Satisfiability – algorithms and logic (extended abstract)

Pavel Pudlák*

Mathematical Institute
Academy of Sciences
Prague, Czech Republic

Abstract. We present some recent results on algorithms for satisfiability of *k-CNF* formulas: fastest probabilistic algorithms. We mention some results in proof complexity that can be used to derive lower bounds on classes of algorithms for satisfiability.

1 Introduction

The satisfiability problem is to determine for a given boolean formula whether there exists a satisfying assignment. In general we can consider formulas in arbitrary basis, but most attention is given to CNF's, ie., conjunctions of disjunctions of variables and negated variables. The disjunctions are called *clauses*, the variables and negated variables are called *literals*. The set SAT of satisfiable CNF's was the first set for which NP-completeness was proved [1] and it is still the most important NP-complete set. Since it is an NP-complete set we do not expect that there can be found very efficient algorithms for it. Most researchers believe that there are only exponential algorithms for SAT, still we would like to know how much exponential they must be in general. This problem seems to have less structure than many other combinatorial problems. Therefore more restricted classes are considered, namely *k-SAT*, which is SAT restricted to conjunctions of disjunctions of size at most k, also called *k-CNF*'s. It seems that the best algorithms can only achieve running time which is a fixed root of the number of all assignments.

In this lecture we will describe some recent developments in design of algorithms for SAT. In particular we shall mention new probabilistic algorithms which use much less time than known deterministic ones. Then we shall consider the popular area of proof complexity. Results in this area can be used to prove lower bounds for some classes of algorithms. Finally we shall mention use of quantum computations for SAT. Also we present a new algorithm for 3-SAT which has worse running time than previously know ones, but it introduces a new idea in this field.

---

* The author was partially supported by grant A1019602 of the Academy of Sciences of the Czech Republic and by a joint grant of NSF (USA) and MŠMT (Czech Rep.) INT-9600919/ME-103.

2 Deterministic algorithms

2.1

Most of the proposed algorithms for *SAT* can be viewed as instantiations of the Davis–Putnam procedure.[2] Davis–Putnam procedure is an algorithm which searches the tree of all assignments to propositional variables and at each step it checks if there is a a clause which is false for the partial assignment so far constructed and it checks if all clauses are satisfied. If one clause is false, the search of the branch is stops and the algorithm searches another branch. This is repeated until a satisfying assignment is found or all branches are excluded. When the clauses are short (as in a k-*CNF*, for k constant) the algorithm abandons the partial assignment quite often and in this way it saves a lot of time. The various applications of Davis-Putnam procedure differ in the strategy by which we choose next variable in the search process. We can also view the process as modifying the formula at each step when we assign new variable: we omit all clauses that are true and delete all literals that are false in the remaining clauses. We backtrack when an empty clause appears. Eg. $(x_1 \vee x_2) \wedge (\neg x_1 \vee x_3 \vee x_4)$, after assigning $x_1 := 1$, becomes $x_3 \vee x_4$. After assigning $x_1 := 0, x_2 := 0$, we get just the empty clause, so we have to backtrack.

2.2

The best known such algorithm was described by Monien and Speckenmeyer [10] (see **Algorithm A** in the table below). The strategy of this algorithm is to choose the shortest clause (in some fixed ordering of clauses) and and the first variable in this clause (in some fixed ordering of variables). Suppose the shortest clause has length k. W.l.o.g. let the clause be $x_1 \vee \ldots \vee x_k$, where $x_1, \ldots, x_k$ are variables. The following partial assignments make this clause true 1, 01, 001, $\ldots 0 \ldots 01$ (with $k-1$ zeros), while $0 \ldots 00$ (k zeros) makes it false. Note that these partial assignments will be actually used, since after assigning several 0's the clause will not be satisfied and it will still be the shortest one. Let $F_k(n)$ be the number of steps needed in the worst case. We get the following recurrence $F_k(n) \leq F_k(n-1)+F_k(n-2)+\ldots+F_k(n-k)$. This is too rough and can be easily improved by considering *autark* variables. These are variables which occur in all clauses with the same sign (ie. all without negation, resp. all with negation). For such variables we know that one of the values can be always used whatever values we use for other variables (positive, if the occurrences are positive and vice versa). More generally, an autark partial assignment is an assignment which makes any clause true if the clause contains some assigned variable. When considering assignments to variables to a clause, we first look for an autark assignment. If we find one, which can be done efficiently, we take it and we do not have to consider other assignments to these variables. If not, then any assignment hits another

[2] Apparently this procedure should rather be attributed to Davis, Logemann and Loveland [3]. In this paper we shall use the traditional name having in mind that it may not reflect the history quite accurately.

clause, but does not make it true. Such a clause is then shortened. Thus, except possibly for the first clause, in each step we either have a clause of length at most $k-1$, or use an autark assignment, or we have a clause of length k and in the previous step we used an autark assignment. The autark assignments save more time than having clauses of shorter length, so we can upper bound the number of steps by assuming that in each step we consider a clause of length $k-1$. Hence the recurrence is better: $F_k(n) \leq F_k(n-1) + F_k(n-2) + \ldots + F_k(n-k+1)$. In case of $k=3$, this gives Fibonacci numbers.

2.3

Improvements have been considered mainly for the case of 3-*SAT*, see [15, 16] (**Algorithm B**). The use of Davis-Putnam procedures in all these algorithms gives the impression that this is the only possible way of designing a nontrivial algorithm. Therefore we propose below a new algorithm for 3-*SAT* based on a different idea.

Let the number of variables be n, w.l.o.g. suppose n is even and that all clauses have length exactly 3. The algorithm divides the variables into two equal parts (arbitrarily). For each of the parts takes the clauses which contain variables only from this part and lists all satisfying assignments. This is done by considering all $2^{n/2}$ assignments one by one in each part. (We cannot save much by using some sophisticated subroutine here, as the number of satisfying assignments can be so big.) For each of the two parts we do the following. Take a satisfying assignment a for this part and look at the clauses which contain exactly two variables in this part, thus having one variable in the other part. If the clause is not satisfied, then the remaining variable is forced to have a particular value. It may happen that in this way a variable from the other part is forced to be 0 by one clause and 1 by another. Then the assignment a cannot be extended to all variables and we discard it. Otherwise we extend it by all the forced values in the other part. Thus we get partial assignments which assigns all variables on one part and some on the other. Clearly, there is a satisfying assignment for the whole formula iff there is a pair of consistent partial assignment one from one part the other from the other part.

Thus the task of finding a satisfying assignment reduces to the following question. We have two lists of partial assignments, each of size at most $2^{n/2}$. We have to find a consistent pair consisting of a partial assignment from one list and a partial assignment from the other list. We suspect that algorithms for such problems have been considered, but we were not able to find any related result. We will show that one can solve this problem faster than using the trivial procedure running in time 2^n. The bound that one gets from the algorithm that we will sketch below is worse than the bounds on previously considered algorithms for 3-*SAT*, but very likely it can be significantly improved.

Suppose we have two sets of strings of $\{0, 1, *\}$ of length n denoted by A and B. In A the strings have no $*$'s in the first half and in B the strings have no $*$'s in the second half. We want to find an $a \in A$ and a $b \in B$ which are consistent, ie., on each position where both have a non-$*$ they coincide. Let α be a suitable constant such that $0 < \alpha < 1/2$, the optimal value can be computed later. For a $b \in B$ which has less than $(1/2-\alpha)n$ $*$'s it is possible to find a consistent string in A, if there is any, in B in about $2^{\alpha n}$ steps. To this end we use a data structure to represent A which is a binary tree of depth $\leq n/2$ with branchings corresponding to the first $n/2$ coordinates and the leaves labelled by the remaining parts of the strings. In the same way we can search for a consistent string for every $a \in A$ which has less than $(1/2-\alpha)n$ $*$'s. Altogether this will need at most $2.2^{n/2}.2^{(1/2-\alpha)n} = 2^{n-\alpha n+1}$ steps. If we do not find a consistent pair we discard such strings. Now each $a \in A$ has at most αn 0's and 1's in the second half and each $b \in B$ has at most αn 0's and 1's in the first half. For each $a \in A$ we replace the string a by strings constructed as follows. Take a subset of coordinates in the first half of size $\leq \alpha n$ and replace all entries in the other coordinates in the first half by $*$'s. Do the same with strings in B with the coordinates in the second half. Let the new sets be A^* resp. B^*. Then, clearly, there is a consistent pair (a,b), $a \in A$, $b \in B$, iff $A^* \cap B^* \neq \emptyset$. Finding the intersection can be done efficiently, eg., by sorting the union of the two lists A and B, hence the main term in the estimate will be the size of the two lists which is

$$2^{n/2} \sum_{i \leq \alpha n} \binom{n/2}{i} = 2^{n/2} \sum_{i \leq 2\alpha \cdot n/2} \binom{n/2}{i} \approx 2^{n/2+H(2\alpha)n/2},$$

where H is the binary entropy function. Thus the optimal choice for α is given by the equation $1-2\alpha = H(2\alpha)$ and the running time of the algorithm is $O(2^{0.887n})$.

There are several things that one may try for getting an improvement: find a faster algorithm for finding consistent pairs, use random partition into two sets of variables, but the most interesting would be to find a way to iterate this *divide and conquer* strategy, as that may lead to a substantial shortening of the running time.

3 Probabilistic algorithms

3.1

Recently we proposed a simple algorithm which is essentially the Davis-Putnam procedure with tested variables chosen in random order and assigned random independent values [11] (we shall call it **Algorithm C**). For 3-*SAT* the best known deterministic algorithm Schiermeyer [16] is faster, but for $k > 3$ our algorithm beats Monien–Speckenmeyer [10], which is apparently the fastest algorithm for this case that had been published.

To explain the analysis of this algorithm, the best is to start with the special case where there is an isolated satisfying assignment to a k-*CNF* formula. This means that in Hamming distance 1 there is no satisfying assignment. We sketch an estimate on the probability that this assignment is found.

For a satisfying assignment a to a CNF formula Φ we say that C is a *critical clause* of Φ on a, if exactly one literal of C is true on a. That literal and its variable will also be called *critical.* It is easily seen that for an isolated satisfying assignment each variable is critical in some clause. More generally, if a is an arbitrary satisfying assignment and there is no satisfying assignment which differs from a only in the value of x_i, then x_i is a critical variable of some clause for a.

The critical clauses help us to save some bits when searching for a satisfying assignment. If we assign successively values to variables and it happens that we have so far assigned values according to some satisfying assignment a and it also happens that we have assigned all variables of a critical clause for a except for its critical variable, then the value of the critical variable is determined and we do not have to make choice. Fix an a and a critical clause C of length k. If we take a random order then with probability $1/k$ the critical variable will be the last among the variables of C. If a is isolated, then every variable is critical for some clause so we shall save in the average n/k bits. It is not difficult to turn this intuitive argument to a rigorous proof of an upper bound $2^{(1-1/k)n+o(n)}$ for the expected running time of the algorithm finding an isolated satisfying assignment.

In general we do not know that there is an isolated satisfying assignment. If there is no isolated satisfying assignment, then each satisfying assignment has a neighbour and the less it is isolated the more neighbours it has. When there are many satisfying assignments, then we are very likely to hit one by the random choice. It turns out that the trade-off between isolation and number of satisfying assignments works well and we get exactly the same bound in general. This can be seen as follows. Consider some ordering of variables and an assignment a. Let x_i be a variable and let a' be the part of a before x_i. We shall call a variable x_i *forced*, if a' forces the value for x_i, because x_i is the last variable of some clause which is not yet satisfied by a', ie., x_i appears as the last variable in some critical clause, where x_i is the critical variable. We shall call a variable *branching*, if both $a'0$ and $a'1$ can be extended to a satisfying assignment. Otherwise the variable is called *free.* For a random order the average ratio of free variables to forced variable is at most $(k-1):1$. To hit a satisfying assignment we need to guess the free variables only. Thus the worst case is really when there is only one satisfying assignment.

This algorithm, with a small modification, gives apparently the best bound for polynomial size CNF's with unbounded length's of clauses, namely $2^{n-\epsilon\sqrt{n}}$ for an $\epsilon > 0$. Here is a sketch of the proof. In the modified algorithm we first choose randomly half of the variables. Then we randomly independently assign 0's and 1's to the chosen variables and apply Algorithm C. A standard simple argument shows that after the first phase the probability that a clause of size $\geq \sqrt{n}$ is not satisfied is at most $2^{-\Omega(\sqrt{n})}$. Hence with probability approaching 1 we are left with a $\sqrt{n}$-*CNF* and we will find a satisfying assignment in the expected time $2^{(1-1/\sqrt{n})\frac{n}{2}}$ in the second phase. In this way we may, however, miss some satisfying assignment which does not make true the long clauses in the first phase. So we look explicitly for such assignments to the chosen half of the variables. By Chernoff bounds each clause of length $\sqrt{n}$ has close to $\sqrt{n}/2$ chosen variables. Hence we take the long clause one by one, assign the chosen variables in the clause so that it is false, and try all possible assignments to the chosen variables outside of the clause. Thus we have to consider only $2^{n/2-\sqrt{n}/2+o(\sqrt{n})}$ assignments to the chosen variables. For each of these, we try all assignments to the rest.

3.2

Algorithm C is so simple that it surely must be possible to improve it. One possible direction has been considered and a substantial improvement has been achieved. The idea is to consider assignments which are more isolated – for some $d > 1$ there no other satisfying assignments in Hamming distance $\leq d$. In order to get a better performance it does not suffice to use the simple algorithm above. We shall use it, but first we have to pre-process the formula. This consists in deriving new clauses using resolution. Resolution is in some sense a dual procedure to Davis-Putnam, we shall talk about it more later. Let us just recall that the basic rule of resolution produces from two clauses having a complementary literal a new clause which is the union of the two less the two complementary literals. In the preprocessing phase we derive all clauses which can be derived without exceeding the clause size k^d. In this way we obtain new, slightly longer clauses which enhance the probability that a variable is forced in the random process.

In order explain the idea, let us consider only a single application of the resolution rule. Let $C_1 = x_1 \vee \neg x_2 \vee \neg x_3$ be a critical clause for the assignment $11111\ldots1$, which we assume to be isolated. In this clause x_1 is critical. Then x_2 must also be critical for some clause, say for $C_2 = x_2 \vee \neg x_i \vee \neg x_j$. We can resolve the two clauses and get $x_1 \vee \neg x_3 \vee \neg x_i \vee \neg x_j$. This is again a critical clause in case $1 \neq i, j$. Otherwise the clause is a tautology as it contains $x_1 \vee \neg x_1$. However there is no way to ensure the last condition, hence we have to do something else. If there is no satisfying assignment in distance 2, there must be a clause C_3 which is not satisfied by $00111\ldots1$. C_3 must contain some positive literals, hence it contains x_1 and/or x_2. If it contains x_2, we resolve C_3 with C_1 and we surely get a critical clause with the critical variable x_1. If not, then C_3 is already a critical clause with the critical variable x_1 and it is different from C_1. Thus we always have a new critical clause, therefore the value of x_1 is more likely to be forced.

Let us denote the *search problem* to find a satisfying assignment for *k-CNF*'s having exactly one satisfying assignment by *unique-k-SAT*. In particular for *unique*-3-*SAT* we get an algorithm (**Algorithm D**) with expected running time $O(2^{0.387n})$, while algorithm C gives only $2^{2n/3+o(n)}$.

The analysis of the above algorithm is not simple and it is even much harder to analyze the algorithm for general *k-CNF*'s where no isolation is guaranteed. Until now it was possible to get asymptotically the same bounds for *k-SAT* as for *unique-k-SAT* only for $k > 4$. For $k = 3, 4$ the constants are worse than in the case of unique satisfying assignments. The best bound obtained so far for 3-*SAT* is $O(2^{0.446n})$, nevertheless, already this beats the best previous algorithm of Schiermeyer [16]! This proof was obtained by a computer search, but it is possible to beat the best previous record without a computer search if one is satisfied with only $O(2^{0.533n})$. It seems likely that the case of the unique satisfying assignments is the worst one also for $k = 3, 4$.

3.3

As usual, one would like to know if randomness is really needed. Put otherwise, can we derandomize these algorithms? In case of the algorithms for finding isolated satisfying assignments, it is fairly easy. A closer look at the proof reveals that we do not need to take the ordering completely randomly. For critical clauses of length $\leq k$ we need only that each ordering of a subset of k variables occurs equally likely (since we use the expectation of the number of forced variables). If k is a constant, there are polynomial size probability spaces which have this property and which can be constructed in polynomial time.

For general k-SAT it is a much more difficult task. Consider only the special case when the satisfying assignments have a lot of neighbours, so we can use only the fact that there are many of them. To derandomize this particular case we need to construct a small *hitting set* for such sets of assignments. The construction of such sets is not known. Instead of derandomizing algorithm C directly, we can use the following idea [11], (we shall call it **Algorithm E**). We shall look for *minimal* satisfying assignments, ie., those which have the minimal number of 1's. Then we use the following argument: either there is a minimal solution with few 1's and then we can find it by searching the small set of all such assignments, or a minimal assignment has a lot of 1's and then it has large isolation as it is isolated at least in the coordinates which have value 1. Unfortunately this gives worse constants in the exponent than the randomized algorithm C, see the table.

4 Circuit complexity

The probabilistic algorithms were discovered when working on the complexity of depth 3 AND–OR–NOT circuits. Valiant observed long time ago that proving large lower bounds on such circuits would have interesting consequences. In particular a lower bound $2^{n/o(\log\log n)}$ on depth 3 circuits computing a boolean function of n variables implies that the function cannot be computed by a linear size log-depth circuit. Since 1986 [6] the best lower bounds are only of the form $2^{c\sqrt{n}}$ for a constant c. The progress has been achieved only in improving the constant c. A circuit of depth 3 with top gate OR is an OR of CNF's, thus better understanding of CNF's helps to improve lower bounds on such circuits. By considering isolated (Hamming distance 1) satisfying assignments to k-*CNF*'s the complexity of the parity function has been determined up to a multiplicative constant: the minimal size of depth 3 circuits computing parity of n bits is of the order $n^{1/4}2^{\sqrt{n}}$, see [11] (previously even the best constant in the exponent was not known). In [12] a lower bound $\Omega(2^{c\sqrt{n}})$ with $c > 1$ was proved for BCH codes of small nonconstant minimal distance. It is rather paradoxical that improving *lower* bounds on circuit complexity is connected with improving *upper* bounds on algorithms for SAT.

5 Logic of algorithms

5.1

A big achievement of proof complexity is the result that every algorithm for 3-*SAT* based on Davis-Putnam procedure has worst case complexity at least $2^{c.n}$ for a positive constant c. This follows from a result of Urquhart [18] which uses ideas of Tseitin [17] and Haken [5]. It is worthwhile to explain this result in more details.

The propositional resolution calculus is the system based on the resolution rule described above. Successive applications of the rule produce new clauses from a given set of clauses. The system is complete in the sense that for any clause C which logically follows from a given set it is possible to derive a subclause of C. The system is sound, which means that it is possible to derive only clauses which logically follow from the given set. We consider also the empty clause which has no literals and which represents falsehood. Thus a set of clauses is unsatisfiable iff the empty clause can be derived. Hence we can prove that a CNF is unsatisfiable: take the set of clauses of the formula and derive the empty clause. Proving that a formula is unsatisfiable is the same as proving that the negation is satisfiable by all assignments, ie., the negation is a *tautology*. The size of a resolution proof is the number of clauses, including the initial ones, that are used in order to derive the empty clause.

Resolution can be used to prove that 2-*SAT* is in P. The point is that resolution of two clauses of length at most 2 is a clause of length at most 2 too. The number os such clauses is bounded by a polynomial, so we can systematically generate them all.

A connection with resolution and Davis-Putnam algorithms is given by the following proposition.

Proposition 1. *Suppose that a Davis-Putnam algorithm stops on an unsatisfiable CNF after N steps. Then there exists a resolution proof of unsatisfiability of size $\leq N$.*

Proof. Let T be the search tree on an unsatisfiable formula Φ. This means that for each of T there is a clause of Φ which is false under the partial assignment given by the branch leading to the leaf. We shall pick one such clause for every leaf and extend this labelling to every vertex of the tree as follows. Let v be the parent of u and w, with a clause C the label of u and a clause D the label of w. Let x_i be the variable according which the tree branches at v. If x_i does not occur in C (resp. D) we label v by C (resp. D). If both contain x_i, they contain it with different signs. Then we resolve C with D using x_i and use the result as the label of v. This labelling has the property that the clause C belonging to a vertex v is false under the partial assignment given by the path from the root to v. As the root determines the empty partial assignment, it can only be labelled by the empty clause. Thus the labelling is a resolution proof of unsatisfiability of Φ.

Now it suffices to use a lower bound on resolution proofs. Let us observe that Davis-Putnam algorithms produce resolution proofs in a tree form, while the lower bound on resolution proofs is for general proofs.

Theorem 2 [18]. *There exists a sequence of tautologies Φ_n which are k-DNF's, contain $d.n$ variables, for some constants k, d, and for some positive constant c, every resolution proof of Φ_n has size at least $c.n$.*

It follows that every Davis-Putnam algorithm must use at least $c.n$ steps before it rejects the CNF obtained by negating Φ_n. The tautologies can be constructed explicitly. They express the easy fact that in a graph the number of vertices of odd degree must be even. This tautology cannot be expressed by a k-DNF with a constant k, therefore the statement is restricted to subgraphs of special graphs of constant degree where we also fix which vertices have odd degree (it suffices to have exactly one).

5.2

Another general framework for solving NP-problems is *integer linear programing.* The basic idea here is to represent the problem by linear inequalities with rational coefficients so that the solutions of the problem are encoded as *integer* solutions of the inequalities. The fact that solving linear inequalities in the domain of rational numbers (the *linear programing problem*) can be done efficiently helps in some cases, but not always. It is necessary to use some rules which are valid only for integer solutions. The most popular one among such systems is the *cutting plane* system. In this system one can derive new inequalities by taking positive linear combinations and round down the constant term in the inequality, if all coefficients at variables are integers. For this system unsolvable sets of inequalities have been constructed which do not have subexponential proofs of unsolvability [13]. It follows that any algorithm for k-SAT based on these rules has worst case running time at least 2^{n^ϵ}, for some absolute $\epsilon > 0$, which can be determined from the lower bound on cutting plane proofs. Of course, an algorithm for k-SAT with running time 2^{n^ϵ} with any constant $\epsilon < 1$ would be a sensational result. Thus there is a lot of room for improving, most likely the lower bounds.

For other systems for integer linear programing it is still an open problem to prove nontrivial lower bounds.

5.3

Another popular algorithm, especially in algebra, is Buchberger's algorithm for constructing a Gröbner basis of an ideal of polynomials. In logic we restrict ourselves to the domain $\{0, 1\}$. In terms of polynomial equations this means that we assume equations $x_i^2 = x_i$ for every variable x_i. The natural logical framework for Buchberger's algorithm in the domain of $\{0, 1\}$ is called the *polynomial calculus.* In this calculus we derive polynomials from a given set of polynomials by adding polynomials that have been derived and by multiplying a polynomial that have been derived by an arbitrary polynomial. A contradiction is reached, when we derive a constant nonzero polynomial. Exponential lower bounds on proofs in polynomial calculus have been recently obtained [14, 8], hence we can again conclude that any direct use of the Gröbner basis algorithm cannot produce an algorithm for k-SAT running in subexponential time.

5.4

Fix some sufficiently general class of formulas, eg. DNF's, and let $TAUT$ be the set of tautologies in this class. A propositional proof system is, roughly speaking, a nondeterministic algorithm for $TAUT$. More precisely it is a polynomial time computable function $f : \Sigma^* \rightarrow TAUT$ which is onto and polynomial time computable, and Σ is a finite alphabet [2]. We say that w is a proof of $f(w)$ in the system determined by f. In concrete systems not all strings are proofs, but we can always modify the system by saying that "nonsensical" strings are proofs of some default tautology.

We have considered three concrete proof systems – resolution, cutting planes and polynomial calculus which cover certain types of algorithms. For a general algorithms we observe:

1. Every algorithm for SAT determines a proof system. Namely, for a $\Phi \in TAUT$, the proof of Φ is the computation of the algorithm on input $\neg\Phi$ (the computation that shows that $\neg\Phi$ is not satisfiable).
2. Every algorithm is based on an idea, a theory, assumptions etc., which are used to prove the soundness of the algorithm. This can be used to determine the logical framework, *the logic of the algorithm*, which can be turned into a proof system.

So far we have only one type of pairs consisting of an algorithm and a proof system: a Davis-Putnam algorithm and the resolution proof system. If we could find more such pairs, we could show that other types of algorithms have to run in exponential time, since exponential lower bounds have been proved for several other proof systems. In order to find a proof system for an algorithm, it is always possible to apply the idea in observation 2. One can express the mathematical assumption in a first order theory and then construct a propositional proof system from it using well-known means (see [9]). Unfortunately that would result in a very strong proof system for which we are not able to prove any lower bounds. A more promising way of finding a natural proof system for a given algorithm is to analyze the computations and try to turn them into proofs of some familiar proof systems. The following problems illustrate what I have in mind.

Problem 3. Do randomized algorithm provide some proofs? In particular, the algorithm C is just randomized Davis-Putnam procedure, does it provide resolution proofs on unsatisfiable formulas?

Problem 4. By derandomizing and slightly modifying the algorithm C, we obtained the deterministic algorithm E. Does this algorithm provide resolution proofs on inputs that it rejects? Maybe it gives at least bounded depth Frege proofs?

6 Quantum speed-up

It is well-known that SAT is a self reducible problem, therefore the search problem is equally difficult as the decision problem. It is not surprising then that all so far proposed algorithms are based on some search procedure. Recently Grover [4] proved an interesting general result on quantum computations. He showed that finding a unique element in a list of N elements can be done in expected time $O(\sqrt{N})$ using quantum computations (more precisely $O(t\sqrt{N})$, where t is an upper bound on the time needed to check for an element if it belongs to the list). It follows that some problems for which we use search can be solved faster. For instance, finding a satisfying assignment to a formula with n variables can be done on quantum computers in expected time $2^{n/2+o(n)}$, where we can consider arbitrary polynomial size formulas, or even circuits. Let us recall that the best probabilistic algorithm that we know for the satisfiability of CNF's runs in expected time $2^{n-\epsilon\sqrt{n}}$ for some $\epsilon > 0$.

It seems unlikely that an NP-complete problem would have a polynomial time quantum algorithm. Still quantum algorithms may be substantially faster than the deterministic ones. An interesting question is when we can combine a nontrivial classical algorithm with Grover quantum search algorithm to get a faster algorithm. Let us analyze only the probabilistic algorithm C for the *unique*-3-SAT and leave others to the reader. For this it is very easy to use quantum search to get a square root speed-up (**Algorithm Q**). Take a random ordering of variables. In the favourable case, which occurs sufficiently often, 1/3 of the variables will be forced if we assign successively the values of the unique satisfying assignment. So we need only to find $\frac{2}{3}n$ bits. To find them we search the $2^{\frac{2}{3}n}$ strings using Grover's quantum search. Thus we get a quantum algorithm running in expected time $2^{n/3+o(n)}$.

7 Conclusions

In this short survey we have considered only the most popular discipline in the area of algorithms for SAT, namely the time complexity measured in terms of the number of variables of the formula and we have considered the worst inputs. A lot of research has been done on random instances of SAT. There are also results on algorithms for SAT where time is measured in terms of the number clauses or in terms of the length of the formula [7]. There has also been a lot of experimental work done in this area. While the experimental work may be useful for practical applications, say, when the inputs can be considered random, it does not reveal very much on the behaviour of the algorithms on worst inputs.

Consider for instance the popular idea of assigning the value to a variable which makes more clauses containing the variable true. More precisely, in some process of assigning values to variables, suppose that $x_i = 0$ will make k clauses containing x_i true and $x_i = 1$ will make l clauses containing x_i true. Then choose $x_i = 0$ iff $k > l$. The following formula shows that the such an algorithm would perform very badly on some instances. Suppose the number of variables is divisible by 3. Divide the variables in blocks of size 3 and on each block force the only satisfying assignment to be 111. Thus the only satisfying assignment of the formula will consist of all 1's. This formula has $7n/3$ clauses. Now add all clauses of size 3 which contain exactly one positive literal. Thus most variables will appear negatively, hence improving the satisfying assignment locally with the above rule will lead away from the satisfying assignment.

The most promising and rewarding area for future research seems to be the lower bounds for various classes of algorithms. As shown in Section 5 several tools for such lower bounds have been developed and now we should try to get tighter bounds for classes of algorithms for which we do have lower bounds and determine more classes to which the techniques can be applied.

8 Table

The table below gives the constant c in the upper bounds $2^{cn-o(n)}$ on the expected running time of some algorithms considered above for *k-SAT*. The number in parentheses is for *unique*-3-*SAT*.

k	A	B	C	D	E	Q
3	.695	.582	.667	.446 (.387)	.896	.334
4	.879		.75		.917	.375
5	.947		.8	.651	.931	.4

References

1. S.A. Cook, The complexity of theorem proving procedures, *Proc. 3-rd STOC,* 1971, 151-158.
2. S.A. Cook and A.R. Reckhow, The relative efficiency of propositional proof systems, *J. of Symbolic Logic* 44(1), 1979, 36-50.
3. M. Davis, G. Logemann and D. Loveland, A machine program for theorem proving, *Communications of the ACM 5*, 1962, 394-397.
4. L.K. Grover, A fast quantum mechanical algorithm for database search, *Proc. 28-th STOC 1996*, 212-218.
5. A. Haken, The intractability of resolution, *Theor. Computer Science,* 39, 1985, 297-308.
6. J. Håstad, Almost optimal lower bounds for small depth circuits, *Proc. 18-th STOC,* 1986, 6-20.
7. E.A. Hirsch, Two new upper bounds for *SAT*, *Proc. 9-th SODA,* 1998, to appear.

8. R. Impagliazzo, P. Pudlák and J. Sgall, Lower Bounds for the Polynomial Calculus and the Groebner Basis Algorithm, to appear in *Computational Complexity.*
9. J. Krajíček, Bounded arithmetic, propositional logic, and complexity theory, Cambridge Univ. Press 1995.
10. B. Monien and E. Speckenmeyer, Solving satisfiability in less than 2^n steps, *Discrete Applied Math. 10*, 1985, 287-295.
11. R. Paturi, P. Pudlák and F. Zane, Satisfiability coding lemma, *Proc. 38-th FOCS*, 1997, 566-574.
12. R. Paturi, P. Pudlák, M.E. Saks and F. Zane, An improved exponential-time algorithm for *k-SAT*, preprint, 1998.
13. P. Pudlák, Lower bounds for resolution and cutting planes proofs and monotone computations, *J. of Symb. Logic* 62(3), 1997, 981-998.
14. A. A. Razborov, Lower bounds for the polynomial calculus, to appear in *Computational Complexity.*
15. I. Schiermeyer, Solving 3-Satisfiability in less than 1.579^n steps, *CSL'92*, LNCS 702, 1993, 379-394.
16. I. Schiermeyer, Pure literal look ahead: An $O(1.497^n)$ 3-satisfiability algorithm, preprint, 1996.
17. G.C. Tseitin, On the complexity of derivations in propositional calculus, *Studies in mathematics and mathematical logic, Part II,* ed. A.O. Slisenko, 1968, 115-125.
18. A. Urquhart, Hard examples for resolution, J. of ACM 34, 1987, 209-219.