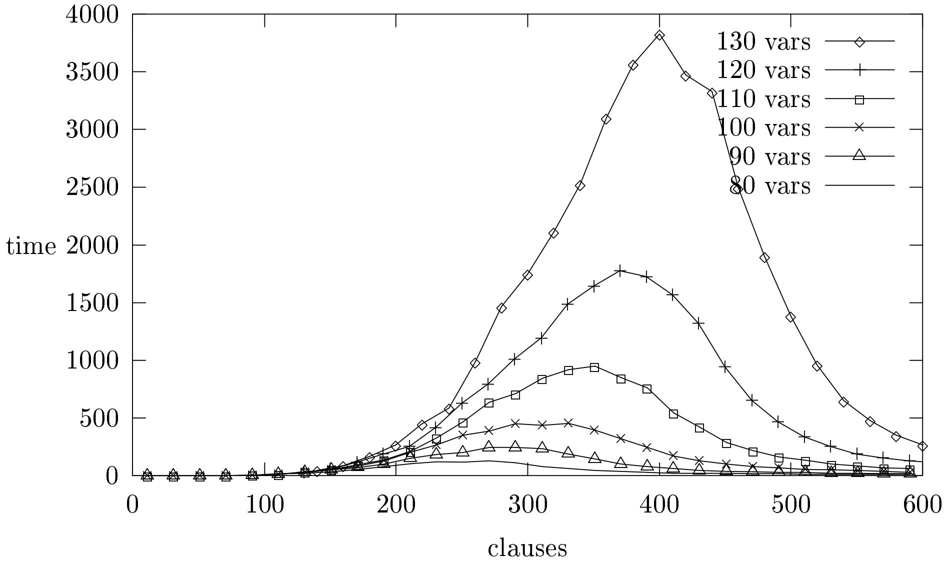

Figure 4: Running time of linear search, depending on the number of variables. All weights are equal to 1.

Results have been reported in Figure 4. The number of variables ranges from 80 to 130 (the tallest curve, of course, corresponds to the experiment with 130 variables). It is clear that increasing the number of variables the running time increases in an exponential way, and that the position of the region of hardest sets of clauses increases as well.

The position of the peak is hard to locate, since the hard region, when the number of variables is less or equal than 100, is quite large. A qualitative estimate seems to indicate that the way the "center" of the hard region changes depending on the number of variables is more or less linear: the peak is located in the point in which the number of clauses is about three times the number of variables.

4.4 Uniformly Distributed Weights

In this experiment, we used weights randomly chosen in the set $\{0, \ldots, 10\}$, with equal probability. We use this distribution of weights only to see the efficiency of search when weights are not bounded to be in the set $\{0, 1\}$. We considered both iterative and binary search. Figure 5 reports the average running time of the two programs.

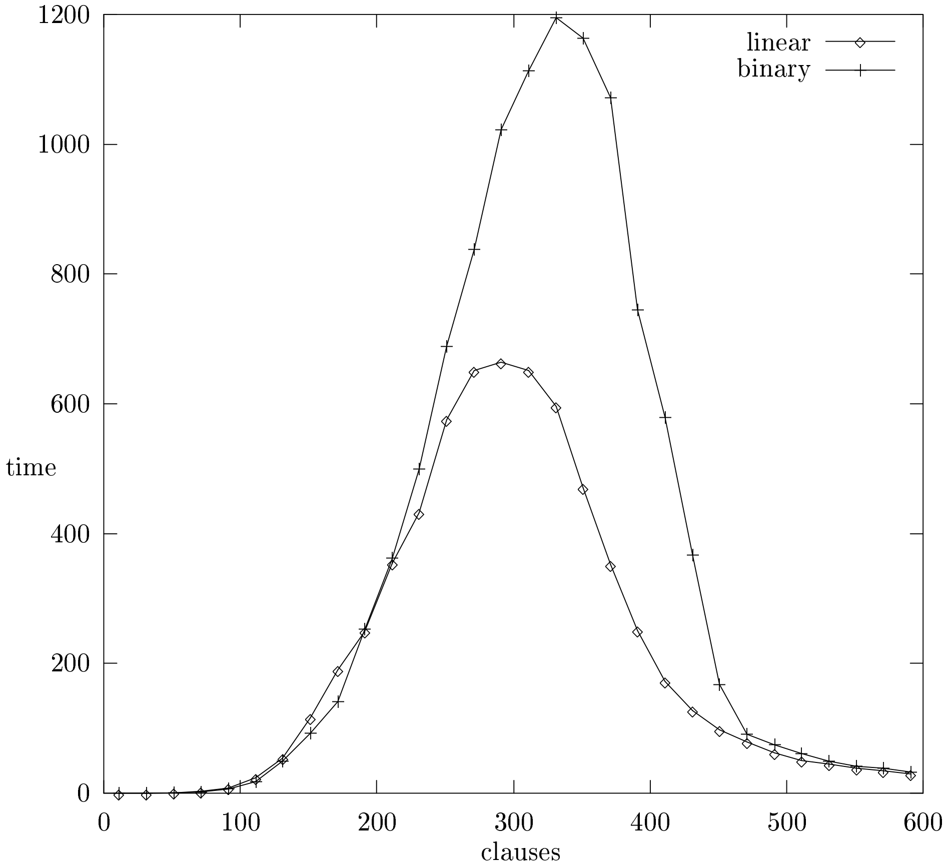

Figure 5: Weights randomly chosen in the set $\{0, \ldots, 10\}$.

Two conclusions can be drawn from this experiment. First, search is slower than in experiments with all weights equal to 1. This can be explained theoretically. Indeed, Theorem 2 state that the problem of finding minimal models is harder if the weights are not bounded. The experiment shows that there may be a correlation between weight range and complexity.

The second result of this experiment is that linear search is still better than binary search, and that the difference is greater than that of the case of weights equal to 1. We are of course supposed to give an explanation of this phenomenon.

4.5 Evaluation of Heuristics

The aim of this experiment is to test the efficiency of the two implemented heuristics. We considered linear search only, in four possible cases: no heuristics, positive heuristics only, over heuristics only, and both. As a result, we have four programs to test. We generate sets of clauses in the usual way, and then we run all four programs over them. Each set of clauses is given as input of all four programs (that is, we did not generate different sets of clauses for the four programs).

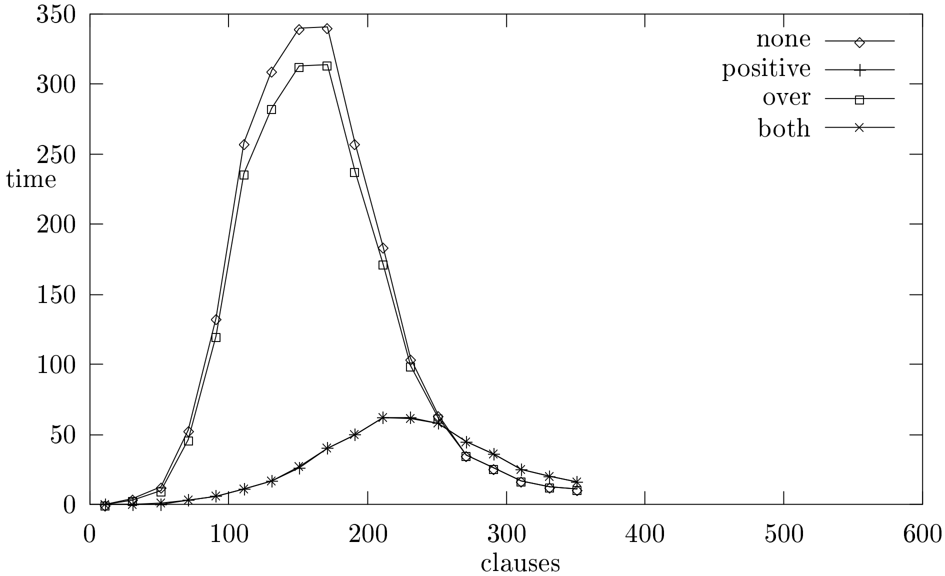

Figure 6: Comparison of heuristics; 70 variables, all minimized.

In the first experiment, we set all weights to 1. The running time is in Figure 6. Since the program in which no heuristics is implemented is very slow, we were forced to make this test on 70 variables only. The result of this test is that the "positive" heuristics allows saving much time, especially in the underconstrained region. Over 250 clauses the complexity of evaluating the set of positive clauses is not balanced by the reduction of the search tree size. However the extra cost is very little, especially when compared to the saving in the underconstrained region.

Just to give an idea of how the heuristics works on 100 variables, we performed 50 tests over sets of 250 and 300 clauses. The results are in Table 1. Apparently, the running time is much lower when the "positive" heuristics is implemented.

Heuristics	250 clauses	300 clauses
None	9204	2583
Over	8542	2388
Positive	302	429
Both	303	424

Table 1: Comparison between different heuristics, 100 variables, all minimized.

A second experiment made is about the performance of heuristics when only a part of the variables is to be minimized. In Figure 7 we show the running time of the four programs when the probability p is equal to 0.1. This experiment has been made with randomly generated clauses of 100 variables.

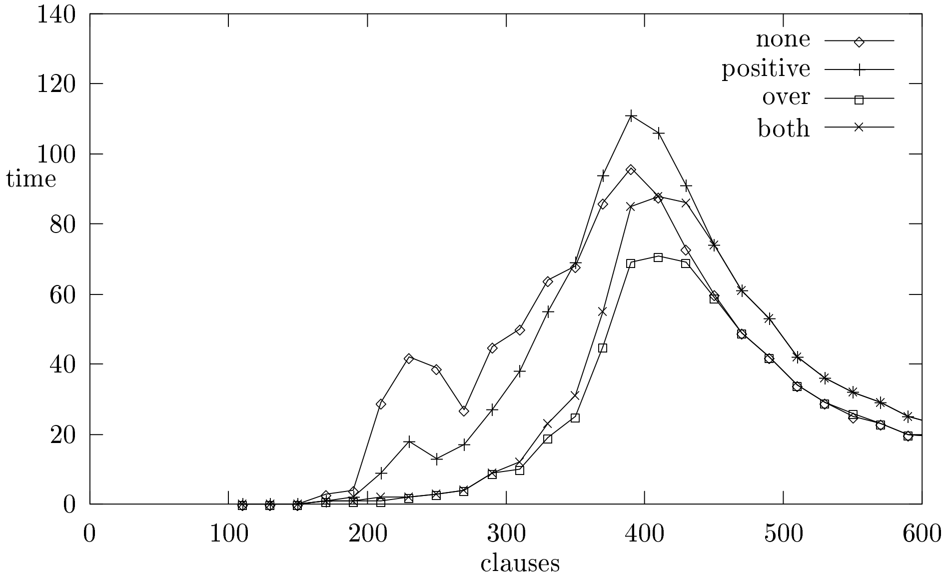

Figure 7: Comparison of heuristics, 10% variables minimized.

Comments on Figure 7: there is not much difference, in the hard region, between the performance of the four programs. The program implementing the "over" heuristics only is always faster than the other three. In the hard region the slowest program is that implementing the "positive" heuristics only. The reason is that, in the case only 10% of variables have weight different from 0, the number of positive clauses is on average very little. As a result, determining the set of positive clauses takes some time but does not give any advantage in reducing the search tree size.

In the underconstrained region (when the number of clauses is less than three times the number of variable), implementing the “over” heuristics gives an enormous advantage. Apparently, in this region there are instances that are very hard for the basic algorithm, but can be solved easily using the “over” heuristics.

Comparing the diagrams, it seems that the best choice is to always implement both heuristics. In the case only a part of the variables is to be minimized, implementing the “over” heuristics only is slightly better, but the difference is not very great.

5 Comparison

In this section, we show the results of the experiments regarding the comparison of our algorithm with one based on local search. MAXWALKSAT [KSJ97] is a local search algorithm that solves the MAXSAT problem: given a set of clauses, each associated to a weight, find the truth assignment that maximize the sum of the weights of the satisfied clauses. Clearly, the problem of finding models of minimum weights can be easily translated to MAXSAT and vice versa: both translations can be done in linear time.

However, making a comparison between our programs and MAXWALKSAT presents some problems to take into account. First, the programs solve different problems (even if conceptually similar). Second, since MAXWALKSAT is a local search program, its behavior depends on some parameters that must be fixed before running it.

From the first point it follows that there is a choice to make regarding the way inputs are generated. Since the two programs must be run over the same instances, and the two programs take as input instances of different problems, we must make a translation.

We can either generate instances of the minimal model problem and then converting them to MAXSAT instances, or vice versa. The choice made in this work is to take the problem of minimal models as the benchmark, that is, sets of clauses are randomly generated, and:

1. our program is run to find a cardinality-minimal model;
2. the same set of clauses is converted to an instance of MAXSAT, and MAXWALKSAT is run to solve it.

The translation is done as follows: for each clause γ of the original instance, the corresponding MAXSAT instance contains a clause γ weighted 10000. The MAXSAT instance contains also a clause $\neg x_i$, weighted 1, for each variable x_i.

The second aspect (there are some parameters that must be chosen to run local search programs) generates still more choices to make. In MAXWALKSAT, there are two main parameters to decide: the number of tries and the number of flips each try (see [KSJ97] for details). The performance is greatly influenced by the value of these parameters. Our decision was to take the default values as suggested by the authors, that is, ten tries and 100000 flips each try.

5.1 Optimality

The first experiment is a comparison of the ability of finding optimal solutions. Our programs, being based on a complete search, always find the optimal solution. On the contrary, MAXWALKSAT may be not able to find the minimal model, or even not able to find any model (even if it exists).

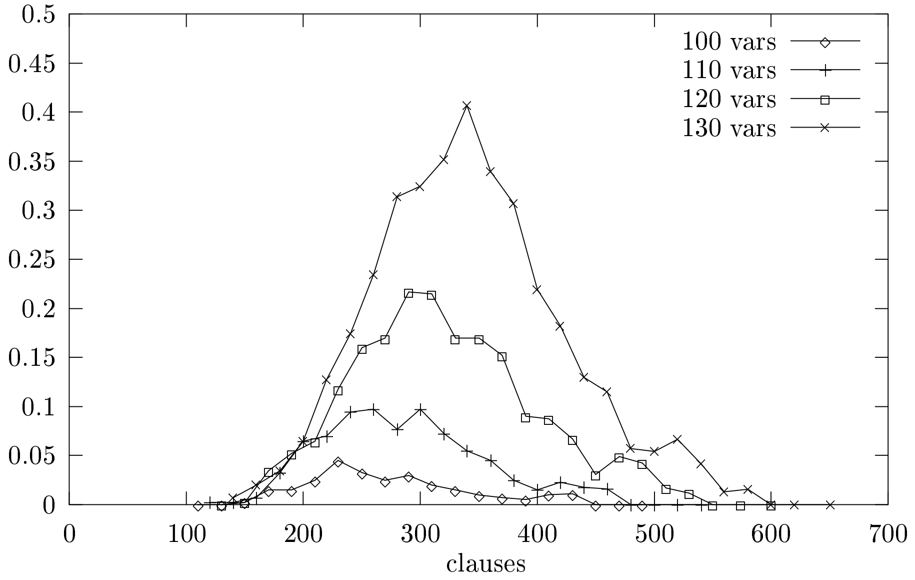

Figure 8: MAXWALKSAT: Distance from optimum.

In this experiment, we generate random sets of clauses in the usual way, we run our program to determine the number of true variables in the cardinality-minimal models; then, we convert the instance to MAXSAT and check whether the best solution found by MAXWALKSAT corresponds to a minimal model or not. When the number of variables is greater than 100, there are some cases in which MAXWALKSAT is not able to find any model, even if it exists. In some other cases, the best solution found is not minimal.

The results of this experiment are summarized in two diagrams. In the first one, we show the distance of the best solution found by MAXWALKSAT to the optimal value. For instance, if the best model found by MAXWALKSAT has 37 variables assigned to true, and the cardinality-minimal models have 35 true variables, the distance from the optimum is 2. Figure 8 shows the average distance of the solution from the optimum, considering a number of variables ranging from 100 to 130. Each point is the average of 200 experiments.

As Figure 8 shows there are some cases in which MAXWALKSAT is not able to find a minimal model. The average distance is not very great. However, it seems to double for each ten variables added. The position of the peak seems also to be linearly related to the number of variables, and is at the position in which the number of clauses is around two times and half the number of variables.

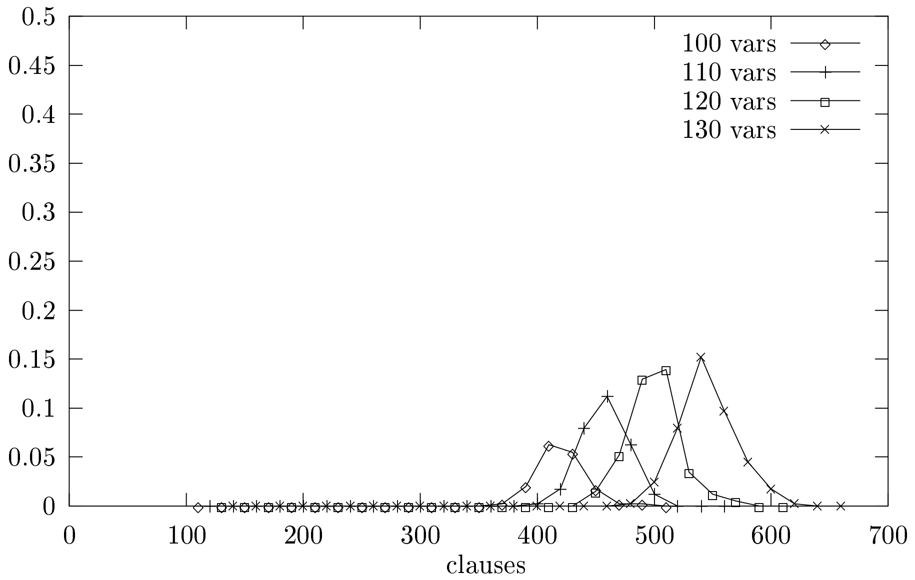

Figure 9: MAXWALKSAT: Violated hard constraints.

The second diagram we show is the number of violated hard constraints. Indeed, in some cases MAXWALKSAT is not able to find any model, even if it exists. When this happens, we consider the number of clauses MAXWALKSAT is not able to satisfy. In other words, we consider the "best model found" by MAXWALKSAT, and we consider the number of clauses it does not satisfy. The result is in Figure 9.

The result is that the number of violated hard constraints has quite a clear "low-high-low" pattern, and that the peak is located around the phase transition point. The "height" of the peak does not increase exponentially with the number of variables.

Let us draw some conclusions about this experiment. First, since our algorithm is complete, this is actually a test on the ability of MAXWALKSAT in finding minimal models. The result is that the difference between the best solution found by MAXWALKSAT and the minimal models is not very great. Indeed, on 130 variables the maximum average distance is about 0.4. This is acceptable in domains like the planning one, in which a plan slightly longer than the optimal ones is not a problem. On the other hand, in domains in which the optimal solution is needed, MAXWALKSAT may be not suitable.

5.2 Running Time

Comparing the running time of a complete algorithm with a local search one is a hard problem. Assuming that MAXWALKSAT is not given the number of true variables of the minimal model, the running time is simply given by a linear function of the number of clauses (assuming that the number of variables and the number of flips and tries is fixed). As a result, the running time of MAXWALKSAT only makes sense if the value of the optimum is given.

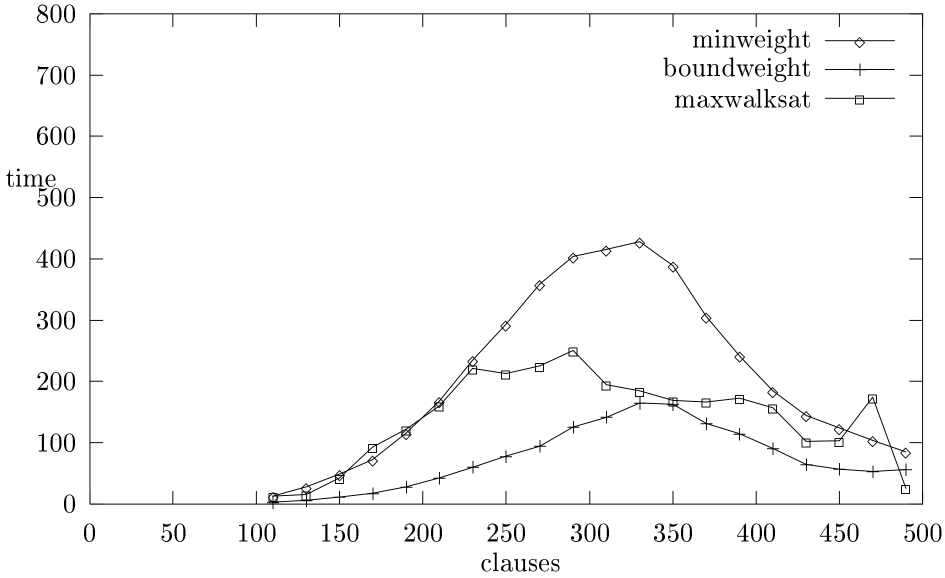

Figure 10: Running time of GSAT compared with complete search, 100 variables.

Operatively, this experiment is as follows:

1. a random set of clauses is generated;
2. a minimal model is found;
3. the instance is converted to MAXSAT;
4. MAXWALKSAT is run over the converted instance and the weight of the minimal models.

In order to make a meaningful comparison, both programs should be given the same information. Since MAXWALKSAT knows the value of the optimum, we consider also what happens when our complete search program is given it: as a final step of the test, we run a program consisting in a single call to the function `find_model_bounded`, in which the bound is the number of true variables in the minimal models. The cases in which MAXWALKSAT was not able to find the optimal solution have been neglected, since the running time is

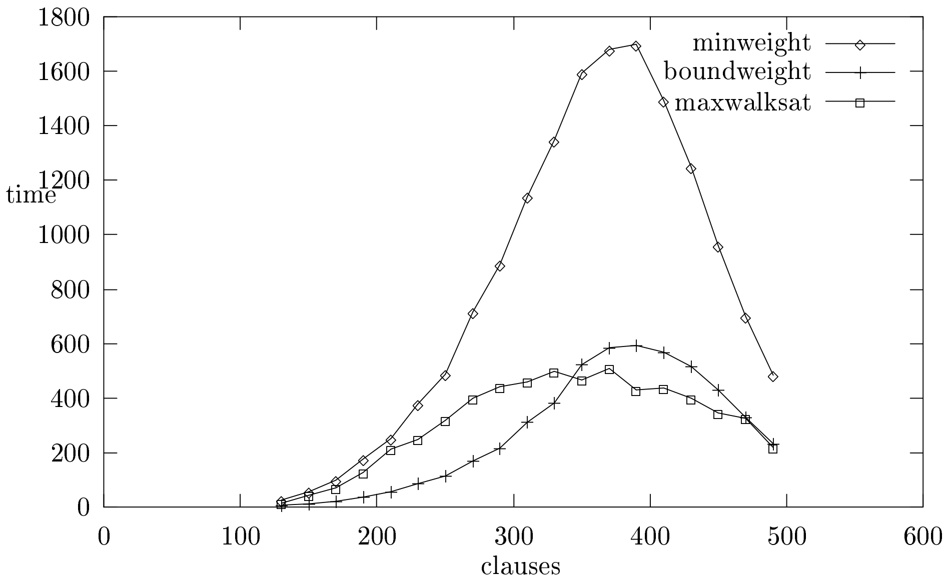

Figure 11: Running time of GSAT compared with complete search, 120 variables.

meaningless. The unsatisfiable instances have been disregarded as well. The results of this experiment are given in Figure 10 and in Figure 11, for 100 and 120 variables, respectively. The latter experiment was stopped at 500 clauses because no satisfiable instance were found for a greater number of clauses.

Apparently, MAXWALKSAT is faster than our program for minimal models. On the other hand, the program for finding models with a bound on the weight has an efficiency similar to MAXWALKSAT.

6 Conclusions

In this paper, we shown algorithms for finding minimal models of a set of propositional clauses. These algorithms have been implemented, and their efficiency has been evaluated using random set of clauses, using different ways of generating weights.

The most surprising result is that iterative search is better than binary search. This is in accord to what reported in [EMW97]: finding a plan becomes harder and harder as we approach the optimum. While this is in accord to other experimental work, it is in contrast with the theory.

When weights are not bounded, it can be foreseen from theory that the behavior of iterative search is much worst. Indeed, assuming no bound on the weights, iterative search requires an exponential number of calls to the function that finds models, while binary search requires only a polynomial number of calls. The only test made with weight different from 0 and 1 is that with weights randomly chosen in $\{0,\ldots,10\}$. The result is that iterative search is, in this case, still better than binary search. We should remark, however, that it is very hard to make any inference from a single test using small weights. We notice also that the weights are bounded, in our programs, by the fact that they are represented by integers.

Our algorithms have been experimentally compared with another one for solving the similar problems. The benchmark problem for optimization is MAXSAT, that is, finding a model that satisfies the maximum possible number of clauses, or more generally, the sum of the weights of satisfies clauses (weights are associated with clauses). Translating MAXSAT to the problem of finding minimal models, and vice versa, can be done in linear time. Our choice to solve the minimal model problem instead of MAXSAT is due to the fact that DPLL can be easily adapted to find minimal models, while adapting it to solve a problem expressed as a MAXSAT instance is more complex. This approach is taken in [BF99]: a possibly non-optimal solution is found using a local search algorithm, and then a backtracking procedure is run to prove that the solution is optimal, or to find a better solution. Comparing this approach to our is not simple. Indeed, using a two-phase algorithm implies that we cannot use the number of branching nodes to compare the efficiency of the two algorithm. Moreover, the program described in [BF99] is not optimized (for instance, the variable to branch is chosen using only the binary clauses, which is not the best heuristics), which makes difficult to make a comparison based on running time.

Let us compare the backtracking phase of the algorithm described in [BF99] and ours. What makes the DPLL algorithm more efficient than simple backtracking is the unit propagation rule, which is correct only on clauses that have to be satisfied. For instance, suppose that the instance contains two unsatisfied clauses:

$$\begin{aligned} 10000: &\quad x_1 \vee x_2 \\ 1: &\quad x_1 \vee x_3 \end{aligned}$$

Assuming x_1 false, we can conclude that x_2 is false only if the first clause cannot be violated: for example, if the total weight of the clauses unsatisfied by the current best solution is less than 10000, then the first clause must be satisfied (in general, this is true whenever the weight of clauses satisfied by the current model plus the weight of the clause is greater than the current upper bound). On the other hand, this condition is very unlikely to hold for the second clause.

Besides the limitations on the use of unit propagation, the algorithm in [BF99] does not employ a rule like our positive heuristics, and the heuristics for the choice of the variable to branch is not optimal.

Before giving the last considerations about how our approach relates to other ones solving similar problems, we give the solution of the quiz proposed in Example 1. The only set of weights such that M_5 has minimum weight is $\{0, 0, 0\}$ (by hypothesis, weights are nonnegative).

A problem similar to the ones analyzed in this paper is that of minimal inference, or circumscription [McC80]. Many algorithms for solving this or similar problems exist: for example, [BED93] propose a backtracking algorithm, while [Nie96] describe the algorithm and implementation of an extension of analytic tableau. The minimal inference problem differs from the cardinality/weight minimal model problem because minimality is defined in terms of set containment, instead of cardinality or weight. In other words, a model M is minimal if there is no other model N such that $N \subset M$ (more complex definitions of minimality exist, but the core is always the set containment relation). As a consequence, a theorem prover for (set containment) minimal model reasoning is expected to be computationally more powerful than ours, since any problem in Π_2^p can be polynomially reduced to it. We believe that our approach has two advantages: first, expressing numerical information (such as the number of variables assigned `true`) in a propositional theory means adding a lot of extra clauses and auxiliary variables, which may slow down performance; second, specific heuristics can be used if the aim is to find cardinality/weight minimal models. As a result, while a solver for the cardinality/weight minimal models cannot solve some problems, it should be more efficient for problems involving a minimization of cardinality/weight of solutions.

There are problems that cannot be solved neither by reducing them to the minimal weight model problem nor to the minimal inference problem. For example, all problems that are Π_3^p-complete, or PSPACE-complete cannot be translated to the minimal model problem in polynomial time. Such problems can be translated to the problem of quantified boolean formulas (QBF). Solvers for this problem exist [CGS98]. While such programs can also solve the problem of minimal models, we expect our algorithms to be better, because of the specific heuristics that can be used in the case of minimal model search.

Acknowledgments

Many thanks to Fabio Massacci for the help provided during the realization of the programs, and in particular for providing parts of the random formula generator. Thanks also go to Henry Kautz for his comments on an earlier version of this paper.

Part of this work has been done while the author was visiting Linköping University. The author thanks Erik Sandewall for his hospitality.

References

[BED93] R. Ben-Eliyahu and R. Dechter. On computing minimal models. In *Proceedings of the Eleventh National Conference on Artificial Intelligence (AAAI-93)*, pages 2–8, 1993.

[BF99] 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.

[BG99] R. Baumgartner and G. Gottlob. On the complexity of model checking for propositional default logics: new results and tractable cases. In *Proceedings of* *the Sixteenth International Joint Conference on Artificial Intelligence (IJCAI-99)*, 1999. To appear.

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

[CGS98] M. Cadoli, A. Giovanardi, and M. Schaerf. An algorithm to evaluate quantified boolean formulae. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence (AAAI-98)*, pages 263–267, 1998.

[DLL62] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem proving. *Communications of the ACM*, 5(7):394–397, 1962.

[DP60] M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7:201–215, 1960.

[EG92] T. Eiter and G. Gottlob. On the complexity of propositional knowledge base revision, updates and counterfactuals. *Artificial Intelligence*, 57:227–270, 1992.

[EMW97] M. Ernst, T. Millstein, and D. Weld. Automatic SAT-compilation of planning problems. In *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence (IJCAI-97)*, pages 1169–1176, 1997.

[Joh90] D. S. Johnson. A catalog of complexity classes. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume A, chapter 2. Elsevier Science Publishers (North-Holland), Amsterdam, 1990.

[Kre88] M. V. Krentel. The complexity of optimization problems. *Journal of Computer and System Sciences*, 36:490–509, 1988.

[KS96] H. A. Kautz and B. Selman. Pushing the envelope: planning, propositional logic, and stochastic search. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence (AAAI-96)*, pages 1194–1201, 1996.

[KS98] H. A. Kautz and B. Selman. BLACKBOX: A new approach to the application of theorem proving to problem solving. In *Working notes of the Workshop on Planning as Combinatorial Search, held in conjunction with AIPS-98*, 1998.

[KSJ97] H. A. Kautz, B. Selman, and Y. Jiang. A general stochastic approach to solving problems with hard and soft constraints. In Dingzhu Gu, Jun Du, and Panos Pardalos, editors, *The Satisfiability Problem: Theory and Applications*, volume 35 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 573–586. American Mathematical Society, 1997.

[LA97] C. M. Li and Anbulagan. Heuristics based on unit propagation for satisfiability problems. In *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence (IJCAI-97)*, pages 366–371, 1997.

[Lib97] P. Liberatore. The complexity of iterated belief revision. In *Proceedings of the Sixth International Conference on Database Theory (ICDT-97)*, pages 276–290, 1997.

[LS97] P. Liberatore and M. Schaerf. Reducing belief revision to circumscription (and viceversa). *Artificial Intelligence*, 93(1–2):261–296, 1997.

[McC80] J. McCarthy. Circumscription - A form of non-monotonic reasoning. *Artificial Intelligence*, 13:27–39, 1980.

[MSL92] D. Mitchell, B. Selman, and H. Levesque. Hard and easy distributions for SAT problems. In *Proceedings of the Tenth National Conference on Artificial Intelligence (AAAI-92)*, pages 459–465, 1992.

[Nie96] I. Niemela. A tableau calculus for minimal model reasoning. In *Proceedings of the fifth workshop on theorem proving with analytic tableau and related methods*, pages 278–294, 1996.

[Pap94] C. H. Papadimitriou. *Computational Complexity*. Addison Wesley, Reading, MA, 1994.

[SKC94] B. Selman, H. A. Kautz, and B. Cohen. Noise strategies for improving local search. In *Proceedings of the Twelfth National Conference on Artificial Intelligence (AAAI-94)*, pages 337–343, 1994.

[SLM92] B. Selman, H. Levesque, and D. Mitchell. A new method for solving hard satisfiability problems. In *Proceedings of the Tenth National Conference on Artificial Intelligence (AAAI-92)*, pages 440–446, 1992.