HOW MANY SOLUTIONS DOES A SAT INSTANCE HAVE?

Pushkin R. Pari, Lin Yuan, and Gang Qu
Electrical and Computer Engineering Department and Institute for Advanced Computer Studies
University of Maryland, College Park, MD 20742 USA

ABSTRACT

Our goal is to investigate the solution space of a given Boolean Satisfiability (SAT) instance. In particular, we are interested in determining the size of the solution space – the number of truth assignments that make the SAT instance true – and finding all such truth assignments, if possible. This apparently hard problem has both theoretical and practical values. We propose an exact algorithm based on exhaustive search that Solves the instance Once and Finds All Solutions (SOFAS) and several sampling techniques that estimate the size of the solution space. SOFAS works better for SAT instances of small size with a 5X-100X speed-up over the brute force search algorithm. The sampling techniques estimate the solution space reasonably well for standard SAT benchmarks.

1. INTRODUCTION

The Boolean satisfiability (SAT) problem seeks to decide, for a given formula, whether there is a truth assignment for its variables that makes the formula true. As the first computational task shown to be *NP-hard*, SAT plays the central role in theoretical computer science and finds numerous applications in various fields. Due to its discrete nature, SAT appears in many contexts in the field of VLSI CAD, such as automatic pattern generation, logic verification, timing analysis, delay fault testing, and channel routing[4].

Over the years, many SAT solvers have been developed based either on *local search* (e.g. GSAT, POSIT, SATO, Satz, WalkSAT, and Rel_SAT) or on *backtrack search* (e.g. GRASP, Chaff, and Zchaff). Links to these solvers can be found at the on-line SAT library [3]. Most of them focus on finding one truth assignment or proving that no such assignment exist. Although some solvers (e.g., GSAT[5]) do provide the option of finding multiple solutions, there is little discussion to the best of our knowledge, on how to obtain all the solutions to a given SAT instance or to determine the size of the solution space. Satometer [1] is the only similar work that estimates the percentage of the search space actually explored by a backtrack SAT solver. Nevertheless, these are important (and of course hard) problems that not only have theoretical value to unveil the structure of the SAT problem, but also can find real life applications, particularly in multiple objective optimization problems. For example, many VLSI CAD applications (such as logic optimization and channel routing) have their SAT formulation and knowing all the solutions gives designer freedom to optimize other design objectives simultaneously.

Naturally, there are two different approaches to finding all the solutions. One is by conducting a brute force search, which evaluates all the possible truth assignments for the variables. This guarantees that all solutions will be found, but the exponential growth of the solution space makes this method impractical. Another method is to repeatedly run a SAT solver until it fails to report any new solutions. The advantage of this approach is that it can find multiple solutions quickly. However, it may miss some solutions particularly, stochastic local search based solvers like GSAT and WalkSAT report unsatisfiable if no solutions are found within a given time.

We propose SOFAS (Solve Once and Find All Solutions) to exhaustively search for all the solutions. SOFAS reports all the solutions, unlike most solvers which tries to find one truth assignment. The basic idea is to scan the SAT instance clause by clause and prune the search space by deleting non-solutions. SOFAS speeds up the solution-pruning process significantly by renaming the variables, reordering the clauses, and carefully managing the solution candidates. The current version of SOFAS can only handle problems of moderate size, but it correctly finds all the solutions and is 5X-100X faster than the pure brute force search. We believe that its performance can be greatly enhanced by adding features such as the Davis Putnam procedure [2] and recursive learning [6, 7].

Enumerating all solutions eventually becomes an insurmountable task as the number of variables increases. Therefore, we propose a couple of sampling techniques to help the process by estimating the size of the solution space. The first method randomly assign values to a set of variables and then tries to determine the size of solution subspace with these variables fixed. The second method runs strategically different SAT solvers to find multiple solutions to the same instance and then compares these solutions to estimate the whole solution space.

We describe SOFAS in Section 2 and discuss the sampling techniques in Section 3. Section 4 reports our preliminary results and Section 5 concludes the paper.

2. SOFAS: SOLVE ONCE AND FIND ALL SOLUTIONS

Figure 1 gives the pseudo code of SOFAS. It reads the clauses one at a time, in a pre-determined order, and eliminates the variable assignment(s) that cannot satisfy this clause. When all the clauses are checked, the remaining assignment(s) are all solutions to the given SAT instance.

Figure 2 depicts the key steps of SOFAS by an example of the following 5-variable SAT formula: $\mathcal{F} = (x_1+x_4'+x_5)(x_3'+x_4+x_5)(x_2+x_4'+x_5')(x_2'+x_3'+x_4)(x_3+x_4'+x_5)$. We represent the solution space by a binary tree where the 2^n leaves, denoted by numbers $0, 1, 2, \cdots, 2^n - 1$ from left to right, correspond to all the possible assignments. At each non-leaf node, we pick an unassigned variable x_i and associate its left subtree with the all assignments with $x_i = 0$ and the right subtree with all assignments with $x_i = 1$. For example, the shaded node at the 4th level in Figure 2(a) represents $\{x_1 = 0, x_2 = 1, x_3 = 1\}$ while variables x_4 and x_5 remain unassigned.

We view each clause as a constraint that eliminates the assignments that fail to satisfy this clause. For instance, any assignment with $\{x_2 = 1, x_3 = 1, x_4 = 0\}$ will make clause $(x_2' + x_3' + x_4)$, and hence the formula $\mathcal{F}$, *FALSE*. Such assignments correspond to

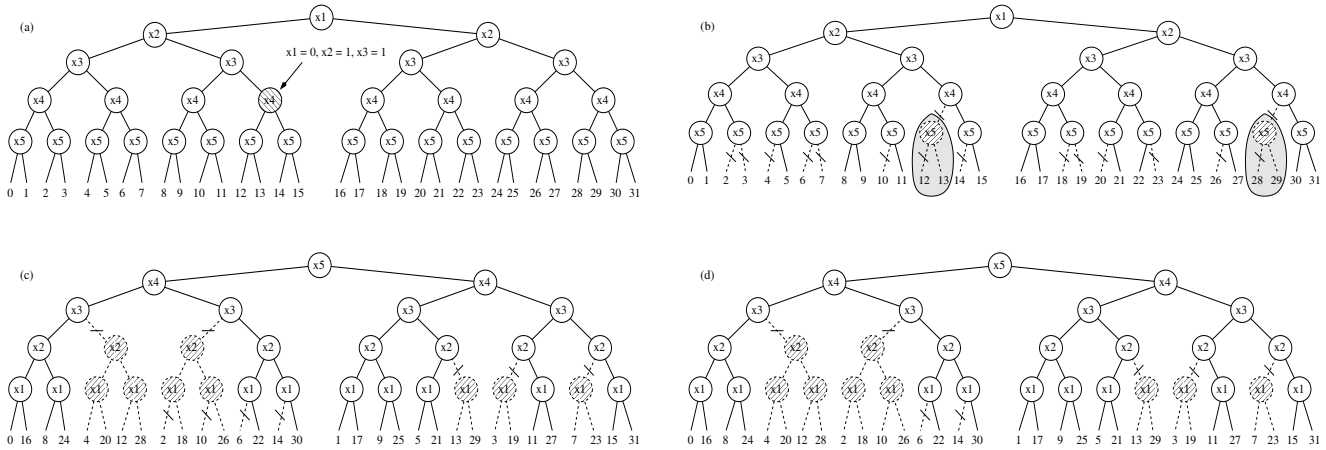

Fig. 2. Illustration of efficiency of SOFAS in finding all solutions to a small SAT instance. From left to right: (a) structure of the tree, (b) 16 original cuts, (c) 9 cuts after variable renaming, (d) 7 cuts after clause reordering.

Input: *a formula $\mathcal{F}$ over n Boolean variables $\{x_1, x_2, \cdots, x_n\}$.*
Output: *all the variable assignments that make $\mathcal{F}$ evaluate TRUE.*
Algorithm: **S**olve **O**nce and **F**ind **A**ll **S**olutions
/* Phase I: renaming the variables */
1. *compute the weighted occurrence of every variable x_i;*
2. *for $k = n, \cdots, 1$*
3. *rename the variable with the least weighted occurrence to y_k;*
4. *update the weighted occurrence of every remaining variable x_i;*
/* Phase II: reordering the clauses */
5. *sort the clauses by the largest indices of their variables in ascending order;*
/* Phase III: cut the non-solutions */
6. *mark all clauses unchecked;*
7. *let the solution space to be all the possible assignments;*
8. *while the solution space is non-empty and there are unchecked clauses;*
9. *cut all assignments that cannot satisfy the top ranked unchecked clause;*
/* Phase IV: report result */
10. *if the current solution space is empty*
11. *report $\mathcal{F}$ unsatisfiable;*
12. *else*
13. *report the solution space: all truth assignments to $\mathcal{F}$;*

Fig. 1. Pseudo code of SOFAS.

the two shaded subtrees in Figure 2(b), which will be pruned. A *cut* is the prune of a subtree and the clause $(x_2' + x_3' + x_4)$ results in two cuts. For the above formula $\mathcal{F}$, checking the five clauses in the order given in the definition of $\mathcal{F}$ results in 16 cuts as shown in Figure 2(b). Notice that there are 4 cuts associated with the last clause $(x_3 + x_4' + x_5)$, however, two of them have already been pruned by the first clause $(x_1 + x_4' + x_5)$. The 16 remaining leaves stand for all the truth assignment to $\mathcal{F}$.

Fact: Finding all truth assignments to a SAT formula is equivalent to eliminating all those that violate one or more clauses or to pruning the corresponding leaves from the binary decision tree.

SOFAS is based on this observation. It seeks to minimize the total number of cuts by renaming variables and reordering clause as one can see from the following lemmas.

Lemma 2.1 A clause with k variables in an n-variable formula makes 2^{n-k} truth assignments non-solutions. Furthermore, in the binary decision tree, these non-solutions correspond to $2^{i_k - k}$ subtrees each of size 2^{n-i_k}, where i_k is the largest index of the k variables.

In line 1, we define a clause with k literals to contribute 2^{-k} to the *weighted occurrence* of each of its variables. If a variable with a *weighted occurrence* ω is renamed to have the highest index n, then all the clauses that have this variable will result in $2^n \cdot \omega$ cuts based on Lemma 2.1. In light of this, we rename the variable with the smallest weighted occurrence to have the highest available index (line 3) and repeat this untill all the variables are renamed. This renaming procedure results in the following variable name convertion for our example formula $\mathcal{F}$: $\{y_1 = x_5, y_2 = x_4, y_3 = x_3, y_4 = x_2, y_5 = x_1\}$. We can then rewrite $\mathcal{F}$ as $(y_1 + y_2' + y_5')(y_1 + y_2 + y_3')(y_1' + y_2' + y_4)(y_2 + y_3' + y_4')(y_1 + y_2' + y_3)$. There will be only 9 cuts as depicted in Figure 2(c).

Some leaves of the binary tree may be pruned more than once when we consider the clauses one by one. For example, the leaves 2 and 10 in Figure 2(c) have been cut twice: first by the first clause $(y_1 + y_2 + y_5')$, then indirectly by the last clause $(y_1 + y_2' + y_3)$.

Lemma 2.2 For two clauses C and C', let x_k and $x_{k'} (k' \leq k)$ be the variable with the largest index and T and T' be any subtree pruned by C and C' respectively. If $k > k'$, then $T \cap T' = \phi$ or $T \subset T'$; if $k = k'$, then $T \cap T' = \phi$ or $T = T'$.

Lemma 2.2 suggests that if we check clause C' and make the corresponding cuts before we consider C, then no subtrees will be pruned twice. Phase II of the SOFAS algorithm (line 5 in Figure 1) enforces this by sorting the clauses by their largest indexed variables in ascending order. As a result, we need only 7 cuts for the rearranged formula $\mathcal{F} = (y_1 + y_2 + y_3')(y_1 + y_2' + y_3)(y_1' + y_2' + y_4)(y_2 + y_3' + y_4')(y_1 + y_2 + y_5')$ as shown in Figure 2(d).

After renaming variables and reordering clauses, SOFAS reads the clauses one at a time. For each clause, a set of non-overlapping intervals of the same length will be generated based on Lemma 2.3, to represent the non-solution subtrees. The union of all such intervals gives all the non-solutions and thus defines the solution space.

Lemma 2.3 For a k-literal clause with x_{i_k} as the highest indexed variable, the non-solution leaves can be represented by the union

of intervals $[S+A, S+A+l-1]$, where $l = 2^{n-i_k}$ is the length of the interval, A is a clause-dependent constant, and S takes 2^{i_k-k} different values depending on the k literals in the clause.

In SOFAS we developed an efficient algorithm that (i) identifies a group of intervals that have already covered and skips them and (ii) merges consecutive intervals to keep the number of intervals minimal at all times. Details of this algorithm are omitted due to space limitation.

3. SOLUTION SPACE ESTIMATION BY SAMPLING

As the size of the SAT instance increases, both the search space and the potential solution space grow exponentially. Consequently any attempt in finding all the solutions will require an exponential run time. In this section, we present two efficient sampling techniques for the estimation of the size of the solution space.

3.1. Sampling over Smaller SAT Instances

This technique takes samples of solution space, by reducing the original SAT instance, which have a much smaller search space. It is based on the assumption that the average solution space size over a large number of smaller SAT instances generated from the original formula reflects the size of the original solution space.

In step 1, we create an unbiased estimation by eliminating all variables that have the same values over the entire solution space. We then create a smaller SAT formula in steps 2 and 3. If a selected variable x is assigned value '1', for example, we delete all the clauses with literal x and remove x' from all the remaining clauses. Note that this gives us a formula with k fewer variables and a much smaller solution space ($1/2^k$ of the original one). We then determine the solution space in step 4 where an unsatisfiable instance is considered to have zero solution. The repetition of steps 3 and 4 in steps 5 and 6 will help us to get a better estimation in step 7. Note that we do not assume a random distribution of the solution space. Instead, we take a large number of samples to estimate the average size of each solution subspace.

1. apply the Davis Putnam procedure [2] to determine the values of those variables that must have a constant value in all solutions. Let C be the list of c such variables.
2. randomly select k variables other than the c variables in C.
3. assign random values to these c variables
4. update the SAT formula and determine the number of solutions by solving for all solutions.
5. repeat steps 3 and 4 t times with different random assignments to the same set of k variables. Let $\{n_1, n_2, ..., n_t\}$ be the number of solutions in these t trials and $T = n_1 + \cdots + n_t$ be the total number of solutions.
6. repeat steps 3-5 K times and obtain the total number of solutions for each trial $\{T_1, T_2, ..., T_K\}$.
7. estimate the number of solutions for the original SAT formula to be $\frac{T_1+T_2+\cdots+T_K}{K*t} \cdot 2^k$

Fig. 3. Sampling over SAT instances of smaller size.

3.2. Sampling by (Strategically) Different Solvers

This is a variation of the following classical sampling technique: take 10 balls randomly from a blackbox, mark them and put them back into the box. Then take again 10 balls randomly, if 5 of them have been marked, then we estimate that there are around 20 balls in the box because half of the redrawn samples repeat. This relies on the fact that the sample drawing is conducted randomly.

However, when we apply a solver to a SAT instance, we have no guarantee that the solver will give us a random satisfying solution. When we repetitively solve the same problem with the same solver, it is not clear whether we will get the same solution or a different solution; and if different, whether the two solutions correlate with each other. In fact, many solvers have the tendency to find the same solution when solved repetitively.

To overcome these problems, we start with two solvers, $\mathcal{S}_1$ and $\mathcal{S}_2$, preferably strategically different solvers. We apply each solver to find a certain number of distinct solutions. To ensure that the solvers find different solutions each time, we append a new clause to the formula once a new solution is found. For example, if we have a solution $x_1 = 0, x_2 = 1$, and $x_3 = 0$ to a formula $\mathcal{F}$, we then add the clause $x_1 + x_2' + x_3$ to $\mathcal{F}$. Solving this new augmented instance guarantees us a new solution. Suppose that we have obtained k_1 and k_2 solutions by $\mathcal{S}_1$ and $\mathcal{S}_2$ respectively, where k solutions are reported by both solvers. We are able to estimate that the original instance has $\frac{k_1 \cdot k_2}{k}$ solutions.

This sampling technique solves the original SAT instance. However, it only looks for a certain number of solutions rather than finding all of the solutions. We argue that the run time to find a limited number of sample solutions will be much less than the time it takes to enumerate all the solutions. Our experiments also validate this argument. Finally, we mention that the two proposed methods – sampling over smaller SAT instances and sampling over different solvers – are orthogonal, and they can be combined for better run time efficiency.

4. EXPERIMENTAL RESULTS

We implement SOFAS and a naïve brute force search algorithm using the same data structure (for a fair comparison of their performance) and compared them on a set of uniform randomly generated 3-SAT formulas with 15 to 30 variables and a constant 4.3 clause-variable ratio. In the naïve brute force search algorithm, we exhaustively check for all the possible truth assignment one by one. An assignment becomes a solution if and only if all the clauses are satisfied. Otherwise, we move on to check the next truth assignment. SOFAS is 5X-100X times faster than this brute force search. We then compare the speed for various solvers, including SOFAS, Posit, Sato, and Satz, to find all the solutions for 3-SAT benchmarks. The 3-SAT instances include random formulas with 15, 20, and 25 variables as well as standard DIMACS and satlib benchmark instances [8, 3]. For solvers we find all the solutions by solving for one solution and then enforcing the solver to find a new solution until it fails to find one. Due to space constraints, we only mention that for these small- and medium-sized 3-SAT benchmarks, SOFAS is compatible with Posit, Sato, and Satz, particularly when there are many solutions. Furthermore, SOFAS always give the correct number of solutions, while other solvers cannot guarantee to find all the solutions occasionally.

We now evaluate the accuracy of the two solution space estimation methods. The SAT problems are the unforced uniform random 3SAT benchmarks from [3]. For space consideration, here we only report our results on two sets of benchmarks: 500 instances with 50 variables and 218 clauses, and 100 instances with 75 variables and 325 clauses. They are all satisfiable instances with the number of solutions ranging from one to a few thousand, which

we obtain from repetitively running Zchaff.

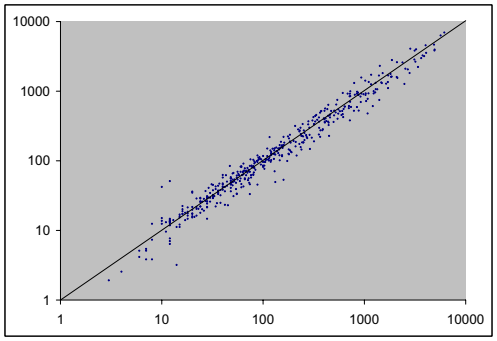

Fig. 4. Accuracy of sampling method I on 500 50-variable 3SAT instances. X axis: actual number of solution; Y axis: estimated number of solutions.

Figures 4 and 5 demonstrate the accuracy of the first sampling technique by plotting the actual and estimated number of solutions. The values of k,t and K are set to be 5, 10 and 10 respectively. That is, we randomly choose 10 sets of 5 variables and assign 10 different assignments for each set. We solve all the corresponding smaller sized SAT problems for all solutions by Zchaff and then estimate the number of solutions for the original problem by the formula given in Figure 3. The 45-degree line indicates the situation when the estimation meets exactly the actual number. Points above and below this line are the overestimated and underestimated cases respectively. Both figures show that our estimation is fairly close to the actual number of solutions. In fact, for the 500 50-variable benchmarks, the average error, measured by $\frac{1}{500}\sum_i \frac{estimation-actual\ number}{actual\ number}$, is only **0.2%** with most of the error comes from instances that have less than 20 solutions.

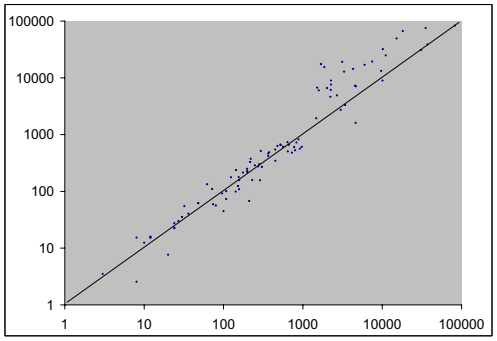

Fig. 5. Accuracy of sampling method I on 100 75-variable 3SAT instances. X axis: actual number of solution; Y axis: estimated number of solutions.

For the second method, we use Zchaff and Satz as the two strategically different solvers to obtain 100 and 250 (when applicable) solutions independently for each instance. We then compare these reported solutions and use the number of solutions found by both solvers to estimate the solution space of the original problem as we have discussed earlier. Figure 6 reports the result on 200 50-variable instances with at least 100 solutions. Although this method is faster than the previous sampling technique, one can see that it tends to underestimate the size of the solution space, particularly for those with large number of solutions. The reason is that the solutions, found by both solvers and those in the entire solution space, normally form groups rather than being randomly distributed. Therefore, instead of finding individual solutions that are in common, the two solvers usually report groups that have many solutions in common. This misleads us to underestimation. We expect to improve the accuracy by investigating on how to force solvers to find solutions that are far away to each other.

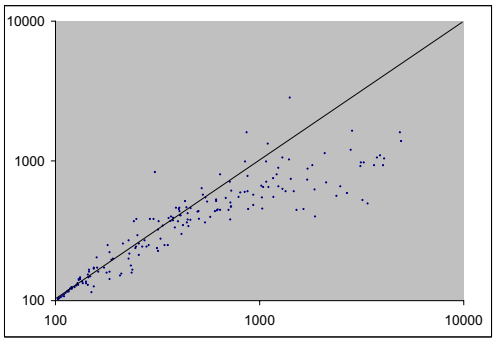

Fig. 6. Accuracy of sampling method II on 200 50-variable 3SAT instances. X axis: actual number of solution; Y axis: estimated number of solutions.

5. CONCLUSION

We present two different approaches to determine the solution space of a given SAT instance. The first one is an exact and effective algorithm based on pruning non-solutions targeting small to medium sized SAT problems. The second approach consists of two sampling techniques to estimate the size of the (large) solution space. We test these two estimation techniques on the SATLIB benchmark instances. The results show that they are fairly accurate. With these techniques, one can better understand the solution distribution and eventually the nature of the SAT problem. It also becomes possible to find better solutions for applications that require the knowledge of the entire solution space.

6. REFERENCES

[1] F.A. Aloul, B.D. Sierawski, and K.A. Sakallah. "Satometer: How Much Have We Searched?" *39th ACM/IEEE Design Automation Conference*, pp. 737-742, June 2002.

[2] M. Davis and H. Putnam. "A Computing Procedure for Quantification Theory", *Journal of the Association for Computing Machinery*, Vol. 7, No. 3, pp. 201-215, July 1960.

[3] H.H. Hoos and T. Stuzle. SATLIB: An Online Resource for Research on SAT. In *SAT 2000 (ed. I.P. Gent, H.V. Maaren, and T. Walsh*, pp. 283-292, IOS Press, 2000.

[4] J.P. Marques-Silva and K.A. Sakallah. "Boolean Satisfiability in Electronic Design Automation," *37th ACM/IEEE Design Automation Conference*, pp. 675-680, June 2000.

[5] B. Selman, H.A. Kautz, and B. Cohen. "Noise strategies for improving local search", *Proceedings of the 12th National Conference on Artificial Intelligence, AAAI'94*, pp. 337-343, 1994.

[6] J. P. M. Silva and K. A. Sakallah. "GRASP—A New Search Algorithm for Satisfiability", *Proceedings of the International Conference on Computer-Aided Design*, 1996.

[7] L. Zhang, C.F. Madigan, M.H. Moskewicz, and S. Malik. "Efficient Conflict Driven Learning in a Boolean Satisfiability Solver", *IEEE/ACM International Conference on Computer Aided Design*, pp. 279-285, November 2001.

[8] http://dimacs.rutgers.edu/