On the Complexity of Choosing the Branching Literal in DPLL

Paolo Liberatore
Dipartimento di Informatica e Sistemistica
Università di Roma "La Sapienza"
Via Salaria 113, I-00198, Roma, Italy
Email: liberato@dis.uniroma1.it
WWW: http://www.dis.uniroma1.it/~liberato

1 Introduction

The DPLL algorithm, developed by Davis and Putnam [DP60] and by Davis, Logemann, and Loveland [DLL62], is the most popular complete algorithm for the problem of satisfiability of a set of propositional clauses (SAT). While it is outperformed by local search algorithms on satisfiable formulas [SLM92, SKC94], it is still used because local search algorithms can give a definite answer only if the formula is satisfiable. If the formula is unsatisfiable, local search algorithm do not prove that the formula is unsatisfiable, but they only return with a "not found" answer.

Recent programs using the DPLL algorithm are quite different from the first implementations. First, many optimizations have been discovered, for instance the linear unit propagation algorithm. Second, the heuristics for choosing the next literal to branch on have greatly been improved. Developing better heuristics is crucial, especially for the first steps of the algorithm, as a wrong choice may cause an exponential increase of the running time.

There are many papers, in the literature, that focus on the heuristics for choosing the literal to branch in the DPLL algorithm. Hooker and Vinay [HV95] performed a probabilistic and experimental analysis of several heuristics. Usually, when new heuristics are introduced, they are compared with the best ones available at the moment. We are not aware, however, of any work about the general problem of finding the best literal to branch on. Papers in the literature usually analyze particular heuristics, not the general properties of the problem of finding the best literal.

In this paper we give a theoretical analysis of the problem of choosing the next literal to branch on. First, we show a trivial example on which many heuristics produce an exponential search tree while the problem can be solved with a constant-size search tree. Then we consider the complexity of the problem, showing that the problem of deciding whether a literal is among the ones that produce an optimal-size search tree is both NP-hard and coNP-hard, and is in PSPACE. We also give an analysis of approximability of the problem.

2 Preliminaries

The DPLL algorithm is basically a backtracking algorithm in the search space of the partial models, enhanced with three rules aiming at reducing the size of the search tree or the total running time. The backtracking algorithm can be described as follows.

```
model dpll( partial model M ) {

  /* X */

  if M satisfies all clauses
    return M
  if there is a false clause
    return unsatisfiable

  choose unassigned literal l

  M1=dpll( M u {l} )
  if M1 != unsatisfiable
    return M1
  M2=dpll( M u {-l})
  if M2 != unsatisfiable
    return M2
  else
    return unsatisfiable
}
```

The procedure starts with the empty partial model, that is, the model in which all literals are unassigned. If the set of clauses is satisfiable, it returns

one of its partial models, otherwise it returns that the set of clauses is unsatisfiable. The *search tree* is an empty tree if no branching variable is chosen (i.e., the set can be immediately proved satisfiable or unsatisfiable without branching). Otherwise, it is a tree having as a root the variable chosen as the first unassigned literal. Its two subtrees are defined recursively in the same way, by setting the branching variable to true and false, respectively.

The original DPLL uses three rules to speed-up the search. These rules are applied at the beginning of the recursive procedure, in the point marked `/* X */` in the algorithm above.

Unit Propagation: A clause for which all literals but one are assigned, and is not currently satisfied, is called *unit clause*. If a clause is unitary, we can set the value of the only unassigned literal in such a way the clause is satisfied.

Monotone Literals: If, neglecting the satisfied clauses, an unassigned 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.

The third rule is not used in modern implementations of the DPLL algorithm, so we do not consider it any more.

There is a point that is not specified in the algorithm above: how to choose the unassigned literal. Different heuristics of choice have huge differences in performance. Three basic principles are used:

1. choose the literal appearing most often in the set of clauses;
2. choose the literal appearing most often in the shortest clauses;
3. choose the literal producing the maximal number of literals by unit propagation.

There are several heuristics based on these three principles. For instance, the second rule is not usually applied directly in this way: rather, the number of occurrences of the literal and its negation in the shortest clauses is computed, and these two numbers are combined using some rule

[JW90, DABC93, HV95]. A similar procedure is used when the third principle is used. Crawford and Auton [CA93] and Freeman [Fre95] use the third rule only on the first levels of the search tree (for the other levels, the second one is used), while Li and Anbulagan [LA97] use the rule for all levels.

A measure of the efficiency of DPLL is the number of nodes in the search tree. This is defined as the number of nodes that are crossed before arriving either to a valid model or to a point where unsatisfiability is proved. As a result, we define minimal a tree if there is no other tree with less nodes.

The point we analyze is the choice of the next literal.

Definition 1 *A literal x (or $\neg x$) is optimal for a set of clauses F if there exists a minimal search tree of F whose root is labeled with x.*

The three heuristics above do not always choose an optimal literal to branch on. Consider the following sets of clauses (a similar set has been proposed by Ouyang [Ouy96]).

$$
\begin{aligned}
F \;= \{\;& x_1 \vee x_2, \\
& \neg x_1 \vee \neg x_2, \\
& \vdots \\
& x_{n-1} \vee x_n, \\
& \neg x_{n-1} \vee \neg x_n \;\} \\
G \;= \{\;& \neg a \vee \neg b \vee \neg c,\ \neg a \vee \neg b \vee c,\ \neg a \vee b \vee \neg c,\ \neg a \vee b \vee c, \\
& a \vee \neg b \vee \neg c,\ a \vee \neg b \vee c,\ a \vee b \vee \neg c,\ a \vee b \vee c \;\}
\end{aligned}
$$

Clearly, $F \cup G$ is unsatisfiable. Moreover, the heuristics based on the number of literals in the shortest clauses will always choose variables in $\{x_1, \ldots, x_n\}$ as the first $n/2$ branching variables, which means that the obtained tree will be composed of an exponential number of nodes. However, a minimal tree of $F \cup G$ is obtained by branching on a and b: this choice generates a search tree composed of three nodes.

We obtain the same result using other heuristics, for instance the one based on the number of literals generated by unit propagation. The heuristics based on the number of occurrences is in this case better, but it is straightforward to produce a set of clauses that causes it to generate a search tree that is exponentially larger than the optimal ones.

3 Complexity

Having shown that the heuristics currently used do not always choose an optimal literal to branch on, we wonder whether such choice can actually be made. We prove that it is unlikely that such choice can be taken efficiently, as deciding whether a literal is optimal is at least as hard as deciding the satisfiability of propositional formulas.

Theorem 1 *Deciding whether a literal is optimal is* NP*-hard.*

Proof. Let F be a set of clauses over the alphabet $X = \{x_1, \ldots, x_n\}$. We prove that F is satisfiable if and only if a is one of the optimal literals for the set of clauses G defined as follows, where a and b are two new variables (not in X).

$$G = F \cup \{a \vee b, \neg a \vee \neg b\}$$

Let us assume that F is satisfiable. Then, a minimal search tree is composed of a as root, and a chain of literals ending with a model of F. As a result, a is optimal.

Let us assume that F is unsatisfiable. If we begin with branching on a, the search tree has a in the root, and two children, one with a and $\neg b$, the other one with $\neg a$ and b. For each of the two children, we have a subtree showing that F is unsatisfiable. This tree is non-optimal: the tree showing that F is unsatisfiable is half as large, and it proves that G is unsatisfiable without branching on a or b. □

The proof of hardness uses the fact that, by definition, the optimal way of proceeding when the formula is satisfiable is by always guessing the right value for each literal, and thus the optimal search tree for a satisfiable formula is a chain. There is, however, another possible definition of the search tree, in which all possible branches have to be visited even if a model is found. In order to generalize the result above, we consider the case in which we *assume* the set of clauses to be unsatisfiable. In this case, there is no possible disagreement on how the search tree is defined (the way in which we decide the sign to try first for a branching literal does not matter as well).

If the set of clauses is assumed to be unsatisfiable, the problem can be proved to be coNP-hard. With a similar proof, it can be shown to be still NP-hard. The proof requires a number of preliminary lemmas. We give a short sketch.

First, we prove that there exists a (unsatisfiable) set of clauses whose optimal search trees are composed of an exponential number of nodes. This

is proved by Lemma 1. Then, we prove that given a set of clauses F over a set of variables X, there exists a set of clauses G over the set of variables $\{a, b\} \cup X$ which is satisfiable if and only if F is, and a is an optimal branching literal of G. This is proved by Lemma 2.

The coNP-hardness is proved by linking these two results as it is done in Lemma 3. This lemma proves that, if H and G are two sets of clauses over two disjoint sets of variables and $H \cup G$ is unsatisfiable, then the optimal search trees of $H \cup G$ are the optimal search trees of H and G (in other words, no optimal search tree of $H \cup G$ contains both variables of H and G).

We prove that a set of clauses F is unsatisfiable if and only if a is an optimal literal for $H \cup G$, where G is obtained from F by Lemma 2, and H is an unsatisfiable set of clauses whose minimal search trees contains at most 2^{n+2} nodes, where n is the number of variables of F. If F is satisfiable then G is satisfiable as well, and thus the optimal search trees of $H \cup G$ are the optimal search trees of H (which does not contains a). If F is unsatisfiable then G is unsatisfiable as well. Since G contains $n + 2$ variables, its optimal search tree contains at most 2^{n+1} variables, thus they are preferred over the search trees of H, which are larger. Since a is an optimal branching variable of G, it follows that it is optimal for $H \cup G$ if and only if F is unsatisfiable.

Given a set of clauses F, we define a polynomial transformation E, such that $E(F)$ is a set of clauses that is satisfiable if and only if F is satisfiable, but the monotone literal rule never applies, and unit propagation can be used only in a limited way. Namely, the search trees of $E(F)$ are equivalent to the search trees of F obtained without using the unit propagation and monotone literal rules.

Given a set of clauses F over the set of variables $X = \{x_1, \ldots, x_n\}$ we define $E(F)$ as the following set of clauses over the set of variables $X \cup Y$, where $Y = \{y_1, \ldots, y_n\}$.

$$E(F) = \{x_i \vee \neg y_i, \neg x_i \vee y_i \mid x_i \in X\} \cup \{\gamma^d \mid \gamma \in F\}$$

γ^d denotes the clause obtained from γ by replacing each literal x_i with $x_i \vee y_i$, and each literal $\neg x_i$ with $\neg x_i \vee \neg y_i$. Note that $\{x_i \vee \neg y_i, \neg x_i \vee y_i\}$ is equivalent to $x_i \equiv y_i$, that is, once x_i is given a value, y_i is forced to assume the same value, and vice versa.

Clearly, $E(F)$ is satisfiable if and only if F is satisfiable. Moreover, the monotone literal rule can never be applied, and the only cases in which unit

propagation is applied is by inferring the value of x_i from that of y_i and vice versa.

Lemma 1 *For each search tree of $E(F)$ generated using DPLL there is a search tree of the same size for F generated without using the unit propagation and monotone literal rules, and vice versa.*

Proof. First of all, the monotone literal rule cannot be applied to $E(F)$. Indeed, any variable x_i appears at least in two clauses: $x_i \vee \neg y_i$ and $\neg x_i \vee y_i$. The only case in which one of them is already satisfied is when y_i is assigned a value. However, once y_i is assigned, x_i is assigned as well, and the monotone literal rule can be used only on unassigned literals. The same holds for y_i, that is, the truth value of y_i cannot be obtained by applying the monotone literal rule.

The second step of the proof is to show that the only case in which we can infer the value of x_i by unit propagation is when y_i is assigned to a value. In each clause in which x_i occur, y_i is present as well. As a result, the only case in which the only unassigned literal of a clause is x_i is when y_i is already assigned. However, once y_i is assigned to either true or false, x_i is forced to assume the same value by the pair of clauses $x_i \vee \neg y_i$ and $\neg x_i \vee y_i$.

The rest of the proof is almost straightforward. A search tree of F generated without using unit propagation and monotone literal rules is also a search tree for $E(F)$ using DPLL, since the only effect of the monotone literal and the unit propagation rules are to set each y_i to the same value of x_i. Conversely, given a search tree for $E(F)$ generated using DPLL, we can replace each variable y_i with the corresponding x_i, thus obtaining a tree of F for the case in which unit propagation and monotone literal rules are not used. □

An easy consequence of this lemma is the possibility of building sets of clauses having exponentially large optimal search trees. Indeed, a search tree obtained by neglecting the unit propagation and monotone literal rules can be converted into a regular resolution tree "of the same size and shape" [CR74]. Since there are sets of clauses for which all regular resolution trees are exponentially large [Tse70], it follows that there are sets of clauses for which the minimal search trees of DPLL contain an exponential number of nodes. This claim can also be proved by showing that there exists a set of clauses encoding the pigeon-hole principle on which DPLL generates only exponential search trees. The proof, however, is slightly longer than the present one.

A further step is to modify a set of clauses in such a way an optimal literal is always known.

Lemma 2 *Given any set of clauses F, and two variables a and b not occurring in F, the following set of clauses G is satisfiable if and only if F is satisfiable.*

$$G = \{a \vee \neg b, \neg a \vee b\} \cup \{a \vee b \vee \gamma \mid \gamma \in F\} \cup \{\neg a \vee \neg b \vee \gamma \mid \gamma \in F\}$$

Moreover, if F is unsatisfiable, a is an optimal branching literal for G.

Proof. The set of clauses G is equivalent to $a \equiv b$ plus a formula $(a \equiv b) \rightarrow \gamma$ for each clause $\gamma \in F$. Thus, G is equivalent to $(a \equiv b) \wedge F$, which is satisfiable if and only if F is satisfiable.

Let now assume that F is unsatisfiable and prove that a is one of the optimal literals of G. This is proved by induction on the number of variables in F. More precisely, we prove that, for each set of clauses F over n variables, G has a minimal search tree rooted with a.

$n = 0$ Since F is unsatisfiable, it must contains the empty clause $\bot$ (the only two sets of clauses of 0 variables are $\emptyset$ and $\{\bot\}$, and only the latter is unsatisfiable).

In this case, since $\gamma \vee \bot = \gamma$ for any clause γ, it follows that G is:

$$G = \{a \vee \neg b, \neg a \vee b\} \cup \{a \vee b, \neg a \vee \neg b\}$$

The unsatisfiability of G cannot be proved without assigning a value of a variable, since the set does not contains the empty clause, and the unit propagation and monotone literal rules cannot be applied. This proves that the empty tree is not a search tree of G. Moreover, assigning any value to a we obtain contradiction, thus there exists a search tree composed of a single node labeled by a, which is thus optimal.

$n > 0$ We assume that the claim holds for any $m < n$, and prove that it holds for n. Let F be a set of clauses over n variables. First of all, we show that the empty tree is not a search tree for G. Indeed, since any clause contains a pair of literals over $\{a, b\}$, it follows that G contains no unit clause. Moreover, the value of a and b cannot be inferred by monotone literal rule, as both a and b occur both positively and

negatively. As a result, the unsatisfiability of G cannot be proved without choosing a branching variable.

Let T be a minimal search tree for G. Since T is not empty, it has a root. Assume that T is not rooted with a. If it is rooted with b, we can replace b with a and obtain a minimal search tree rooted with a.

Let us assume that the root of T is labeled with a variable x, that is neither a nor b. Since each clause contains both a and b, the truth value of these two variables cannot be obtained from the value of x by unit propagation. Since they occur with both signs, their value cannot be inferred by the monotone literal rule. Since both a and b are still unassigned, and each clause contains both, it follows that no clause can be falsified by setting the value of x.

As a result, T is composed of a root labeled with x, and two non-empty subtrees. Let us consider the subtree corresponding to assigning true to x. Since the tree is optimal, this is a minimal subtree for the formula $G \cup \{x\}$, which can be rewritten as follows.

$$G' = \{a \vee \neg b, \neg a \vee b\} \cup \{a \vee b \vee \gamma \mid \gamma \in F'\} \cup \{\neg a \vee \neg b \vee \gamma \mid \gamma \in F'\}$$

where F' is obtained by assigning x to true in all clauses of F. Since F' contains at most $n-1$ variables, we can apply the induction hypothesis, and conclude that G' has an optimal search tree having a in the root. With a similar proof, we conclude that there exists a minimal search tree for $G \cup \{\neg x\}$ with a in the root.

At this point, we can obtain an optimal search tree for G by moving a in the root, and x in the children. This transformation does not change the size of the search tree, and the truth assignments in the grandchildren of the root are the same four ones of the original tree. Thus, this is a search tree for G whose root is labeled with a.

We remark that the assumption that the optimal search trees of G are not empty is essential. Indeed, if the minimal search trees of G' were composed of 0 nodes, then swapping the nodes would produce a larger tree. □

The following lemma relates the optimal trees of two sets of clauses with the optimal trees of their union.

Lemma 3 *Let H and G be two sets of clauses over two disjoint sets of variables. If $H \cup G$ is unsatisfiable, its optimal search trees are optimal search trees of either H or G.*

Proof. A full proof of this lemma is long but tedious. Let us consider for example the case in which both H and G are unsatisfiable. The proof is by induction on the number of variables in $H \cup G$. The case in which one of the two sets contains 0 variables is trivial. If $H \cup G$ contains $n > 0$ variables, we consider an optimal search tree T. If the root is labeled with a variable x of H, then we can prove that all other nodes are labeled with variables of H. Indeed, the two subtrees of T are optimal search trees for $(H \cup \{x\}) \cup G$ and $(H \cup \{\neg x\}) \cup G$, which (propagating the value of x) are formulas containing $n-1$ variables. Assume, by contradiction, that a subtree contains a variable of G. Applying the induction hypothesis, if one of these subtrees contains a variable in G, then it is a search tree of G. As a result, T is composed of the root and an optimal search tree of G. Since the search tree of G alone is a search tree of $H \cup G$, this contradicts the assumption that T is optimal. □

Using the above lemmas, we can prove that the problem of deciding whether a literal is optimal for a set of clauses is coNP-hard.

Theorem 2 *Deciding whether a literal is optimal for a set of clauses is* coNP*-hard, even if the set is known to be unsatisfiable.*

Proof. Our proof uses the set of clauses U_n such that its minimal search trees without using the unit propagation and monotone literal rules are composed of at least 2^n nodes. As explained above, $E(U_n)$ is a set of clauses such that its minimal search trees are composed of at least 2^n nodes.

Given a set of clauses F over n variables, we build a set of clauses H as follows,

$$H = G \cup E(U_{n+2})$$

where G is the following set of clauses.

$$G = \{a \vee \neg b, \neg a \vee b\} \cup \{a \vee b \vee \gamma \mid \gamma \in F\} \cup \{\neg a \vee \neg b \vee \gamma \mid \gamma \in F\}$$

The sets G and $E(U_{n+2})$ are built using two disjoint sets of variables. The set H is unsatisfiable by construction: since U_{n+2} is unsatisfiable, $E(U_{n+2})$ is unsatisfiable as well. Moreover, a is an optimal literal if and only if F is unsatisfiable.

Let us assume that F is satisfiable. By Lemma 3, the optimal search trees of H are the optimal search trees of $E(U_{n+2})$, which does not contain a at all.

Let us assume that F is unsatisfiable. By Lemma 2, one of the optimal literals of G is a. Moreover, G contains $n+2$ variables, thus its search trees contain at most 2^{n+1} nodes, whereas the optimal search trees of $E(U_{n+2})$ are composed of at least 2^{n+2} nodes. As a result, the minimal search trees of G are smaller than those of $E(U_{n+2})$. By Lemma 3, the optimal search trees of H are the optimal search trees of G, and there is a search tree of G rooted with a. We can conclude that a is an optimal literal. □

The proof can be easily modified to conclude that the problem of deciding whether a literal is optimal is NP-hard even if the set of clauses is known to be unsatisfiable.

Giving an upper bound for the problem of choosing the next literal to branch is more difficult. Indeed, the obvious algorithm of guessing whether there exists a search tree rooted with the literal does not work, as the size of the optimal search trees may be exponential. The only upper bound we can give is PSPACE.

Theorem 3 *Deciding whether a literal is optimal is in* PSPACE*.*

Proof. We give an algorithm for determining the best literals to branch on, and show that this algorithm runs using only a polynomial amount of space. The algorithm is as follows. First, we check whether the set of clauses is satisfiable. If it is, the minimal search tree is a chain of literals. Determining the minimal size of the possible chains of literals only takes a polynomial amount of space.

Let us consider the case in which the set of clauses is unsatisfiable. The algorithm reported in Figure 1 determine the size of the minimal trees of a set of clauses. Deciding whether a literal a is optimal can be done by determining the minimal size s of trees for the set of clauses, and then the minimal size s_1 and s_2 for the cases in which a is set to true and false, respectively. The literal a is optimal if and only if $s = s_1 + s_2 + 1$.

The procedures called by the algorithm are quite standard, and do not deserve any detailed explanation of how they work.

`unsat( LL, F )` Check whether there is a clause whose literals are all falsified by the (partial) assignment `LL`.

```
list of literals LL;
set of clauses F;

int minsize() {
  int L;
  int MIN;
  int SIZEPOS,SIZENEG;
  int NUMMONOT,NUMUP;

  monot( LL, F, NUMMONOT );

  if( unsat( LL, F ) )
    return 0;

  MIN=-1;

  for( L=1; L<=NUMVARIABLES; L++ )
    if( (L not in LL) && (-L not in LL) ) {
      up( L, LL, F, NUMUP );
      SIZEPOS=minsize();
      rmhead( LL, NUMUP );

      up( -L, LL, F, NUMUP );
      SIZENEG=minsize();
      rmhead( LL, NUMUP );

      if( (MIN==-1) || (MIN>SIZEPOS+SIZENEG) )
        MIN=SIZEPOS+SIZENEG;
    }

  rmhead( LL, NUMMONOT );

  return MIN+1;
}
```

Figure 1: The algorithm for finding the size of the minimal search tree.

`monot( LL, F, NUMMONOT )` Apply the monotone literal rule, adding the literals assigned in this way to the head of the list `LL`. The integer `NUMMONOT` is the number of literals obtained by applying the rule.

`up( L, LL, F, NUMUP )` The literal `L` is propagated, and the generated literals are attached to the head of the list `LL`. The variable `NUMUP` contains the number of generated literals.

`rmhead( LL, N )` This is the inverse of the previous functions: the first `N` elements are removed from the list `LL`.

It is easy to prove that this algorithm only requires a polynomial amount of space. Proving that it returns the number of nodes in the minimal search tree is harder, and we do not give a formal proof of correctness. □

4 Approximability

Given that the problem of deciding whether a literal is optimal is NP-hard, we may relax the constraints, asking for a sub-optimal literal. Using the same reduction of Theorem 1, we can prove that we cannot do best than finding a literal producing a search tree that is twice as large as the optimal ones.

Theorem 4 *If it were possible to decide, in polynomial time, whether a literal is the root of a search tree that is at most two times as large as the optimal ones, then* P=NP.

Proof. Consider the reduction of Theorem 1, and assume that we can decide in polynomial time whether there exists a search tree, rooted with the literal, that is at most two times the size of the optimal search trees. Then, we can solve the problem of satisfiability as follows. Given a set of clauses F, we easily compute G as in the reduction of Theorem 1. Then, we check whether a is a sub-optimal literal. If F is satisfiable, a is an optimal literal, thus it is also sub-optimal. If F is unsatisfiable, the search trees generated by a are more than two times as large as the optimal ones, thus a is not a sub-optimal literal. In other words, the satisfiability of F can be determined by checking the sub-optimality of a. □

Having shown that it is likely impossible (i.e., unless P=NP) to decide whether a literal is the root of a search tree twice as large as the optimal

ones, we consider the problem of how much close to the optimum we may achieve in polynomial time. For example: is it possible to decide whether a literal is the root of a tree that is two times as large as the optimal ones plus one? We prove that any literal satisfies this condition.

Theorem 5 *Any literal is the root of a search tree having two times the number of nodes of the optimal trees plus one.*

Proof. Let us consider what happens branching on a literal l which is not optimal. The idea is to try to repair the mistake by making the optimal choice in the children. To each children we attach one of the optimal trees, and then we delete from the obtained tree all nodes on which we branch on literals already assigned (this may happen: for example, if the optimal tree contains a branch on l, this node must be removed as well as the subtree on which we set l to a value different from the assigned one). In the worst case, no node can be removed from this tree. In this case, the tree is composed of the root and two copies of the optimal tree, thus its size is two times the size of the optimal trees plus one. □

What this theorem proves is the fact that being able to decide whether a literal is sub-optimal for a factor of 2 does not imply that we can find a tree of reasonable size. Indeed, any literal satisfies this property, which means that the heuristics based on the number of occurrences of literals in binary clauses makes choices consistent with the principle of choosing a sub-optimal literal. However, we have found an example in which this heuristics generates a search tree that is exponentially larger than the optimal one.

This is due to the fact that having chosen a sub-optimal literal in the root does not mean that we are able to follow the optimal policy in the descendants. Indeed, in any of the two children of the root we face the same choice of literal. If we were able to choose the optimal literal from this point on, we would be able to produce a search tree two times as large as the optimal ones. However, we are forced to make an approximate choice in the children and then in any other descendant, which means that the resulting tree may be much larger than two times the optimal ones.

5 Conclusions

We have shown that the current implementations of DPLL do not always choose the best literal to branch on. In light of a more formal analysis,

however, this is perfectly reasonable, as determining whether a literal is the root of a minimal-size search tree is both NP-hard and coNP-hard problem, which means that is *harder* than deciding the satisfiability of the formula. The NP-hardness is not surprising, while the fact that the problem of choosing the literal is strictly harder than the problem of satisfiability is instead quite surprising. Indeed, choosing the literal is part of the DPLL algorithm for solving the problem of satisfiability, and one would not expect a choice done as part of an algorithm to be harder than the problem solved by the algorithm. The only upper bound we give is PSPACE. Determining the exact complexity is still an open problem.

We have also analyzed the problem of choosing a sub-optimal literal. Namely, we have proved that we cannot decide in polynomial time whether a literal is the root of a tree that is two times as large as the optimal ones, unless P=NP. We also proved that allowing the tree to be two times the size of the optimal ones plus one, any literal is sub-optimal. This gives a complete characterization of how much the problem can be approximated.

A similar investigation has been done for a class of Frege systems by Bonet, Pitassi, and Raz [BPR97]. They relate the possibility of building an optimal proof of TC^0-Frege system in time polynomial in the size of the proof to the existence of a class of polynomial circuits determining factorization.

Acknowledgments

The author thanks Fabio Massacci and Marco Schaerf for their suggestions. Many thanks to Riccardo C. Rosati for a useful discussion.

References

[BPR97] M. Bonet, T. Pitassi, and R. Raz. No feasible interpolation for TC^0-Frege proofs. In *Proc. of FOCS'97*, pages 254–265, 1997.

[CA93] J. M. Crawford and L. D. Auton. Experimental results on the crossover point in satisfiability problems. In *Proc. of AAAI'93*, pages 21–27, 1993.

[CR74] S. Cook and R. Reckhow. On the lenght of proofs in the propositional calculus. In *Proc. of STOC'74*, pages 135–148, 1974.

[DABC93] O. Dubois, P. Andre, Y. Boufkhad, and J. Carlier. SAT versus UNSAT. In D. Johnson and M. Trick, editors, *Second DIMACS challenge: cliques, coloring, and satisfiability*. AMS, 1993.

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

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

[Fre95] J. Freeman. *Improvements to propositional satisfiability seach algorithms*. PhD thesis, University of Pennsylvania, 1995.

[HV95] J. N. Hooker and V. Vinay. Branching rules for satisfiability. *J. of Automated Reasoning*, 15(3):395–383, 1995.

[JW90] R. G. Jeroslow and J. Wang. Solving propositional satisfiability problems. *Annals of Mathematics and Artificial Intelligence*, 1:167–187, 1990.

[LA97] C. M. Li and Anbulagan. Heuristics based on unit propagation for satisfiability problems. In *Proc. of IJCAI'97*, pages 366–371, 1997.

[Ouy96] M. Ouyang. How good are branching rules in DPLL? Technical Report 96–38, DIMACS, 1996.

[SKC94] B. Selman, H. A. Kautz, and B. Cohen. Noise strategies for improving local search. In *Proc. of AAAI'94*, pages 337–343, 1994.

[SLM92] B. Selman, H. Levesque, and D. Mitchell. A new method for solving hard satisfiability problems. In *Proc. of AAAI'92*, pages 440–446, 1992.

[Tse70] G. Tseitin. On the complexity of derivation in propositional calculus. In A. Slisenko, editor, *Studies in constructive mathematics and mathematical logic*, volume II, pages 115–125. Consultants Bureau, 1970.