A Randomized Algorithm for Solving The Satisfiability Problem

Laura A. Cecchi*
Departamento de Informática y Estadística
UNIVERSIDAD NACIONAL DEL COMAHUE
e-mail: `lcecchi@uncoma.edu.ar`

Keywords: Satisfiability Problem - NP-complete problems

Abstract

In spite of the NP-completeness of the *satisfiability decision problem (SAT problem)*, many researchers have been attracted by it because SAT has many applications in Artificial Intelligence. This paper presents a randomized Davis-Putnam based algorithm (RSAT) which solves this problem. Instead of selecting the next literal to be set true or false through a heuristic selection rule, RSAT does it through a random algorithm. RSAT not only improves the well-known Davis-Putnam Procedure that has been implemented with a heuristic selection rule, but avoids the incompleteness problem of the local search algorithms as well.

RSAT is described in detail and it is compared with the heuristic based Davis-Putnam algorithm HDPP. We discuss the main features of the RSAT implementation and we especially analyze the random number generator features.

Although the scope of the experiment is bound by the number of variables, our results indicate that the heuristic can be guessed by a random number generator and even improved. Empirical analysis that support the final conclusions are shown.

*Supported by a fellowship of the Universidad Nacional del Comahue, Neuquén, Argentina.

A Randomized Algorithm for Solving The Satisfiability Problem

1 Introduction

The first computational task shown to be NP-complete by Cook [Coo71] was *propositional satisfiability (SAT problem).* Although the strong argument of NP-completeness of the SAT problem suggests that it does not exist any algorithm to solve this problem in polynomial time, many researchers have been attracted by SAT. The reason is that unlike many other NP-complete problems, SAT has special concern with several computational areas. An important application arises from the Artificial Intelligence area because of its direct connection with reasoning: Let Γ be a deductive database and α be a sentence, then $\Gamma \vdash \alpha$ if and only if $\Gamma \cup \{\neg\alpha\}$ is not satisfiable. Furthermore, there are two closely related search problems:

- *Model generation:* find an interpretation of the variables under which the formula becomes true or report that none exists. Obviously, the existence of a model implies the satisfiability of the formula. Model generation has many applications in Artificial Intelligence; for instance, it can prove the consistency of a theory.

- *Theorem-proving:* find a formal proof (in a sound and complete proof system) of the negation of a formula in question or report that there is no such proof.

Another example is the fault testing for switching circuits. We can examine the circuit corresponding to the conjunction of disjunction expressions and we can ask whether the combination of input values will cause the circuit as a whole to output the right value or not.

Therefore it is important to have algorithms which are able to solve a wide range of instances of the SAT-problem in tolerable time. In order to get a quick answer we may exploit the special structure of the formula under consideration. For example, the class of Horn formulas [DG84] and the 2-SAT formulas [BC94] can be solved in linear time. But sometimes we have no idea whether we can advantageously exploit the structure of an instance to be solved or not. In this case we have to resort to a general algorithm like the well-known *Davis-Putnam Procedure* (DPP)[DP60] which was the first effective automated theorem proving method for producing resolution refutations [Vel89]. Most implementations of good algorithms for testing Boolean formulas are based on DPP. These implementations differ in the data structures for representing formulas as well as in the selection method for choosing a propositional literal.

The purpose of this paper is to introduce a Davis-Putnam based algorithm which analyzes an alternative strategy for selecting a literal: to let a randomized algorithm make the decision. Impetus for this work has been given by the variety of random based SAT algorithms which have been proved to be more efficient than DPP. For instance, hill-climbing (local search) algorithms like *GSAT* [SLM92] and *Random Walk.* The main problem of local search is its incompleteness. Since we are essentially interested in a complete algorithm for solving all propositional formula instances, we have been investigating the behaviour of a *randomized SAT Davis-Putnam based algorithm (RSAT)* by comparing it with heuristic based DPP.

The organization of this paper is as follows. First, we introduce some basic concepts together with the notation that will be used in what follows. Section 3 presents randomized algorithms analizing not only its most attractive features but its disadvantages as well. Section 4 gives a detailed RSAT description, discussing the difference between RSAT and the heuristic based DDP. The following section presents some features of RSAT implementation that are closely related to the running time needed to compute a satisfiability test. In section 6 RSAT is compared empirically with DPP, showing some experimental results. Finally we summarize the main results and we propose future works.

2 Basic Concepts

All Boolean formulas F considered in this paper are in conjunctive normal form (CNF). Let's introduce some notation.

- Let V be the set of m boolean *variables.*
- $L = \{x, \bar{x} | x \in \mathcal{V}\}$ is called the set of *literals* corresponding to V.
- $C_i = (l_1 \vee l_2 \vee \ldots \vee l_{r_i})$ is a *clause* such that $l_j \in L : 1 \leq j \leq r_i$. A clause of length r_i es called a r_i-clause. In particular, a 1-clause is called a unit clause. If every clause in a SAT testing input formula is a K-clause, then the SAT problem will be called K-SAT problem.
- $F = C_1 \wedge C_2 \wedge \ldots \wedge C_n$, is called a *formula* in CNF, where each $C_i : 1 \leq i \leq n$ is a clause. Henceforth, when referring to formula we mean a CNF formula.

A *truth assignment* is a mapping t which assigns each variable in V a *true* value or a *false* value. For each $x \in V$, $t(x)$ is *true* if and only if $t(\bar{x})$ is *false.* Let F_x $(F_{\bar{x}})$ denote the resulting subformula of F before assigning the true (false) value to the variable x. A clause C is satisfiable if and only if $t(l)$ is true for at least one literal l in C. An assignment *satisfies* a formula F if and only if it satisfies each clause in F. Finally, a formula F is satisfiable if there is at least one satisfying truth assignment for F. The SAT problem is the decision problem that tests whether a formula in CNF is satisfiable or not.

3 Randomized Algorithms

By definition, a randomized algorithm leaves some of its decisions to chance, *i.e.*, at least once during the algorithm, a random number is used to make a decision. The fundamental characteristic of these algorithms is that they implement non deterministic computation, in other words, they react differently if they are applied twice to the same instance.

When an algorithm is confronted with a choice, it is sometimes preferable to choose a course of action at random rather than to spend time finding out which alternative is the best. The algorithm that will be introduced, profits from this advantage of the randomized algoritms. However the worst-case running time of the randomized algorithm is always the same as the worst-case running time of the nonrandomized algorithm. Clearly, it is always possible that bad luck will force the algorithm to explore many unfruitful possibilities.

Since randomized algoritms outperform not only systematic exploration of the nodes of the implicit tree but a variety of intelligent exploration techniques as well, different problems have been implemented through a randomized algorithm. Some examples are *primality testing* which determines whether or not a large number is prime and *the eight queens problem* [BB88] which places eight queens on a 8-by-8 rectangular board so that no two queens are attacking each other.

As we need random numbers, we must have a method to generate them. Actually, true randomness is virtually impossible to do on a computer, since the numbers will depend on an algorithm. Generally, it is enough to produce pseudorandom numbers which are numbers that seem to be random. Pseudorandom number generators are deterministic procedures that are able to generate long sequences of values that seem to have all the statistical properties of a random sequence. To start the sequence, the pseudorandom number generator requires an initial value called a *seed.* On giving the same seed, the peudorandom number generator will always produce the same sequence. In order to obtain different sequences, the seed must be chosen, for example, so that it depends on the time or the date.

Fortunately, most modern programming languages include such a generator although some implementations should be used with caution. As no genuinely random generator is available, some care must be taken when analizing randomized algorithms.

There exists a variety of SAT algorithms which use random decision, for instance, local search GSAT. Basically local search begins with a random truth assignment and tries to reach a solution by gradually moving to search tree nodes with the largest increase in the total number of clauses that are satisfied. Unfortunately, local search could fail to find a satisfying assignment of propositional formulas even if one exists, *i.e.*, local search is incomplete as we have remarked above. But this is not the only problem local search has. In case the formula is insatisfiable, it will always answer that no satisfying assignment was found, leaving us in doubt. RSAT was defined to avoid all these problems but it has his owns that will be discussed below.

4 The RSAT algorithm

The obvious way of solving algorithmically the SAT problem with m logical variables is to generate all possible assignments systematically and to test each one with the given formula. If it satisfies the formula quit; otherwise, continue generating and testing. If none of the 2^m possible assignments satisfies the formula, it is unsatisfiable. Unfortunately, the time taken by this algorithm tends to exponentially with m. When the instances of the problem are satisfiables, the algorithm may generate anywhere between 1 and 2^m assignments before arriving at a solution.

A slightly better algorithm was discovered by M. Davis and H. Putnam in the early 1960s. Figure 1 shows the well-known DPP which is the base of the RSAT algorithm. It performs a backtracking search in the space of all truth assignments, incrementally

```
Procedure DP
Input: A propositional formula $F$ in cnf.
Output: "Satisfiable" or "Unsatisfiable".

  Step 1: Unit Clause Rule
          While $F$ contains an unit clause but no two complementary unit clauses
              Select an unit clause and bind a variable in it to satisfy the clause.
  Step 2: Satisfiability Checking
          If all clauses are satisfied, return "Satisfiable".
  Step 3: Unsatisfiability Checking
          If an empty clause exists, return "Unsatisfiable".
  Step 4: Splitting Rule
          Select a variable whose value is not assigned. Assign true to it and call DP.
          If the result is "Satisfiable" then return "Satisfiable". Otherwise, assign
          false to the variable and call DP again. Return the result of it.
```

Figure 1: DPP improved with the unit clause rule.

assigning values to variables. If current partial assignment has failed to satisfy one of the clauses, the algorithm performs a backtracking since there is no point in continuing. If, however, all clauses are satisfied by the current partial assignment, then DPP exits and any remaining variables may be assigned arbitrary values. So the satisfiability and unsatisfiability checking avoid to explore all the implicity search space in almost all the cases.

The DPP efficiency can still be improved if the formula is simplified as much as possible before reaching the splitting rule (Step 4). A powerful simplification is the *unit clause rule* that is applied in Step 1. This rule sets true the literals of the unit clauses except when two complementary unit clauses are contained in the formula. In other words, all clauses containing a unit clause literal will be deleted, and every clause $C_i \vee \neg x \vee C_j$ containing a negated unit clause literal $\neg x$ will be replaced by $C_i \vee C_j$.

In the original DPP, the pure literal rule is also used to improve it. The pure literal rule forces a literal l_i to be set *true (false)* if l_i occurs only positively (negatively) in the formula. Some authors [BS][Yug95] have pointed out that including the pure literal rule in the DPP slows down the algorithm, so it is not considered here.

Despite the improvement, the performance of the DPP heavily depends on the ability of predicting which one of the propositional literals should be selected to be given a truth value so that the search space remains as small as possible. Guessing a propositional literal l may depend upon whether we expect the formula F to be satisfiable or not. In case F is unsatisfiable, we should choose x such that F_x as well as $F_{\bar{x}}$ are quite smaller than F, because we have to visit the search tree corresponding to F almost completely (remember that the satisfiability and unsatisfiability checking and the unit clause rule will prune some search tree branches). When F is satisfiable we should choose x such that the given truth value corresponds to a satisfying truth assignment, in order to visit the minimum number of branches in the search tree. Unfortunately, determining such literal is not so easy.

The most usual way to do it is to apply a good heuristic selection rule. An heuristic selection rule may be to assign a value to the literal with the greatest number of occurrences in the formula. The underlying idea is to reduce the clause size of the greatest number of clauses so that we are left with unit clauses as soon as possible. Instead of deciding the next literal through a quite intelligent heuristic we can do it randomly. RSAT implements this strategy by randomly choosing the variable to be bind next.

RSAT as a based DPP algorithm, applies the unit clause rule and the satisfiability and unsatisfiability checking in the same way DPP does it. On Step 4, RSAT specifies how to select the next variable, remaining as follows:

Step 4: Splitting Rule
    Select randomly a variable of the current formula whose value has not been assigned yet. Assign true to it and call RSAT.
    If the result is "Satisfiable" then return "Satisfiable".
    Otherwise, assign false to the variable and call RSAT again. Return the result of it.

It is perhaps important to remark that once a random generator has produced a random number, it must be uniquely mapped to a variable of the current formula. We can obtain a variety of RSAT algorithms by implementing different mapping. On section 5 we will describe the way, we have done it.

At first sight, it seems that a quite intelligent rule is better than a random rule, however this is not the case. On flipping a coin, we can select the right literal without wasting time computing an heuristic. Even though some decisions seem to be systematic we can avoid checking all the paths jumping almost directly to the solution by a random choice.

5 RSAT Implementation

RSAT has been implemented in PASCAL on a 32 Mbytes memory Pentium machine under MS-DOS. In spite of the memory size, PASCAL allows the heap to have up to 640 Kbytes of memory available, so this places a constraint upon the implementation. In order to avoid memory space problems for storing formulas, the suitable data structure chosen is a linked list:

- The formula is stored as a list of clauses.
- A clause is represented by a list of its literals. The heading clause includes the clause length.

Furthermore, RSAT implementation tries to free space by releasing memory cells as much as possible. RSAT shares all the implementation features with the implemented DPP which uses the heuristic selection rule explained above (HDPP in what follows).

Since randomized algorithms implement a non deterministic computation, RSAT running time depends not only on the input Boolean formula but on the random number generator as well. So to discuss RSAT efficiency, we must also analyze the random number generator background. The RSAT implementation uses the PASCAL predefined function `Random` which produces pseudorandom numbers from a seed initiated by the procedure `Randomize`. In spite of the fact that the function `Random` follows a mathematic rule to produce a pseudorandom number sequence, the pseudorandom numbers are still difficult to predict.

In the original DPP nothing is said explicitly about which unit clause will be bind next (Step 1). Both RSAT and HDPP implement a deterministic selection by choosing the first unit clause occurrence to appear.

The name of each variable contains a pseudorandom number that will uniquely identify this variable, in order to compute the mapping straight.

Even thought RSAT, DPP and HDPP implement on Step 4 a deterministic bind order (first set the literal true and then false) actually we can change it. Furthermore we can still choose randomly whether we follow the true branch or the false one when splitting up the formula. However this alternative RSAT algorithm is not analyzed in this paper.

6 Experimental Results

RSAT has been compared to the improved DPP with the heuristic selection rule explained above (section 4) by testing both algorithms on the same collection of random formulas. Selecting good test instances is very important when evaluating the performance of algorithms empirically. The formula sample to be tested on must include not only random formulas but natural ones. The first ones will reflect general cases while the latter will reflect the real world. Even though we can think that random formulas lack of any underlying hidden structure, we may obtain wrong experimental results [CI95] unless some care is taken in sampling formulas. At the moment of writing this paper, RSAT has only been tested on a subset of random formulas that we describe below.

The complexity of RSAT and HDPP strongly depends on the distribution of CNF formulas to be tested on. In our sample, each instance is obtained by generating M random clauses. All clauses contained in the CNF formulas are fixed length and have been generated by randomly selecting K variables from the set V of N variables. Each variable is negated with a 50% probability. It is neither guaranteed that all N variables occur in the formula, nor assured that no double clauses are generated.

When generating the sample, if all formulas considered are either satisfiable (in particular tautologies) or insatisfiable then it seems that Davis-Putnam based algorithms will test them easily and we may not draw right conclusions. The hardest formulas for Davis-Putnam based SAT solvers appear to lie around the region where there is 50% chance for the randomly generated formula to be satisfiable. The main difficulty of sampling hardest formulas is the relation between the number of clauses M and the number

	r_k
3-SAT	4.3
4-SAT	9.9
5-SAT	21.1

Table 1: Clause-Variable Ratio.

	K-RSAT	K-HDPP
$K = 3$	$1,48$	$2,02$
$K = 4$	$1,595$	$2,05$
$K = 5$	$1,38$	$2,435$

Table 2: Attempt average over 200 hard K-formulas with up to 50,25, 15 variables respectively.

of variables N. For instance, in case of 3-SAT the hardest formula ratio turned out to be near $4,3$ ($M \approx 4.3N$), even thought the exact ratio is not known. Experiments over 3-SAT formulas have shown that random 3-SAT formulas are satisfiable (unsatisfiable) with high probability if the ratio is less than $2,9$ (greater than $5,2$). Table 1 shows the clause-variable ratio estimation r_k corresponding to the hardest K-formulas for Davis-Putnam based SAT solvers [BS]. Because of the space problem, RSAT and HDPP have been tested over a set of hardest random formulas with a limited number of variables respecting the ratio estimations of Table 1.

Rather than compare running time we will compare attempt numbers,*i.e.*, the number of branches the algorithm goes through reflecting indirectly the running time. Table 2 summarizes some of our results. It shows the relationship between attempt average of K-RSAT and K-HDPP over 200 random K-CNF formula instances. For example, when $K = 3$, it was necessary to try nearly $1,48$ times before success running on the 3-RSAT algorithm while in case of 3-HDPP it is necessary to try nearly $2,02$ times before success. In case $K = 5$ HDPP has tried one more time than RSAT.

Although the scope of the experiment is bound by the number of variables , it is clear that this heuristic can be *guessed* by a random generator and even improved. We can also expect that as the heuristic computation time is generally slower than a random computation time, RSAT will also excel HDDP in this feature.

7 Conclusions

In this paper we have presented a randomized Davis Putnam based algorithm that outperforms the heuristic based DPP, and does not suffer from the incompleteness problem of the local search algorithms.

Our experimentals results on a subset of fixed clause length random formulas, have shown that a random decision can "guess" the heuristic one and even improve it. Even though we are not able to guarantee that RSAT will manage all CNF formulas better than HDPP, the partial results obtained encourage us to go on comparing RSAT over other subsets of formulas. Furthermore good test instances might be helpful because they can give us some hint to improve RSAT.

New algorithms have been developed in order to produce special subsets of formulas, like satisfiable formulas that have only one solution [CI95]. Future work would be finding out whether RSAT behaviour will agree with present results when tested over a sample of satisfiable formulae with only one solution.

Although RSAT finds a solution for insatisfiable formulas and is not incomplete, important questions are whether RSAT is better than local search and for what kind of instances it is so. Answering these questions would be also future work.

References

[BB88] Gilles Brassard and Paul Bratley. *Algorithmics: Theory and Practice*. Prentice-Hall, London, U.K., 1988.

[BC94] D.P. Bovet and P. Crescenzi. *Introduction to the Theory of Complexity*. Prentice-Hall, Hertfordshire, U.K., 1994.

[BS] Max Böhm and Ewald Speckenmeyer. A Fast Parallel SAT-Solver - Efficient Workload Balancing. *Accepted for publication in: Annals of Mathematics and Artificial Intelligence*.

[CI95] Byungki Cha and Kazuo Iwama. Performance Test of Local Search Algorithms Using New Types of Random CNF Formulas. *IJCAI*, pages 304–310, 1995.

[Coo71] S.A. Cook. The Complexity of Theorem-Proving Procedures. In *Proceedings of the 3rd Annual ACM Symposium on Theory of Computing*, pages 151–158, 1971.

[DG84] W. Dowling and J. Gallier. Linear-Time Algorithms for Testing the Satisfiability of Propositional Horn Formulae . *Logic Programming*, 3:267–284, 1984.

[DP60] M. Davis and H. Putnam. A Computing Procedure for Quantification Theory . *J. ACM*, 7:201–215, 1960.

[SLM92] Bart Selman, Hector Levesque, and David Mitchell. A New Method for Solving Hard Satisfiability Problems. In *Proc. AAAI-92*, pages 440–446, San Jose,CA, July 1992.

[Vel89] André Vellino. *The Complexity of Automated Reasoning*. PhD thesis, University of Toronto, 1989.

[Yug95] Nobuhiro Yugami. Theoretical Analysis of Davis-Putnam Procedure and Propositional Satisfiability. *IJCAI*, pages 282–288, 1995.