An Overview of a Probabilistic Approach to Solving the Satisfiability Problem
Summary of Papers

Paturi *et al.*: Satisfiability Coding Lemma
Paturi et al.: An Improved Exponential-time Algorithm for *k*-SAT

Timo Asikainen
timo.asikainen@hut.fi

April 16th, 2003

Abstract

The satisfiability problem of propositional logic is the object of intense studies. This seminar report gives an overview of the results achieved in two papers, [PPZ97] and [PPSZ98]. The main contribution of the papers is developing and analysing algorithms that solve the satisfiability problem given in the conjunctive normal form (CNF) in less than 2^n steps, where n is the number of variables in an instance of the problem. In [PPZ97], a randomised algorithm for k-CNF with time complexity $O(n^2|F|2^{n-n/k})$, where k is the maximum number of literals in any clause of the formula, is introduced. Further, a deterministic variant with a running time approaching $2^{n-n/2k}$ as n and k get large is given. In [PPSZ98], a reprocessing step is used to achieve an improvement over the results of [PPZ97]: a randomised algorithm that solves e.g. the 3-CNF problem in $2^{0.446n}$ steps is introduced.

1. Introduction

The satisfiability problem is of central theoretical and practical interest. Consequently, many solvers have been implemented aiming at effectively solving instances of the problem. On the other hand, there is a significant theoretical effort for finding algorithms with favourable computational properties; of course, under the **P** ≠ **NP** assumption, the efforts have concentrated on superpolynomial, mainly exponential algorithms.

The papers, [PPZ97] and [PPSZ98], presented in this report take the theoretical approach to attacking the satisfiability problem. This paper gives an introduction to the algorithms and demonstrates their functionality with simple examples. The algorithms are built on an observation on the structure of the solution space of the satisfiability problem. In more detail, the basic idea underlying both of the papers is that satisfying solutions that are, in a sense, far from each other can be encoded with fewer than n bits, where n is number of variables. Two exponential-time algorithms, a randomised and a deterministic, are presented in [PPZ97]. The algorithms take directly advantage of the idea of encoding solutions. In [PPSZ98], a a pre-processing step is added to the randomised algorithm from [PPZ97]. The added step improves the probability of finding a solution, thus enhancing the performance of the algorithm.

The remainder of this report is organised as follows. In Section 2, we briefly define the basic concepts necessary for understanding the papers. In Section 3, we explain the key idea underlying the referred work. The algorithms based on the idea are described and analysed in Section 4. Thereafter, Section 5 represents an enhancement to the previously presented algorithms, and algorithms leveraging the enhancement. Section 6 rounds up the paper with conclusions and a cursory comparison with related work.

2. Basic concepts

The *satisfiability problem* of propositional logic is the following: given a Boolean formula F, is there a truth assignment to the variables of F such that F evaluates to value *true* under the truth assignment. For practical purposes, it is often interesting to know what is the satisfying truth assignment: the version of the satisfiability problem where a satisfying assignment is given as an answer whenever one exists is called *FSAT*. Further, performing analysis and devising methods for solving the problem is easier, when all the instances are uniform. The most popular form is the *conjunctive normal form* (CNF). The object of study in [PPZ97] and [PPSZ98], and, consequently, in this report is FSAT where the input formula F is in CNF; in the remainder of this paper, we will use the term *satisfiability problem* to refer to this specific version of the problem. Further, we will be interested in certain subsets of the satisfiability problem. We use the term *k-CNF* to denote the version of the satisfiability problem in which any clause in F contains at most k literals; here, k is a fixed integer value 3, 4, etc. Let n denote the number of variables in F. We interpret the truth assignments to the variables as an n-bit binary string x, i.e., $x \in \{0, 1\}^n$. Further, we use the symbol S to denote the set of all satisfying assignments of F.

Both of the papers are strongly based on the notion of *isolation* of solutions. In formal terms, isolation is defined as follows: For an $x \in S$, i.e., x is a satisfying truth assignment, x is *isolated in direction of variable i*, if (and only if) flipping i results in an assignment z not in S. Further, x is termed *j-isolated* if x is isolated in direction of exactly j variables. Finally, an n-isolated x is termed simply *isolated.* In other words, an isolated solution is isolated in direction of all variables.

Let us assume that x is isolated in direction i. By the definition, flipping the value of i in x leads into an assignment z that is not a solution. Therefore, there must be a clause in F that z fails to satisfy. Further, there must exist at least one clause in F such that a literal with variable i is the only true literal under the assignment x. We term the clause a *critical clause* and use the symbol $C_{(x, i)}$ to refer to it.

3. Key Idea

The intuitive idea in the papers is that instances of the satisfiability problem can be classified into two categories:

- Instances with few, highly isolated solutions
- Instances with many solutions

The reasoning goes as follows: as instances in the second category have many solutions, one of the solutions can be found fairly easily by guessing. A slightly more sophisticated argument is used to show that the instances in the first category can be solved relatively easily as well. The argument is that a highly isolated solution can be succinctly encoded; assuming the instance has an isolated solution, one can be found by looking at the encodings instead of full solutions. The reward from the approach comes from the fact that there are considerably fewer succinct encodings than n-bit binary numbers.

In the next subsection we describe algorithms for coding and decoding satisfying solutions of k-CNF instances. Thereafter, we present a formal statement, Satisfiability Coding Lemma, concerning the code lengths of satisfying solutions.

3.1. Coding Satisfying Solutions

Let us consider a k-CNF formula F, a satisfying solution x of F, and a permutation σ of $\{ 1, ..., n \}$. We define an encoding function Φ_σ as follows:

1. Permute the bits of x according to σ; the result is denoted with $\sigma(x)$
2. For each i ($i = 1, ..., n$), delete the ith bit of $\sigma(x)$ if there is a critical clause $C_{(x,\, \sigma(i))}$ such that the variable $\sigma(i)$ occurs after all the other variables in $C_{(x,\, \sigma(i))}$ according to the ordering σ.
3. $\Phi_\sigma(x)$ is the resulting string

Example 1. We will use a running example to demonstrate the essential concepts. Our example instance is the 4-CNF formula

$$F = \{ \{ x_1, \neg x_2, \neg x_3, x_4 \}, \{ x_2, \neg x_4 \}, \{ \neg x_1, x_3, x_4 \}, \{ x_2, \neg x_3, x_4 \}, \{ x_1, x_3 \}\}.$$

Let us fix a satisfying assignment $x = 1110$, i.e., $x_1 = x_2 = x_3 = true$, $x_4 = false$. Further, we use $\sigma = [4, 3, 2, 1]$. Thus, $\sigma(x)$ is 0111. The following table demonstrates the computing $\Phi_\sigma(x)$.

i	$\sigma(i)$	Critical clause	Delete bit	Remains ($\sigma(x)$)
1	4	-	No	0111
2	3	$\{ \neg x_1, x_3, x_4 \}$	No	0111
3	2	$\{ x_2, \neg x_3, x_4 \}$	Yes	01·0
4	1	$\{ x_1, \neg x_2, \neg x_3, x_4 \}$	Yes	01··

As can be seen from the table above, there is no critical clause for variable $x_{\sigma(1)} = x_4$, and a bit is not deleted. For $x_{\sigma(2)} = x_3$, there is a critical clause, but as 1 appears in σ after 3, the bit is not deleted. Further, there are critical clauses for both $x_{\sigma(3)} = x_2$ and $x_{\sigma(4)} = x_1$, and both the variables occur last among the set of variables in their critical clauses in σ. Therefore, the third and fourth bits are deleted from x. This results in $\Phi_\sigma(x) = 01$. ■

Of course, it is essential that the original solutions can be uncovered from the codings. The following algorithm carries out this task.

```
Algorithm Decode(y: encoded solution, σ: permutation, F: k-CNF formula)
    $F_1 := F$
    for i = 1, ..., n
        if $F_i$ has a clause of length one consisting of a literal with the variable $\sigma(i)$
            then set the variable $\sigma(i)$ to make the clause true;
            else set the variable $\sigma(i)$ to the next unused bit of y
        Let v be the value set for $\sigma(i)$
        $F_{i+1}$ := Substitute($F_i$, $\sigma(i)$, v)
```

The **Substitute** function referred to in the **Decode** algorithm operating on a formula F, variable i, and Boolean value v, is defined as follows:

```
Algorithm Substitute(F: CNF-formula, i: variable, v: Boolean value)
    for each clause c containing literal l where the variable is i
        if l evaluates to true when i is set to v
            then remove c from F
            else remove l from c
    return F
```

In common terms, **Substitute** simplifies the formula by taking advantage of the fact that the value of $\sigma(i)$ has become constant.

By comparing the definition of the encoding function Φ_σ and the decoding algorithm **Decode**, it should not be difficult to see that **Decode** indeed inverses the encoding: in the encoding function, a bit corresponding to a variable is deleted from the encoding exactly when the value of the corresponding variable will be set to make a unit clause true in the decoding algorithm. In other words, the fact that $\sigma(i)$ appears last among the set of variables of a clause in the ordering σ is equivalent to the existence of a unit clause with the variable during the decoding - all the other literals have been removed in the **Substitute** function, as they appear before $\sigma(i)$ in σ. For a detailed proof of this argument, the reader should refer to [PPZ97].

Example 2. We demonstrate the operation of the **Decode** algorithm by reversing the encoding of Example 1. That is, we simulate the call **Decode**(10, [4, 3, 2, 1], F), where F is the same formula as in Example 1 above. The simulated steps are depicted in the table below;

i	$\sigma(i)$	F_i	y	x
1	4	$\{\{x_1, \neg x_2, \neg x_3, x_4\}, \{x_2, \neg x_4\}, \{\neg x_1, x_3, x_4\}, \{x_2, \neg x_3, x_4\}, \{x_1, x_3\}\}$	**0**1	···0
2	3	$\{\{x_1, \neg x_2, \neg x_3\}, \{\neg x_1, x_3\}, \{x_2, \neg x_3\}, \{x_1, x_3\}\}$	**1**	··10
3	2	$\{\{x_1, \neg x_2\}, \{x_2\}\}$	e	·110
4	1	$\{\{x_1\}\}$	e	1110

First, observe that the recovered assignment is indeed the same assignment that was encoded above. In the table, the column labelled with y contains the unused bits left in y; e signals that there are no more unused bits. Further, the column labelled with x demonstrates the bits covered from the solution so far. ■

3.2. Satisfiability Coding Lemma

As can be recalled from above, the idea underlying these papers is that the space of succinct encodings of solutions can be searched more efficiently than the 2^n-space of all the possible assignments. Of course, the more succinct the encodings are, the easier it is to search their space. Therefore, the question is, what can be said about the average (over all permutations) coding length of solutions. The answer comes in the form of the following lemma.

Satisfiability Coding Lemma. If x is a j-isolated satisfying assignment of a k-CNF F, then its average (over all permutations σ) description length under Φ_σ is at most $n - j/k$. ■

The result of the lemma can be understood by remembering that x being j-isolated signals that x has j variables with critical clauses. As k is the maximum number of literals in a clause, there is at least $1/k$ probability that any variable with a critical clause appears last in a random permutation in its critical clause. Therefore, j variables and a probability of at least $1/k$ yield an expected value of at least j/k deleted bits, which directly translates into an average coding length of at most $n - j/k$. For a detailed proof, the interested reader should refer to [PPZ97].

4. Algorithms Based on the Satisfiability Coding Lemma

4.1. A randomised algorithm

The manner in which satisfying solutions are encoded above immediately suggests a randomised algorithm for solving the satisfiability problem. The resulting algorithm is **Search** defined as follows:

```
Algorithm Search($F$: $k$-CNF-formula, $I$: integer)
    repeat $I$ times
        $F' := F$
        while there is an unassigned variable in $F'$
            select an unassigned variable $y$ at random            (1)
            if there is a clause of length one containing the literal $y$ or $\neg y$
                then set $y$ to make that clause true
                else set $y$ to true or false at random            (2)
            Let $v$ be the value set for $y$
            $F'$ := Substitute($F'$, $y$, $v$)
        if the formula is satisfied, then output the assignment
```

The algorithm is parameterised with respect to the number of repetitions I. In [PPZ97], the value of I is fixed to $n^2 2^{n-n/k}$, and this is the value that will be used in the analysis.

One should notice that the algorithm is very similar to **Decode**: the ordering σ is replaced with the randomly chosen variable (in the line marked with (2)), and y, the encoding of a solution, is replaced with randomly chosen truth values (line marked (1)).

Example 3. Let us demonstrate the functionality of **Search** by trying to find a solution for F by using the algorithm. The column labelled Left contains the set of unassigned variables, v the randomly chosen variable, and x the current assignments to variables: randomly chosen bits are typeset in **bold**.

Left	v	F	x
1, 2, 3, 4	4	$\{\{x_1, \neg x_2, \neg x_3, x_4\}, \{x_2, \neg x_4\}, \{\neg x_1, x_3, x_4\}, \{x_2, \neg x_3, x_4\}, \{x_1, x_3\}\}$	···**0**
1, 2, 3	2	$\{\{x_1, \neg x_2, \neg x_3\}, \{\neg x_1, x_3\}, \{x_2, \neg x_3\}, \{x_1, x_3\}\}$	·**0**·**0**
1, 3	3	$\{\{\neg x_1, x_3\}, \{\neg x_3\}, \{x_1, x_3\}\}$	·**000**
1	1	$\{\{\neg x_1\}, \{x_1\}\}$	1**000**
-	-	$\{\perp\}$	1**000**

Thus, no solution was found with the random choices made.

We give another example of executing the while-loop; this time, we will assume that we will randomly select the permutation and random bit values that we know will satisfy the formula.

Left	v	F	x
1, 2, 3, 4	4	$\{\{x_1, \neg x_2, \neg x_3, x_4\}, \{x_2, \neg x_4\}, \{\neg x_1, x_3, x_4\}, \{x_2, \neg x_3, x_4\}, \{x_1, x_3\}\}$	···**0**
1, 2, 3	3	$\{\{x_1, \neg x_2, \neg x_3\}, \{\neg x_1, x_3\}, \{x_2, \neg x_3\}, \{x_1, x_3\}\}$	··**10**
1, 2	2	$\{\{x_1, \neg x_2\}, \{x_2\}\}$	·1**10**
1	1	$\{\{x_1\}\}$	11**10**
-	-	$\varnothing$	11**10**

By comparing the above table and that in **Example 2**, the pair of tables can be found to be very similar. ■

What we would like to show about the algorithm is that it finds a satisfying assignment, assuming there is one, with a probability that can be made arbitrarily close to one by increasing the value of I. Further, the running time should be in $2^{(1-\varepsilon)n}$ for some $\varepsilon > 0$.

It can be easily seen that the running time of the algorithm is in $I|F|$, and with the fixed value of I, the time is $n^2 2^{n-n/k}|F|$. What remains to be shown is that the algorithm succeeds in returning a satisfying assignment with a high probability. The fact that the solution is assumed to be j-isolated helps here: the larger j is, the more there will be clauses of length one in the algorithm, and less bits must be guessed (line (2)).

By the Satisfiability Coding Lemma, we know that on average, for a j-isolated solution at least j/k variables appear last in their clauses. As there can be no more than n such variables, there must be a $1/n$ fraction of permutations where there are at least j/k such variables: otherwise, the average could not be j/k. Further, assuming that we have hit a permutation in the $1/n$ fraction, we only need to guess $2^{n-j/k}$ variables. Thereby, the total probability for one iteration of the while loop finding a solution is at least the product of $1/n$ and $2^{n-j/k}$, i.e., $n^{-1}2^{-n+j/k}$.

It can be shown that the probability of finding some satisfying truth assignment by executing the while-loop once is $n^{-1}2^{n-n/k}$. The probability is higher, or at least as high as than finding a specific, j-isolated solution. Intuitively, it is clear that finding one of many solutions is more likely than finding a specific one; on the other hand, if there is only one solution, it must be isolated, then $j = n$, and the probability of finding the specific solution is equal to finding any solution. For a detailed analysis, the reader should refer to [PPZ97].

Finally, as there is an $n^{-1}2^{n-n/k}$ probability for finding some solution with a single execution of the while-loop, it is obvious that repeating the loop $n^2 2^{n-n/k}$ times gives a constant probability for success, and the constant can be made arbitrarily close to one.

4.2. A Deterministic Algorithm

The same ideas that are used in the **Search** algorithm above can be employed in a deterministic algorithm as well. However, the details of a deterministic algorithm are not obvious: the primitive algorithm would go through the entire space of succinct encodings, but as a succinct encoding can have length n, the size of this space would be 2^n, and there would be no improvement to the naïve strategy of systematically testing all the possible solutions.

The deterministic algorithm shown in [PPZ97] uses an interesting strategy to reduce the size of the search space. The deterministic algorithm **SearchDet** is defined as follows.

```
Algorithm SearchDet(F: k-CNF-formula, I: integer)
    for all inputs $x$ with at most $\varepsilon n$ ones
        if $F$ is satisfied by $x$, then output $x$
    for all permutations $\sigma$ in $\psi$, and strings $s$ of $n(1-\varepsilon/k) + 1$ bits
        $x$ = Decode($s$, $\sigma$, $F$)
        if $x$ satisfies $F$, then output $x$
```

In the above algorithm, ε can be considered to be a relatively small number compared with n. The set of permutations that must be searched is ψ and has the property that for any set Y of up to k variables, the probability that for a random permutation in ψ, y appears last among the variables in Y is at least $1/|Y| - 1/n$. In terms of k-CNF instances, each variable will appear last among the variables of its clause with the probability $1/k - 1/n$. For information on constructing ψ and other details, the reader should refer to [PPZ97].

Intuitively, **SearchDet** is based on the idea that a formula either has a solution with few ones, or any minimal solution has many ones; a minimal solution is a solution such that no solution has fewer ones. The first for loop tries to find a solution with few ones. The second for loop, in turn, takes advantage of the idea that a minimal solution must be isolated in all the directions where the variable has the value one.

The complexity of **SearchDet** is $O(2^{n-n/2k})$ for large n and k. Compared with the randomised variant **Search**, there is a factor two in front of k. Consequently, the deterministic variant has inferior time complexity when compared with the randomised variant, for all values of k. On the other hand, **SearchDet** can, unlike **Search**, be used to see that a formula is not satisfiable with certainty.

5. Resolution Enhancement

The efficiency of the **Search** algorithm above is based on the intuitive fact that a formula has either many solutions, or a few isolated solutions. An interesting question is, would there be a way of modifying k-CNF formulas in such a way that the solutions could be more easily found by the **Search** algorithm. In [PPSZ98] it turns out that there is a way to achieve this, namely *bounded resolution*.

In the following subsections, we will first define bounded resolution, and thereafter argument why the **Search** algorithm gives better results on formulas to which resolution has been applied.

5.1. Bounded Resolution

First, we say that clauses C_1 and C_2 are in conflict if one of them contains literal v and the other literal $\neg v$. Further, we term C_1 and C_2 a *resolvable pair* if they conflict on exactly one variable. The *resolvent* C_1 and C_2 is the clause $C = D_1 \vee D_2$, where D_i is obtained from C_i by removing the literal with variable v. Finally, the resolvable pair is said to be *s-bounded* if $|R(C_1, C_2)| \leq s$.

The following algorithm **Resolve** implements bounded resolution for a k-CNF formula.

```
**Algorithm** **Resolve**($F$: CNF-formula, $s$: integer)
    $F_s := F$
    **while** $F_s$ has an $s$-bounded resolvable pair $C_1$, $C_2$ with $R(C_1, C_2) \notin F_s$
        $F_s := F_s \wedge R(C_1, C_2)$
    **return** $F_s$
```

Example 4. Let us demonstrate the effect of resolve on the CNF-formula we have used as an example. With this formula, the bound s is not really of significance: F already contains clauses where all four variables occur, and the resolution cannot in any circumstances produce clauses longer than n.

The resolve algorithm results in the following set of clauses F_s:

$\{$ $\boldsymbol{\{ x_1, \neg x_2, \neg x_3, x_4 \}}$, $\boldsymbol{\{ x_2, \neg x_4 \}}$, $\boldsymbol{\{ \neg x_1, x_3, x_4 \}}$, $\boldsymbol{\{ x_2, \neg x_3, x_4 \}}$, $\boldsymbol{\{ x_1, x_3 \}}$, $\{ x_1, \neg x_3, x_4 \}$, $\{ x_1, \neg x_2, x_4 \}$, $\{ \neg x_1, x_2, x_3 \}$, $\{ x_2, \neg x_3 \}$, $\{ x_1, x_2, \neg x_3 \}$, $\{ \neg x_1, x_2, x_4 \}$, $\{ x_3, x_4 \}$, $\{ \neg x_2, x_3, x_4 \}$, $\{ x_1, x_2, x_4 \}$, $\{ x_2, x_4 \}$, $\{ x_1, x_4 \}$, $\{ x_2, x_3 \}$, $\{ x_1, x_2 \}$, $\{ x_2, x_3, x_4 \}$, $\{ x_1, x_3, x_4 \}$, $\{ \neg x_1, x_2 \}$, $\{ x_2 \}$, $\{ x_1, x_2, x_3 \}$ $\}$

The original five clauses are typeset in **bold**. There are altogether 23 clauses, the resolution produced 18 new clauses. For instance, the clause $\{ x_1, \neg x_3, x_4 \}$ is the resolvent of $\{ x_1, \neg x_2, \neg x_3, x_4 \}$ and $\{ x_2, \neg x_3, x_4 \}$, and $\{ x_1, x_2, \neg x_3 \}$ of $\{ x_2, \neg x_4 \}$ and $\{ x_1, \neg x_3, x_4 \}$. ■

The above algorithm can be used in conjunction with the **Search** algorithm. The algorithm **ResolveSat** does this:

```
**Algorithm** ResolveSat($F$: CNF-formula, $s$: integer)
    $F_s :=$ **Resolve**($F$, $s$)
    **Search**($F_s$, $I$)
```

Example 5. The effect of resolution can be demonstrated by seeing if the randomly selected permutations and bits would have yielded a satisfying solution for the resolved version of F.

Left	v	F	x
1, 2, 3, 4	4	$\{ \{ x_1, \neg x_2, \neg x_3, x_4 \}, \{ x_2, \neg x_4 \}, \{ \neg x_1, x_3, x_4 \}, \{ x_2, \neg x_3, x_4 \}, \{ x_1, x_3 \}, \{ x_1, \neg x_3, x_4 \}, \{ x_1, \neg x_2, x_4 \}, \{ \neg x_1, x_2, x_3 \}, \{ x_2, \neg x_3 \}, \{ x_1, x_2, \neg x_3 \}, \{ \neg x_1, x_2, x_4 \}, \{ x_3, x_4 \}, \{ \neg x_2, x_3, x_4 \}, \{ x_1, x_2, x_4 \}, \{ x_2, x_4 \}, \{ x_1, x_4 \}, \{ x_2, x_3 \}, \{ x_1, x_2 \}, \{ x_2, x_3, x_4 \}, \{ x_1, x_3, x_4 \}, \{ \neg x_1, x_2 \}, \{ x_2 \}, \{ x_1, x_2, x_3 \} \}$	···**0**
1, 2, 3	2	$\{ \{ x_1, \neg x_2, \neg x_3 \}, \{ \neg x_1, x_3 \}, \{ x_2, \neg x_3 \}, \{ x_1, x_3 \}, \{ x_1, \neg x_3 \}, \{ x_1, \neg x_2 \}, \{ \neg x_1, x_2, x_3 \}, \{ x_1, x_2, \neg x_3 \}, \{ \neg x_1, x_2 \}, \{ x_3 \}, \{ \neg x_2, x_3 \}, \{ x_1, x_2 \}, \{ x_2 \}, \{ x_1 \}, \{ x_2, x_3 \}, \{ x_2, x_3 \}, \{ x_1, x_2, x_3 \} \}$	·1·**0**
1, 3	3	$\{ \{ x_1, \neg x_3 \}, \{ \neg x_1, x_3 \}, \{ x_1 \}, \{ x_3 \}, \{ x_1, x_3 \} \}$	·11**0**
1	1	$\{ \{ x_1 \} \}$	111**0**
-	-	$\varnothing$	111**0**

As can be seen, a solution is found. Further, only one randomly generated variable is needed; the values of all the other variables are determined by unit clauses. ■

5.2. Why Does Resolution Improve the Algorithm

Above, it is stated that the resolution increases the amount of critical clauses a variable has for a given solution x and a permutation σ. The critical clauses, in turn, increase the probability of guessing a satisfying assignment, as there is less need for guessing bits. In this section, we give an overview of *critical clause trees* that are the mechanism for quantifying the amount of critical clauses. However, we will not go into details about the exact number of critical clauses, or the effect the increased number of critical clauses have on the probability of finding a solution: although very interesting from the theoretical point of view, critical clause trees do not have a direct impact on the **ResolveSat** algorithm, beyond the number of iterations needed. Therefore, the details of the analysis are best omitted.

The authors use a tree construction for characterising the critical clauses. A *cut* of a tree is a set of nodes of the tree such that:

- It does not include the root of the tree.
- Every path from the root to a leaf includes a node of A.

Intuitively, it should be clear that a tree has many cuts. Further, we define a *critical clause tree* for variable v, formula G, and satisfying assignment z as follows:

- Each node in the tree is either labelled by a variable or unlabelled.
- For any path P from the root to a leaf, no two nodes have the same label. In other words, if node a is an ancestor of node b and both are labelled, then they have different labels.
- The root label is v.
- For any cut A of the tree, G has a critical clause $C_{(z,\, i)}$ such that the set of variables occurring in $C_{(z,\, i)}$ is a subset of the union of the labels of A and v.

The effect of resolution is conceptualised by the following lemma:

Lemma. Let F be a k-CNF formula and z be a d-isolated satisfying assignment of F. If v is any variable then for any $s \geq k^d$, there exists a critical clause tree for the variable v, assignment z and formula F of depth d and maximum degree k - 1. ■

To summarise, the resolution causes each sufficiently isolated solution to have many critical clauses with respect to all the variables. Critical clause trees guarantee that there exists a specific amount of critical clauses for each variable-solution pair; the above lemma formalises this. The increased number critical clauses causes a decrease in the amount of bits that must be guessed correctly in order to find a satisfying solution; the pair of examples Example 3 and Example 5 concretises this: The same random inputs were used in both of the examples. In Example 3 two bits were guessed, but no satisfying solution was found. In Example 5, only one bit was guessed, and a satisfying solution was found. Less need for guessing bits implies that a satisfying assignment can be expected to be found more easily, and a smaller amount of iterations is needed in order to find a satisfying solution with probability approaching one.

The number of iterations needed in order for ResolveSat to find a solution to a k-CNF instance with a probability approaching one is:

$$I = 2^{(1-\frac{\mu_k}{k-1})n+o(n)}, \text{ for } k \geq 5$$

In the above formula, μ_k is defined as:

$$\mu_k = \sum_{j=1}^{\infty} \frac{1}{j(j+\frac{1}{k-1})}$$

As k approaches infinity, the value μ_k approaches 1.644. Thereby, it is easy to see that as k gets large, the number of iterations I needed for **ResolveSat** approaches the number 2^n. For $k = 3, 4$, a smaller number of iterations suffices, which results in faster running times. These resulting running times, and those for $k = 5, 6$ are summarised in Table 1.

When considering the results, it is worthwhile to assess whether the resolution strategy could be applied in practise. First, the **Resolve** procedure can produce at most n^s new clauses; the actual amount of clauses produced can be in the same order of magnitude. As s should be a large constant or a function of n tending to infinity with n, the amount of memory required to store the resulting clauses can become a problem when solving large instances. Further, the running time of **Resolve** can be as large as $O(n^{2s})$. Albeit this is not an exponential time complexity, in a practical setting an n^{2s} algorithm with, say $s = 100$, can be rather slow: alhtough for a sufficiently large n, the exponential part of the algorithm will always dominate the polynomial part, it can well be that this does not happen for any n such that instances with problem size n could be solved. Therefore, the advantages in theoretical complexity achieved through the resolution may not be easily converted into efficient implementations.

6. Conclusions

We have presented an overview of an approach to solving the satisfiability problem. The approach is based on rigorous observations about the solution space of the satisfiability problem. Although the papers in which the approach is introduced stem from years 1997 and 1998, the results achieved seem not to be hopelessly outdated: significantly better results have been achieved at least in [HSSW02], but we do not know of other papers where lower theoretical bounds would have been achieved.

The time complexities of the algorithms for various values of k are presented in Table 1:

Table 1 Running times of algorithms Search and ResolveSat for different values of k

k	**Search**	**ResolveSat**
3	$2^{0.667n}$	$2^{0.446n}$
4	$2^{0.750n}$	$2^{0.562n}$
5	$2^{0.800n}$	$2^{0.650n}$
6	$2^{0.833n}$	$2^{0.711n}$

For a uniquely satisfiable 3-CNF, the running time with the resolution-enhanced algorithm is $2^{0.387n}$. This is still the best value known; in [HSSW02], the corresponding complexity is 1.3302^n, which translates into $2^{0.412n}$.

Given any theoretical description of an algorithm, and interesting question is: can the algorithm be efficiently implemented. It is known to us that at least UnitWalk (see, e.g., [HK01]) uses the ideas summarised in this paper, but is not solely based on them. UnitWalk did reasonably well in the SAT 2002 competition [SBH02].

References

[HK01] E. A. Hirsch, A. Kojevnikov. UnitWalk: A new SAT solver that uses local search guided by unit clause elimination, PDMI preprint 09/2001, Steklov Institute of Mathematics, St. Petersburg, Russia.

[HSSW02] T. Hofmeister, U. Schöning, R. Schuler, O. Watanabe. A probabilistic 3-SAT algorithm further improved", Proceedings of the 19th Symposium on Theoretical Aspects of Computer Science, Lecture Notes in Computer Science 2285, 2002, pages: 192-202

[PPZ97] R. Paturi, P. Pudlak, F. Zane. Satisfiability Coding Lemma, Proceedings of the 38th Annual Symposium on Foundations of Computer Science, 1997, pages: 566-574

[PPSZ98] R. Paturi, P. Pudlak, M. E. Saks, F. Zane. An improved exponential-time algorithm for *k*-SAT, Proceedings of the 39th Annual Symposium on Foundations of Computer Science, 1998, Pages: 628-637.

[SBH02] L. Simon, D. Le Berre, E. A. Hirsch. The SAT2002 Competition (preliminary draft). Available at <http://www.satlive.org>. 2002