Polynomial Time Approximations toward Counting Satisfiability

Prabin Amatya, Stefan Andrei, Kami Makki

No. 2, June 2010

Computer Science Department Technical Reports - Lamar University
Abstract

The Boolean Satisfiability (SAT) is the problem of determining the state of the propositional variables such that they evaluate the formula to TRUE. It is also important that we determine whether no such assignments exist, which imply that the propositional formula expressed by the formula is identically FALSE for all possible variable assignments. This later propositional formula is called unsatisfiable, otherwise it is satisfiable. Counting satisfiability is a valuable approach for problems like constraint satisfaction, knowledge compilation, probabilistic reasoning, and computing the permanent of a Boolean matrix. The difficulty with counting satisfiability is that it is as hard as NP-complete problems, but probably much harder. This means that solvers for counting the exact number of solutions may need a large amount of time on large propositional formulas. In this paper, we provide a fast alternative by approximating the number of instances. Our technique is based on successive variable and clause eliminations. Comparison with existing counting SAT solvers demonstrate that these new algorithms for computing the lower and upper bounds approximations are promising.

1. Introduction

The SAT problem means determining whether a propositional formula $F$ has a truth assignment (a.k.a., $F$ is satisfiable). Cook proved in 1971 that the SAT problem is NP-complete [3]. There are many applications of the #SAT problem in the areas of computer science, including artificial intelligence, inductive logic programming, real-time systems, electronic design automation, formal languages, and so on. As mentioned in [6], counting is often the most natural way of verifying equivalence between two theories. Moreover, it can provide degrees of how close is a theory by its approximation.

The user will provide a propositional formula in conjunctive normal form (conjunction of clauses, where a clause is a disjunction of literals) as input to our algorithm and convert it into a SAT encoding expressed in the so-called DIMACS format. A DIMACS format is represented by a .cnf file that stores boolean clauses in Conjunctive Normal Form. It represents the accepted format for state-of-the-art SAT solvers competition. This file has three parts: the comments part represented by lines of text which start with c followed by comments, the second part which starts with p cnf <number_of_variables> <number_of_clauses>, followed by the third part formed by the representation of the boolean clauses.

Cachet [7] was introduced as a combination of formula caching, clause learning, and dynamic component analysis. A more efficient version of Cachet was described in [8]. Compared with the previous version of Cachet, the new algorithm includes component-selection strategies, variable-selection branching heuristics, backtracking schemes, and randomization.

This paper continues the approaches from two papers [1] and [2]. It provides an alternative of existing counting SAT solvers by doing a fast approximation of the number of instances. The main idea is to provide a lower bound and an upper bound approximation for the number of truth assignments using a polynomial time algorithm. A recent paper describes, SampleCount [9], a tool that uses sampling and exact model counters to approximate the number of instances. This differs from our approach since our focus is on polynomial time algorithms for approximating the number of truth assignments.
2. Algorithm and Methodology

A non-trivial and different subclasses of sub-formulas (called Rank 1) for which the SAT problem can be solved in polynomial time, namely \(O(n \times l^2)\), where \(n\) and \(l\) are the number of variables, clauses, respectively. For example, \(F_1 = (p \lor q) \land (\neg q \lor r)\), is a rank 1 formula (note that \(\neg q\) means the negative literal of variable \(q\)), while \(F_2 = (p \lor q) \land (q \lor r)\) is not a rank 1 formula, although both \(F_1\) and \(F_2\) contain the clause \((p \lor q)\). In this way, we partition the class of NP-complete problems from the SAT Dichotomy Theorem by identifying the subclasses of rank \(k\) formulas in the P class.

For a finite set \(A\), \(|A|\) denotes the number of elements of \(A\). \(Z, N\) and \(N^+\) denote the set of integers, positive integers, and the set of strict positive integers, respectively [1]. Let \(C_1, \ldots, C_s\) be clauses over \(V\) \((s \geq 1)\). We denote:

\[
\begin{align*}
& a) \ m_V(C_1, \ldots, C_s) = |\{A \mid A \in V - (C_1 \cup \ldots \cup C_s)\}|; \\
& b) \ dif_V(C_1, \ldots, C_s) = \\
& \quad b1) \ 0 \text{ if } (\exists i, j \in \{1, \ldots, s\}, i \neq j, \text{ such that } \exists L \in C_i \text{ and } \neg L \in C_j) \text{ or } \\
& \quad \quad \text{if } (\exists i \in \{1, \ldots, s\}, \text{ such that } C_i = \text{null}); \\
& \quad b2) \ 2^{m_V(C_1, \ldots, C_s)} \text{ otherwise};
\end{align*}
\]

In other words, the above notation says that \(m_V(C_1, \ldots, C_s)\) denotes the number of atomic formulae from \(V\) which do not occur in \(C_1 \cup \ldots \cup C_s\). Obviously, \(0 \leq m_V(C_1, \ldots, C_s) \leq |V|\), the margin cases are obtained when \(C_1 \cup \ldots \cup C_s\) is a maximal clause and the empty clause, respectively. The positive integer number \(dif_V(C_1, \ldots, C_s)\) is 0 if there is a literal in one of the argument’s clauses and its opposite in another clause or one of the clauses is the empty one. Otherwise, \(dif_V(C_1, \ldots, C_s)\) equals to 2 to the power of \(m_V(C_1, \ldots, C_s)\). A propositional formula \(F = \{C_1, \ldots, C_s\}\) over alphabet \(V\) has rank 1 if \(\text{dif}_V(C_i, C_j) = 0\), for all \(i, j \in \{1, \ldots, s\}, i \neq j\).

Finally, the positive integer number \(det_V(C_1, \ldots, C_s)\) is a sign alternating sum of \(\text{dif}_V()\). The complete formula can be found in [1] and this is \(det_V(C_1, \ldots, C_s) = 2^{|V|} - \sum_{1 \leq l < \ldots < \ell \leq s} \text{dif}_V(C_{l_1}, \ldots, C_{l_\ell})\). If \(F = \{C_1, \ldots, C_s\}\), we also denote \(m_V(C_1, \ldots, C_s)\) as \(m_V(F)\), \(\text{dif}_V(C_1, \ldots, C_s)\) as \(\text{dif}_V(F)\) and \(det_V(C_1, \ldots, C_s)\) as \(det_V(F)\). When there is no ambiguity of the usage of \(V\), then \(V\) can be omitted.

Next, we provide a unified description of Algorithm A [1] and Algorithm B [2] from our previous works.

**Unification of Algorithms A and B**

The **input**: A clausal formula \(F = \{C_1, \ldots, C_s\}\) over \(V = \{A_1, \ldots, A_n\}\).

The **output**: Two positive integers \(lb\) and \(ub\) such that \(lb \leq \text{det}_V(F) \leq ub\).

The **method**:

```java
boolean existNonRank1Clauses = true;
stop = false;
while (existNonRank1Clauses && !stop) {
    existNonRank1Clauses = false;
    calculate dif(C_i, C_j), \forall i \in \{1, \ldots, l\}, \forall j \in \{1, \ldots, l\};
    if (all dif(C_i, C_j) == 0) {
        lb = det(F_lower);
        ub = det(F_upper);
        stop = true;
    } else {
```

-let $F_{\text{lower}}$ be the propositional formula obtained from $F$ based on the lowest $\text{dif}(C_i, C_j)$ or $\text{score}(L)$ criteria by removing on literal and its corresponding clauses; 

-let $F_{\text{upper}}$ be the propositional formula obtained from $F$ based on the highest $\text{dif}(C_i, C_j)$ or $(u, t)$ comparison criteria like the empty clause rule, the unit clause rule or pure literal rule may also be applied.

From this unification, we create two separate refinements, one for estimating the lower bound of the determinant and the other one for the upper bound of the determinant. Next, we describe the refinement of Algorithm A from [1].

**Algorithm A Refinement (Lower Approximation)**

**The input:** A clausal formula $F = \{C_1, \ldots, C_l\}$ over $V = \{A_1, \ldots, A_n\}$.

**The output:** A positive integer $lb$ such that $\det_V(F) \geq lb$.

**The method:**

```java
boolean existDiffNonZero = true, stop = false;
boolean column[n], row[l];
Initialize column[n] and row[l] as true;
while (existDiffNonZero && !stop) {
    // stop = true iff the lb was found without requesting formula be rank 1
    existDiffNonZero = false;
    for (int i = 0; i < l - 1 && !stop; i++)
        for (int j = i + 1; j < l && !stop; j++) {
            if (($\text{dif}(C_i, C_j) > 0)$ && row[i] && row[j]) {
                // i and j represent indexes of non Rank 1 clauses
                existDiffNonZero = true;
                if ($\exists C_i = \{L\}$)
                    if ($\exists C_j = \{\neg L\}$) { lb := 0; stop = true; }
                else {
                    // there is no $C_j = \{\neg L\}$
                    F is:{$\{L\}, C_{i1} \cup \{L\}, \ldots, C_{is} \cup \{L\}, C_{j1} \cup \{\neg L\}, \ldots, C_{jt} \cup \{\neg L\}, C_{kt}, \ldots, C_{ku}$};
                    if ($u > 0$) {
                        // u represents the no. of clauses that don't contain L and $\neg L$
                        F := $\{C_{j1}, \ldots, C_{jt}, C_{kt}, \ldots, C_{ku}\}$;
                        V := $V \setminus V(L)$;
                    }
                    else {
                        // t represents the no. of clauses that contain the $\neg L$
                        if (t == 0) { lb := $2^{|V|}$; stop = true; }
                        else { F := $\{C_{j1}, \ldots, C_{jt}\}$; V := $V \setminus V(L)$; }
                    }
                }
            }
        }
    else {
        Calculate $\text{score}(L) = \text{no. of clauses that contain } L \text{ and the binary clauses that contain } \neg L$
        L will be the highest score.
        int variableToBeRemoved = location % n;
        column[variableToBeRemoved] = false;
        for (k = 0 ; k < l ; k++) {
            if (row[k]) {
                if (c[k][ variableToBeRemoved ] ==1 && location < n)
```
\begin{verbatim}
row[k] = false;
if (c[k][ variableToBeRemoved] == -1 && location > n)
    row[k] = false;
} // end of if
} //end of for
} //end of else branch
} //end of if (dif(Ci, Cj)
} //end of for
} //end of while
if (!stop) lb = det(F);
\end{verbatim}

To refine Algorithm A we implemented two different measures: namely the difference of variables and score. Here the while statement of the above algorithm is due to the fact that at every iteration the clausal forunal reduces its size, by eliminating the unit clauses or by transforming non-unit clauses into unit clauses. The while statement can terminate even earlier if two opposite unit clauses are detected (so that 0 is the returned value in this case). Since there might be at most O(l) iterations of the while statement, and the statements of the while’s body can be done in O(n * l) time complexity, it follows that the algorithm has O(n * l^2) time complexity, where n is the number of variables and l is the number of clauses.

If (t == 0) then the lower bound is evaluated to 2 to the cardinal power of number of variables. If (t != 0) formula is evaluated and corresponding column is removed as well. After difference of variables between two clauses is evaluated as 0 after all the manipulation like checking of existence of literal clause otherwise reduce clause in such a way that it becomes a unit clause. Score of individual literals are calculated, and depending on the highest score the corresponding rows and columns are removed implicitly by placing corresponding row and column values as false. This process is continued until there is no more high score left. After that, we calculate the lower bound value from the remaining columns and rows.

**Algorithm B improvement (Upper Approximation)**

**Input:** F = \{C_1, \ldots, C_l\} a propositional formula, where l \geq 2;

**Output:** F’ = \{C'_1, \ldots, C'_l\} a rank 1 propositional formula such that det_v(F) \leq det_v(F’);

**Method:**
Evaluate and update difference of variables between all pairs of clauses.

```java
boolean existDiffNonZero = true;
while (existDiffNonZero) {
    existDiffNonZero = false;
    int max = 0;
    int i_index, j_index;
    for (i = 1; i < l; i + +)
        for (j = i + 1; j <= l; j + +) {
            int diff_i_j = dif(C_i, C_j);
            if (diff_i_j > max) {
                max = diff_i_j;
                i_index = i;
                j_index = j;
            }
            existDiffNonZero = true;
        }
    if (existDiffNonZero) {
        // determine the highest diff_v and remove the corresponding horizontal clauses and vertical variables.
        if (diff_i_j > max) {
            max = diff_i_j;
            i_index = i;
            j_index = j;
        }
        existDiffNonZero = true;
    }
    if (!stop) lb = det(F);
} //end of while
```
let u be number of literals of \( C_{i\_index} \) which do not appear in \( C_{j\_index} \)
let t be number of literals of \( C_{j\_index} \) which do not appear in \( C_{i\_index} \)
if \((u \geq t)\) {
    if \((u > 0)\) // right extension
        \( C_{i\_index} = C_{j\_index} \cup \{\neg L\} \), where \( L \in C_{j\_index} \) and \( L, \neg L \notin C_{i\_index} \);
    else \{// clauses \( C_{i\_index} \) and \( C_{j\_index} \) are equal\}
        \)// \( u \geq t \)
    else {
        if \((t > 0)\) // left extension
            \( C_{i\_index} = C_{i\_index} \cup \{\neg L\} \), where \( L \in C_{j\_index} \) and \( L, \neg L \notin C_{i\_index} \);
        else \{// clauses \( C_{i\_index} \) and \( C_{j\_index} \) are equal\}
            \)// from if \( \text{existDiffNonZero} \)
        \}// from for \((j = i + 1; j <= l; j + +)\)
    } // end of while
\ub = \text{det}(F), \text{where } F \text{ is updated in previous statements}

In Algorithm B above, the difference of variables between two clauses is evaluated until all of them equate to zero. Here during evaluation of corresponding rows and columns, we eliminate the corresponding row and column with the highest difference variables. This elimination is continued until all of them are equated as zero.

We will focus only on the rank 1 class of formulae because this corresponds to the lowest time-complexity among the hierarchy of rank \( k \) formulae. In order to find a computationally efficient good upper approximation, we define the left and right extensions for clauses. A good approximation refers to one that is able to achieve the following [2]:

a. the approximation is obtained by an efficient algorithm.
b. it should provide a small determinant.

If the difference of variable is 0 we evaluate ‘\( u \)’ and ‘\( t \)’ values. Here ‘\( u \)’ values represent the number of literals of \( C_{i\_index} \) which do not appear in \( C_{j\_index} \), while ‘\( t \)’ is the number of literals of \( C_{j\_index} \) which do not appear in \( C_{i\_index} \).

If \((u \geq t)\) we process for right or left extension. If \((u > 0)\) we do the right extension else we do the left extension.

Let us consider \( C_i \) and \( C_j \) two arbitrary clauses such that \( i < j \) and \( \text{dif}(C_i, C_j) \neq 0 \). By left extension we refer to the case \( C_j \subset C_i \). The case \( C_j \subseteq C_i \) can be solved immediately by removing \( C_i \). Since \( C_j \subset C_i \), we get that there exists a literal \( L \) from \( C_j \) which does not occur in \( C_i \). Then the clause \( C_j' = C_i \cap \{\neg L\} \) is an approximation of \( C_i \) for which \( \text{dif}(C_j', C_j) = 0 \) [2].

Similarly we define right extension, by referring to the case when \( C_i \subset C_j \). It follows that there exists a literal \( L \) from \( C_i \) which does not occur in \( C_j \). Then the clause \( C_i' = C_j \cap \{\neg L\} \) is an approximation of \( C_j \) for which \( \text{dif}(C_i', C_j) = 0 \) [2]. If two clauses cannot be extended either by left extension or by right extension, then they coincide (so one of them can be removed from the set of initial clauses).

Here we provided an alternative to existing counting SAT solvers by doing a fast approximation of the number of instances. Experimental results demonstrate that our technique is promising, especially for applications where efficiency is more important than the precision, i.e., the exact number of truth assignments.

3. Implementation
This tool is implemented in Java Netbeans Integrated Development Environment (IDE) Version 6.9. It is built in a procedural way without any other classes because all the methods are highly related to each other.
Figure 1. A UML diagram of LUANTA

4. Experimental results

We present preliminary experimental results. We use an implementation of our algorithm in Java called LUANTA (Lower and Upper Approximation of Number of Truth Assignments). It can be downloaded freely from [10]. We compare LUANTA against Cachet, one of the most efficient exact counting solvers. Table 1 shows the execution times of Cachet, LEBACS (LowEr Bound Approximation Counting Satisfiability), and LUANTA for randomly generated propositional formulas, written in DIMACS format. None of them is a rank 1 formula. Here we run and measure the execution times of all these tools on the same module, that is Intel i7 CPU @2.67 Ghz, 8GB RAM. The LEBACS and LUANTA were run on Windows 7 professional operating system, while Cachet was run in Ubuntu 10.04 operating system. In addition we compare our tool with Cachet and LEBACS[1]. We used NetBeans IDE 6.8 to create these programs.
Table 1. Comparison between Cachet, LEBACS, and LUANTA

<table>
<thead>
<tr>
<th>S.No.</th>
<th>Input</th>
<th>Var</th>
<th>Cls</th>
<th>Sols</th>
<th>Time-sec</th>
<th>Sols</th>
<th>Time-sec</th>
<th>Lb</th>
<th>Time-sec</th>
<th>Ub</th>
<th>Time-sec</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5-101.cnf</td>
<td>5</td>
<td>10</td>
<td>0</td>
<td>0.002</td>
<td>0</td>
<td>0.001</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>6-32.cnf</td>
<td>6</td>
<td>32</td>
<td>1</td>
<td>0.24</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>8-37.cnf</td>
<td>8</td>
<td>37</td>
<td>87</td>
<td>0.023</td>
<td>87</td>
<td>0.003</td>
<td>0</td>
<td>0</td>
<td>142</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>9-103.cnf</td>
<td>9</td>
<td>103</td>
<td>29</td>
<td>0.02</td>
<td>29</td>
<td>0.027</td>
<td>0</td>
<td>0</td>
<td>206</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>13-70.cnf</td>
<td>13</td>
<td>70</td>
<td>6171</td>
<td>0.02</td>
<td>6171</td>
<td>0.005</td>
<td>32</td>
<td>0</td>
<td>7133</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>15-37.cnf</td>
<td>15</td>
<td>37</td>
<td>29217</td>
<td>0.022</td>
<td>29217</td>
<td>0.001</td>
<td>2</td>
<td>0</td>
<td>31128</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>15-57.cnf</td>
<td>15</td>
<td>57</td>
<td>28627</td>
<td>0.022</td>
<td>28627</td>
<td>0.002</td>
<td>0</td>
<td>0</td>
<td>31032</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>15-62.cnf</td>
<td>15</td>
<td>62</td>
<td>28344</td>
<td>0.027</td>
<td>28344</td>
<td>0.003</td>
<td>2</td>
<td>0</td>
<td>30742</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>24-63.cnf</td>
<td>24</td>
<td>63</td>
<td>1.6729E7</td>
<td>0.028</td>
<td>1.6729E7</td>
<td>0.003</td>
<td>16384</td>
<td>0</td>
<td>1.6734E7</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>29-53.cnf</td>
<td>29</td>
<td>53</td>
<td>5.3679E8</td>
<td>0.026</td>
<td>5.3679E8</td>
<td>0.002</td>
<td>128</td>
<td>0</td>
<td>5.367E80</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>30-80.cnf</td>
<td>30</td>
<td>80</td>
<td>1.07355E9</td>
<td>0.027</td>
<td>1.07355E9</td>
<td>0.008</td>
<td>8</td>
<td>0</td>
<td>1.0735E996</td>
<td>0</td>
</tr>
<tr>
<td>12</td>
<td>34-46.cnf</td>
<td>34</td>
<td>46</td>
<td>1.7179E10</td>
<td>0.024</td>
<td>1.7179E10</td>
<td>0.002</td>
<td>1.342E8</td>
<td>0</td>
<td>2.1474E9647</td>
<td>0</td>
</tr>
<tr>
<td>13</td>
<td>35-68.cnf</td>
<td>35</td>
<td>68</td>
<td>3.4358E10</td>
<td>0.03</td>
<td>3.4358E10</td>
<td>0.004</td>
<td>5.368E8</td>
<td>0</td>
<td>2.1474E9</td>
<td>0</td>
</tr>
<tr>
<td>14</td>
<td>35-130.cnf</td>
<td>35</td>
<td>130</td>
<td>3.4356E10</td>
<td>0.04</td>
<td>3.4356E10</td>
<td>0.025</td>
<td>2097152</td>
<td>0</td>
<td>2.1477E9</td>
<td>0</td>
</tr>
<tr>
<td>15</td>
<td>37-46.cnf</td>
<td>37</td>
<td>46</td>
<td>1.3743E11</td>
<td>0.026</td>
<td>1.3743E11</td>
<td>0.002</td>
<td>2048</td>
<td>0</td>
<td>2.1474E9</td>
<td>0</td>
</tr>
<tr>
<td>16</td>
<td>47-146.cnf</td>
<td>47</td>
<td>146</td>
<td>1.4073E14</td>
<td>0.048</td>
<td>1.4073E14</td>
<td>0.028</td>
<td>2.1474508E9</td>
<td>0</td>
<td>2.1474E836</td>
<td>0</td>
</tr>
<tr>
<td>17</td>
<td>54-119.cnf</td>
<td>54</td>
<td>119</td>
<td>1.80E16</td>
<td>0.039</td>
<td>1.80E16</td>
<td>0.015</td>
<td>2.1474795E9</td>
<td>0</td>
<td>2.1474E8364</td>
<td>0</td>
</tr>
</tbody>
</table>

In Table 1, Column 2 is the input that we enter in conjunctive normal form. Var and Cls are the number of variables and clauses. ‘Sols’ and ‘Time-sec’ represent the number of solutions and time in seconds respectively. In addition, ‘Lb’ and ‘Ub’ represent the lower bound and the upper bound of the number of truth assignments, respectively.

5. Conclusion and future work

In this paper, we provided an alternative to existing counting SAT solver by doing a fast approximation of the number of instances. Experimental results demonstrate that our technique is promising, especially for applications where the efficiency is more important than the precision, i.e., the number of truth assignments.

References


10. S. Andrei, P. Amatya. LUANTA. In *http://galaxy.lamar.edu/~sandrei/LUANTA/*
