ABSTRACT

Title of Thesis:	Several Issues on the Boolean Satisfiability (SAT) Problem
Degree candidate:	Pushkin Raj Pari
Degree and year:	Master of Science, 2004
Thesis directed by:	Professor Dr. Gang Qu Department of Electrical Engineering

Boolean Satisfiability (SAT) is often used as the model for a significant and increasing number of applications in Electronics Design Automation (EDA) and many other fields of computer science and engineering. Although the SAT problem belongs to the class NP-complete - problems that do not have a polynomial run time algorithm but answers for which can be checked for correctness, by an algorithm with run time is polynomial in the size of the input - typical SAT instances are easy to solve. Both theoretical and empirical studies have been conducted on various SAT models to investigate when SAT instances become hard to solve. For more than a decade, crossover point has been the only parameter considered for hardness. Existing results state that for random SAT, the problems becomes relatively harder when the clause to variable ratio is 4.3.

This thesis work is motivated by the observation that not all benchmarks at the crossover point are hard. We conjecture that the structure of the solution space is also related to the hardness. We provide an empirical framework for the validation of this conjecture. Firstly, we show by experiments that (1) the crossover point is not the only metric to characterize hardness and (2) existing benchmarks are inefficient in providing solution space information We present a novel approach to generate the SAT instances with known solution space structure. Another related issue on how to obtain the solution space information is also discussed, where we propose two probabilistic techniques for quick estimation of the solution space with high accuracy.

Several Issues on the Boolean Satisfiability (SAT) Problem

by

Pushkin Raj Pari

Thesis submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Master of Science
2004

Advisory Committee:

Professor Dr. Gang Qu, Chair
Professor Dr. Manoj Franklin
Professor Dr. Bruce Jacob

DEDICATION

To amma, naina and papa

ACKNOWLEDGMENTS

Firstly, I would like to thank my advisor Dr. Gang Qu for his invaluable guidance in conducting my research. I am fortunate to have an advisor with such levels of patience, enthusiasm, and dedication.

I would like to thank Dr. Manoj Franklin and Dr. Bruce Jacob for their time and effort in serving this committee.

I would also like to thank Yuan Lin, Jane Lin, Matthew Schmidt, and Chih-Yuan Huang for all their help and time spent in useful discussions. Special thanks are due to Jane Lin and Matthew Schmidt for their work over the summer.

I thank my friends and labmates for making me feel at home. In particular Bala, Sada, Arun, Ming-Yung, Uday, Cedric, Suresh, Gautham, IyerB, Aditya, Mainak, Ankush, Deepak, Shaoxiong and Guang have made life in graduate school more bearable.

I owe everything in my life to my parents and my loving sister, who made growing up fun.

To folks whose names I have missed, I offer my apologies in addition to my thanks.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction

1.1 Introduction to SAT

Boolean Satisfiability (SAT) is often used as the underlying model for a significant and increasing number of applications in Electronics Design Automation (EDA) and many other fields of Computer Science and Engineering. Further a large number of problems that occur in knowledge representation, learning, planning and other areas of artificial intelligence (AI) are essentially satisfiability problems. The first, and one of the simplest, of many problem which have been proved to be NP complete, SAT holds a central position in the study of computational complexity. Many practical problems are NP hard and may be transformed efficiently to SAT, or have components problems which can be. Although SAT is NP complete in the worst case, many instances are easily solved in practice. Finding ways to generate hard instances is important for understanding the complexity of the problem, and for providing challenging benchmarks to evaluate new algorithms.

Many constraint satisfaction problems can be translated to SAT problem. Consider a real life problem where few friends decide to go for a trip during a long weekend. They have to choose one of the days from Thursday to Monday. Charles wants to watch the Sunday football, so he prefers Friday or Saturday. Diane cannot make it on Monday. Ron and Daisy have already made plans for Friday, but are free on other days. With few constraints it is easy to find a day on which all the friends can make it. As the number of friends, who plan to go for the trip increases the constraints on the choice of day also increases. If there is conflicting requests it becomes impossible to choose a day. If Ron can make it only on Sunday, then there is no day when all the friends are available. The problem becomes easier as the constraints increases, which might make the problem insoluble. Researchers have found that the hardest problems lay in the middle when the number of constraints is just enough to limit the choices available but not enough to eliminate all the possible choices. These kinds of logic puzzles in the theoretical computer science are called as satisfiability (SAT) problem.

These problems can be translated into symbolic logic, where each variable is true or false. Consider the example, three variables, x, y and z, and the logical statement (x OR y OR z) AND ((NOT x) OR (NOT Z)). The OR means that the clause (x OR y) is true only if either x or y is true, and the AND means that the clause (x AND y) is true only if both x and y are true. Solving this problem requires assigning true or false to the variables such that the expression is true.

Here one of the solutions, which make the expression true is when, x is true, y is false and z is false.

1.2 Definition

Consider the following 5-variable SAT formula

$\mathcal{F} = (x_1 + x_4' + x_5)(x_3' + x_4 + x_5)(x_2 + x_4' + x_5')(x_2' + x_3' + x_4)(x_3 + x_4' + x_5)$. Here x_1, x_2, x_3, x_4 and x_5 are called as Boolean variables which takes values 1 or 0 (equivalent to true or false). Each variable can appear in positive and negative forms. Each sum of variables, for example $(x_1 + x_4' + x_5)$ is called as a clause. The clause is true if x_1 is true or x_4 is false or x_5 is true. The symbol '+' is equivalent to OR and the product is equivalent to AND. The product of the clauses like give above, gives one formula or instance. A formula is called satisfiable if and only if all these clauses become true. The goal of the Boolean SAT problem is to find one truth assignment to satisfy the formula, or prove that the formula is unsatisfiable. Here the assignment $x_1 = 1$, $x_2 = 1$, $x_3 = 0$, $x_4 = 1$ and $x_5 = 1$ makes the formula to be true.

Formally, the propositional Satisfiability (SAT) problem is defined as follows [1]:

Instance: A set of clauses C on a finite set U of variables.

Question: Is there a truth assignment for U that satisfies all the clauses in $\mathcal{C}$?

A truth assignment is a mapping from U to {*true, false*} or { 0 , 1 }.

K-SAT is a special case of SAT problem, where each clauses has exactly has k literals. A variable either in positive or negative form is called as a literal. When $K \geq 3$ the SAT problems are classified as NP complete. 3SAT, when $K = 3$, is considered as a special K-SAT problem because all higher KSAT, $K > 3$ can be reduced to 3SAT problems. In general 3SAT problems are used for the purpose of studying the properties of satisfiability problems.

Random 3SAT is a family of SAT problems obtained by randomly generating 3-literal clauses for a formula in the following way: For an instance with n variables and m clauses, each of the k clauses is constructed from 3 literals which are randomly drawn from the 2n possible literals (the n variables and their negations) such that each possible literal is selected with the same probability of 1/2n. Clauses are not accepted for the construction of the problem instance if they contain multiple copies of the same literal or if they are tautological (i.e., they contain a variable and its negation as a literal). Each pair of n and m thus induces a distribution of random-3-SAT instances.

1.3 Finding Hard SAT Problems

In early 90's of the 20th century researchers tried both theoretically and empirically to determine parameters to generate hard SAT benchmarks. Cheeseman et al. has shown that NP complete problems can be summarized by an "order parameter" and that the hard problems occur at the critical value of

such a parameter [2]. They showed that for K-Satisfyability problem this critical value separates the over-constrained from the under-constrained random K-SAT instances. Also, the normalized cost of finding a solution had a phase transition at about the point at which the solution probability is near zero.

Mitchell et al. performed experiments on random SAT formulas. They showed that by using right distribution of clauses and appropriate number of variables it is possible to generate hard instances [3]. He showed that when the clause to variable ratio is around 4.3, the generated the instances had a 0.5 probability of being satisfiable and were computationly challenging instances.

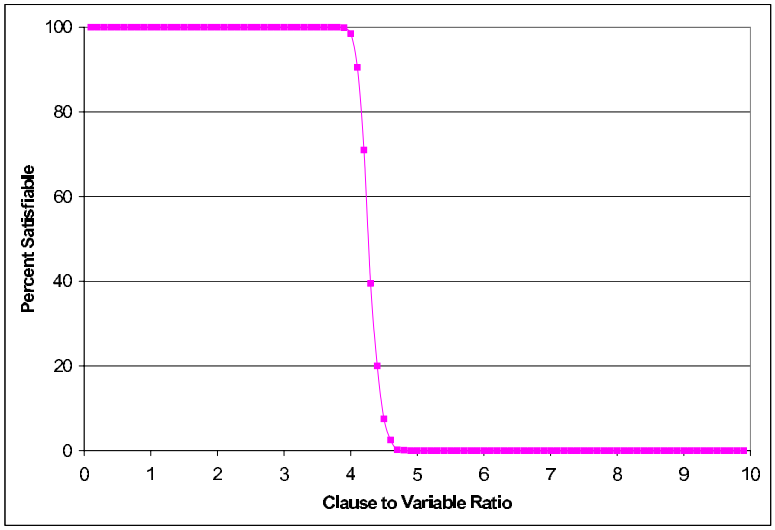

Figure 1.1: Phase transition or crossover point

Crawford et al. has shown experimentally that for random 3SAT problems the number of constraints required for crossover is a linear function of the number of variables [1]. They confirmed the results of Cheeseman et al. and Mitchell et al. They did extensive experiments to determine this crossover point. Figure 1.1

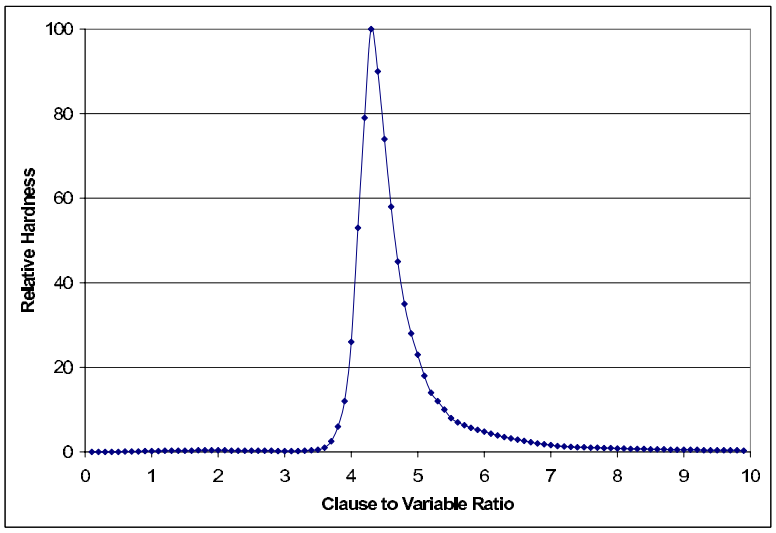

Figure 1.2: Hardness at the crossover point

shows critical region for randomly generated 3SAT, 50 variable instances. We can see that there is sudden phase transition from satisfiable instances to unsatisfiable instances. This transition happens when the clause to variable ratio is between 4 and 5. Figure 1.2 shows that the instances at this region are relatively harder than the other ones. They measured hardness as the number of nodes in the search tree and the graph is drawn for the normalized nodes. It shows that when clause to variable ration is around 4.3 the problems are relatively harder than the other. Also at the same point there is the 0.5 probability for an instance to be satisfiable. They called this point as the "crossover point".

The crossover point is of practical interest, since it has been proved that most of the hard problems seem to be found here. For more than a decade the only parameter used to generate hard random 3SAT problems is the phase transition

for unsatisfiable to satisfiable, when there are about 4.25 clauses for every variable.

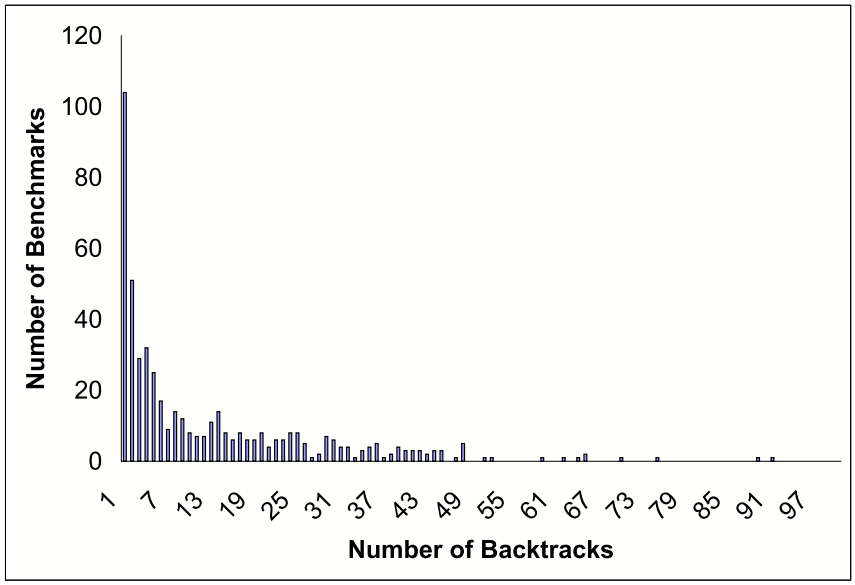

Figure 1.3: Evaluation of hardness at the crossover point

Although the crossover point contains the hard problems, there also exists problems that are easy to solve in this critical region. To check the hardness of the random 3SAT at the critical region, we generated 1000, random 50-variable 3SAT instances with 218 clauses. We used zchaff, a state of the art SAT solver, to solve the formulae. Since zchaff is extremely fast, we use the number of backtracks instead of CPU time as a metric to measure hardness of a formula. Figure 1.3 shows the distribution of the number of backtracks in finding the first truth assignment or determining the formula is unsatisfiable for these 1000 benchmarks. We can see that most of the benchmarks required very few number

of backtracks. This clearly shows that the critical region is not the only metric to characterize hard random 3SAT benchmarks.

This gives room to think of other possible parameters for hardness. SAT as a constraint satisfying problem, the number of clauses was considered for hardness. Intuitively the crossover point could be the major intersection of the real hard problems. Since the solvers are trying to search a solution from the solution space, it is possible that the number of solutions could also contribute to the hardness of SAT problems. We conjecture that the structure of solution space is also related to the hardness. The present benchmarks gives the information whether they are satisfiable or not and nothing about the number of solutions present in them. So we need either to find all the solutions to these benchmarks or an algorithm which generates benchmarks with known solutions.

1.4 Estimation of Solution Space

There are two methods to find all the solutions, either by brute force search or by using a solver to repeatedly solve for new solutions. Conducting a brute force search, which evaluates all the possible truth assignments for the variables over the entire solution space, becomes impractical as the solution space grows exponentially as the number of variables increases. Although the use of solver can be make the procedure faster, still the time to solve for all solutions increases exponentially with increase in number of solutions in a problem. Some solvers

fails to find a solution if the search time exceeds certain bound, in such cases the calculated number of solutions are not accurate.

We propose a couple of sampling techniques to estimate the size of the solution space. The first method randomly assigns values to a set of variable 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.

From the results of the sampling techniques we observed that the second method tends to underestimate the solution space. This gives us the insight that the solution space of a random 3SAT problem is not random, rather clustered or in groups of solution. We define a set of solutions to be in a group or cluster if they form a Prime Implicant.

1.5 Benchmarks with Known Solution Space

We generated 100,000 random 50-variable 3SAT benchmarks at the critical region and solved for all the solutions. We counted the number of instances which have the same number of solutions. We observed that for fewer number of solutions there were many instances and comparatively lesser number of instances for higher number of solutions. For certain number of solutions there were no sample instances. It is very difficult to draw any conclusion without

more or less equal number of benchmarks or instances for various number of solutions. Also it consumes a lot of CPU time to find all the solutions. We need algorithms to generate benchmarks faster, which are random and with a given number of solutions.

Hence we propose a novel benchmark generator engine which can generate benchmark with known number of solutions and certain structure of solution space. Using this engine we can generate random 3SAT benchmarks faster with a known solution space. We generate the random benchmark from a random core. This core consists of fewer number of variables. we add new variables to the existing core and keep the distribution of variables to be random. We also keep control of the number of solutions and keep track of the existing solution space. The algorithm can generate benchmarks for a given solution number within a reasonable tolerance range. These benchmarks have practical values, as they can be used to define a new phase transition or crossover point for the number of solutions or groups or even both, if it exists.

The rest of the thesis is organized as follows. In the next two sections we describe the techniques for solution space estimation and the algorithm for generating benchmark. We conclude in the last section with future work.

Chapter 2

Estimation of Solution Space

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 chapter, we present two efficient sampling techniques for the estimation of the size of the solution space.

2.1 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 of solution space size over a large number of smaller SAT instances generated from the original formula reflects the size of the original solution space. Figure 2.1 gives the step by step procedure for this technique. In step 1, we create a unbiased estimation by eliminating all variables that have the same values over the entire solution space. We use the Davis Putnam algorithm for this purpose. If a variable is forced a truth assignment and if this makes the

formula unsatisfiable, then the variable is assigned the opposite value. Based on this assignment, the clause set is simplified by deleting each clause that is satisfied by this assignment, and in every other clause delete the corresponding literal that contradict the truth assignment. By the end of this procedure, we delete all the literal which have the same value over the entire solution space, also called as pure-literal. 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[4] 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 and update the SAT formula.
4. determine the number of solutions of the new formula obtained above 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 $\frac{T_1+T_2+\cdots+T_K}{K*t} \cdot 2^k$

Figure 2.1: Sampling over SAT instances of smaller size.

2.2 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 solutions with different strategies.

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.

2.3 Simulation Results

We now evaluate the accuracy of the two solution space estimation methods. The SAT problems are the unforced uniform random 3SAT benchmarks from [5]. 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.

Figures 2.2 and 2.3 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

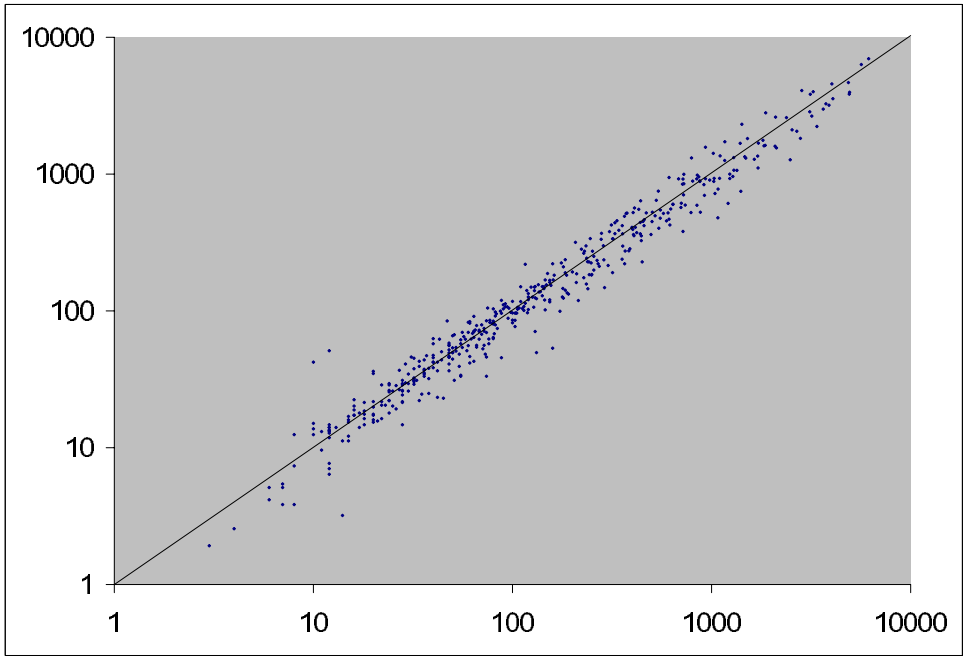

Figure 2.2: Accuracy of sampling method I on 500 50-variable 3SAT instances. X axis: actual number of solution; Y axis: estimated number of solutions.

estimate the number of solutions for the original problem by the formula given in Figure 2.1. 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. fairly close to the actual solutions. The average percentage error 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.

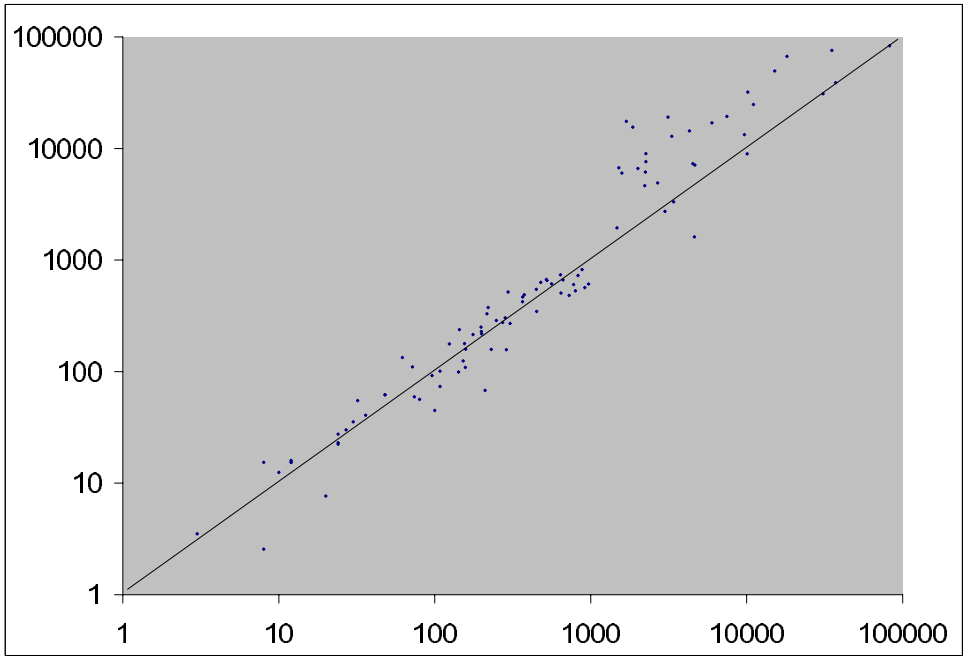

Figure 2.3: Accuracy of sampling method I on 100 75-variable 3SAT instances. X axis: actual number of solution; Y axis: estimated number of solutions.

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 2.4 reports the result on 200 50-variable instances with at least 100 solutions. One can see that the second method 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.

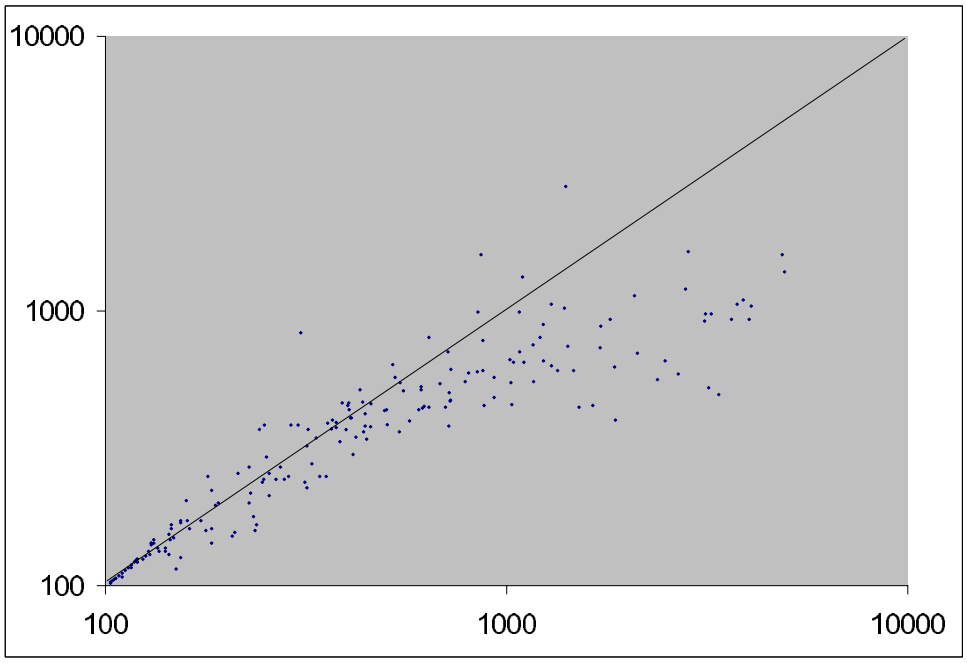

Figure 2.4: Accuracy of sampling method II on 200 50-variable 3SAT instances. X axis: actual number of solution; Y axis: estimated number of solutions.

2.4 Summary and Discussions

We present two different approaches to estimate the solution space of a SAT instance. The first method is more accurate than the second method due to the presence of group or cluster of solutions. Since these sampling methods are orthogonal they can be combined together to get more accurate results with less run time. Although it helps to obtain the number of solutions, the entire solution assignments are still unknown. But it surely does give an idea of the solution size of the SAT problem, without solving for all of the solutions.

Chapter 3

Random 3SAT Benchmark Generator

3.1 SAT Benchmarks

There is always a need for new benchmark generator algorithms to test the efficiency of the SAT solvers and also to understand the SAT problem. A number of test Instance generation algorithms proposed in literature [6, 7, 8, 9], were mainly to generate hard instances based on the number of constraints. Motoki et al. proposed an algorithm to generate 3SAT instances with unique solutions with high probability [10]. It was the only algorithm known, to take solution space into account to generate 3SAT instances. We go a little further and propose an algorithm which can generate instances for a given number of solutions. SAT is a constraint satisfiability problem, hence the previous generator algorithms concentrated on the constraints or clauses to make the instance harder. Also SAT algorithms are similar to search algorithms trying to find a solution assignment from a pool of solutions, if exists. This led us to this novel benchmark generator algorithm which takes into account the solution space and

generates instances with controlled number of solutions.

3.2 Benchmark Generator Algorithm

The benchmark generator algorithm uses some of the properties for clauses in the literature, which makes a SAT instance hard. It generates benchmarks at the crossover point, as it has been shown that the instances at this region are relatively harder. So every instance generated will have 4.3n clauses per variable. Since any SAT instance can be reduced to 3SAT and also it is the most popular form of SAT studied widely. We concentrate only on generating 3SAT instances. Research has been done to show that random 3SAT problems are harder than another. We generate instances which resemble the randomly generated 3SAT instances in the distribution of variables. We quantify the randomness of the generated instances in the experimental results section of this chapter. The most important feature of this algorithm is it lists all the satisfying assignments for the generated instance. The algorithm takes two inputs; one is the number of solutions required and the other being the tolerance range acceptable.

The above mentioned properties of the benchmark generator can be listed as

1. Resemblance to randomly generated 3SAT benchmarks
2. 4.3 clauses per variable
3. Number of Solutions in the final formula
4. List of all the satisfying Assignments

The key challenge is to maintain the same number of clauses (4.3 n) for a given number of solution space and also to keep the variable distribution random. In other words, the variables used cannot violate certain constraints or number of occurrence. We use number of occurrence, pair wise occurrence, triplets as the metrics to measure the randomness of the instances. This would lead to rejecting some of the clauses, purely because it violates the randomness criteria. Another key challenge, when a clause is added, its effect on solution space is also watched. Clauses will be rejected by the benchmark generator algorithm if they either kill all of the solutions or over kill the solutions. Sometime it is not possible to get the exactly the same number of solutions as required. Hence we have a tolerance range for the number of solutions required. The generated benchmarks will have a solution space within the requested tolerance.

The randomness is a very important constraint in this benchmark generation algorithm. In order to maintain this, we generate the instances from a random core. This is the first phase of the algorithm, where a random 3SAT problem is generated and then reduced to lower number of variables just be deleting the other variables. The choice of the core variables is random. This phase is to give a starting point for the next phase where these initial variables are grown to the required number of variables by adding new literals. The second phase is the main phase and has many constraints like maintaining the randomness of the formula, 4.3 clauses to variable ratio and most importantly the number of

solutions in the final formula.

Thus the two phases of the algorithm are

1. Phase I - Generation of Random Core
2. Phase II - Growing the core to desired number of variable with desired number of solutions

3.2.1 Generation of Random Core - Phase I

In order to generate a random 3SAT instance, the generator requires a random core. The function of this program is to remove the variables from the randomly generated 3SAT Instance to generate a random core. The figure 3.1 shows the pseudocode of this phase. for example lets assume $k = 3$. We generated a random 3SAT core with $n = 6$ variables and choose 3 variables by random choice. In step 3 the program scans the entire instance and removes all the other variables. The resulting formula will have 3-literal, 2-literal and 1-literal clauses. Now rename the variables to 1 to 3. This will be the random core which will be grown by the benchmark generator. The solution space for the 3 literal clauses is considered as the starting solution space for the instance. The random core obtained at this stage is a general SAT formula with 3, 2 and 1 literal clauses.

Table 3.1 shows the result of phase-I on an example problem. Original random 3SAT instance has 6 variables and 26 clauses. The value of $k = 3$ and The random choice of variables are 1, 3 and 5.

Step 1: Generate a random 3SAT benchmark with n variables and 4.3n clauses

Step 2: Choose by a random choice - k variables

Step 3: Filter out rest of the variables

Step 4: Rename the existing variables to 1 to k

Step 5: Sort the clauses into categories of 3-literal, 2-literal and 1-literal clauses

Step 6: Find all the solutions for the 3-literal clauses alone

Figure 3.1: Random core generation - Phase I

$k = 3$ and selected k variables = {1, 3,and 5}

3SAT Instance		Filtered Instance		Renamed Clauses
-1 6 2		-1		-1
-6 -3 -5		-3 -5		-2 -3
-1 6 5		-1 5		-1
-6 5 -3		5 -3		-2 3
1 -5 4		1 -5		1 -3
4 -1 3		-1 3		-1 2
1 -5 -2		1 -5		1 -3
5 1 2		5 1		1 3
-1 3 -2		-1 3		-1 2
6 2 -1		-1		-1

Table continued in next page.

3SAT Instance		Filtered Instance		Variable Renaming
-4 -3 1		-3 -1		-1 -2
-5 6 3		-5 3		2 -3
-1 -5 -3		-1 -5 -3		-1 -2 -3
6 -4 1		1		1
5 -1 4		5 -1		3 -1
1 5 2		1 5		1 2
3 5 1		3 5 1		1 2 3
-5 2 4		-5		-3
3 -6 2		3		2
3 1 -6		3 1		1 2
4 -5 -2		-5		-3
-4 2 3		3		2
-4 2 -5		-5		-3
2 -1 -4		-1		-1
6 4 2		-		-
6 -2 1		1		1

Table 3.1: Generation of random core of 3 variables from a 3SAT instance with 6 variable and 26 clauses

After the renaming the variables, the clauses of the random core is sorted in to three categories, namely 1-literal, 2-literal and 3-literal clauses. Table 3.2

	1-Literal	2-Literal	3-literal
1.	-1	-1 2	1 2 3
2.	-1	-1 2	-1 -2 -3
3.	-1	-1 -2	
4.	-1	-1 3	
5.	1	1 2	
6.	1	1 2	
7.	2	1 -3	
8.	2	1 -3	
9.	-3	1 3	
10.	-3	-2 -3	
11.	-3	-2 3	
12.		2 -3	

Table 3.2: Sorting the random core

shows count of those clauses. The 3-Literal clauses corresponds to initial 3SAT clauses, and the corresponding solutions is given in Table 3.3. The information from these two tables is given to the phase-II of the algorithm.

1.	0 1 0
2.	0 1 1
3.	1 0 0
4.	1 0 1
5.	1 1 0
6.	0 0 1

Table 3.3: Initial solution space

3.2.2 Growing the Core - Phase II

In this section we explain the main algorithm of generating random 3SAT benchmarks with controlled solution space. The Figure 3.2 gives the pseudocode for this procedure. The algorithms generates a table which acts as a constraint to the addition of literals. This constraint is imposed to make sure that the generated formula resembles random 3SAT. A clause is generated from three sources, adding literals to 1-literal, 2-literal clauses and also forming clause from new literals.

The pseudocode mainly comprise of the following parts:

1. Generation of literal occurrence table
2. Addition of literals to 2-literal clauses to make them 3SAT
3. Addition of literals to 1-literal clauses to make them 2SAT
4. Forming 3-literal clause using new literals.

The input for this program would be the random core generated by phase-I.

Literals are carefully added to the random core to make them all 3SAT and maintain the randomized distribution of variables. Let n be the number of variables for the 3SAT and if we choose to keep k literals and filter out the rest. The following are the percentage clause distribution of the k literals chosen

Percentage of 3SAT clauses with any of the k variables $= 1 - \frac{C_3^{n-k}}{C_3^n}$

Percentage of 3SAT clauses with only k variables $= \frac{C_3^k}{C_3^n}$

Percentage of 3SAT clauses with at most one k variable $= \frac{k \cdot C_2^{n-k}}{C_3^n}$

Percentage of 3SAT clauses with at most one $(n-k)$ variable $= \frac{(n-k) \cdot C_2^k}{C_3^n}$

Percentage of 3SAT clauses with only $(n-k) = \frac{C_3^{n-k}}{C_3^n}$

The occurrence table is built such that the generated benchmarks maintain the above clause distribution. This table consist of three rows as shown in the figure 3.4.

	V_{k+1}	V'_{k+1}	$\cdots$	V_n	V'_n
T_2			$\cdots$		
T_1			$\cdots$		
T_3			$\cdots$		

Table 3.4: Occurrence table

Each row gives the count of literals to be added to 2-sat, 1-sat and new literal clauses namely T_2,T_1,and T_3 respectively . Let x, y, and z be the number of 2-literal, 1-literal and 3-literal clauses of the reduced core. The total number literal to be added to 1-literal will be y. The number of literals to be added to

2-literal clauses are x+y. The number of new 3literal clauses to be added will be 4.3n-(x+y+z). hence the number of new literals would be 3*(4.3n-(x+y)) . The rows are incremented by random choice such that they total up to the corresponding value. This table will act as a guideline for the addition of literals. Consider the previous example Table 3.2, where $x = 12$, $y = 11$ and $z = 2$ are the number of 2-literal, 1-literal and 3-literal clauses. As shown above the sum of each rows T_2, T_1 and T_3 will be 23, 11 and 3. Table 3.5 shows the distribution of literals for this problem.

	V_4	V_4'	V_5	V_5'	V_6	V_6'	sum
T_2	2	2	4	2	7	6	= 23
T_1	3	1	6	1	0	0	= 11
T_3	1	0	0	1	1	0	= 3

Table 3.5: Occurrence table for example problem

In the figure 3.2, Steps 3 to 8 give the procedure to add literals to 2literal clauses and 1literal clauses. The clauses are chosen randomly, to maintain the random distribution. The existing solution space gets doubled every time a new variable is added. Every time a new 3SAT clause is generated, the solution space will be updated. If the solution space is still large, it is controlled by the addition of new 3 SAT clauses from the rest of variables. The 3SAT clauses are added until the solution space is close to the desired solutions. The occurrence of literals in the new 3SAT clauses are controlled by the 3^{rd} row of the occurrence

table. The clauses are generated by randomly choosing the literals and the corresponding value in the table in the decremented. Also the 3literal clauses generated are test for the following conditions before adding to the 3SAT list.

1.No duplicate literals

2.No clauses with both positive and negative literal

3.Number of solutions killed is not more than S, where S is the number of solutions more than the desired number of solutions $\pm$ Tolerance

Table 3.6 shows the instance generated by the benchmark generator algorithm using the occurrence table in Table 3.5 and the clauses from Table 3.2. The target solution is 4 with tolerance 3. For this problem the required number of solutions matches the desired number of solutions. Similarly instances can be generated for various values of variables and number of solutions. In the benchmark results section we evaluate the randomness of the generated benchmarks.

S.No	clause
1.	1 2 3
2.	-1 -2 -3
3.	-2 -3 4
4.	1 2 4
5.	2 -3 -4
6.	-1 2 -4
7.	1 2 5
8.	-1 -2 5
9.	-2 3 5
10.	1 3 5
11.	1 4 -5
12.	-3 4 -5
13.	-1 4 6

S.No	clause
14.	-3 5 6
15.	1 -3 6
16.	-1 2 6
17.	1 -3 6
18.	2 -5 6
19.	-3 5 6
20.	-1 5 -6
21.	2 -4 -6
22.	-1 5 -6
23.	1 5 -6
24.	3 -1 -6
25.	-1 5 -6
26.	4 -5 6

S.No	Solutions
1.	1 1 0 1 1 0
2.	0 1 1 1 1 1
3.	0 1 0 1 1 1
4.	0 1 0 1 1 0

Table 3.6: 3SAT instance generated by generator

Input: Desired number of variables - n, desired number of solution, 3-literal,

2-literal and 1-literal clauses, current solutions

Output: 3SAT CNF file, Solutions

Step1: Create a literal occurrence table - T

Step2: choose next available new variable - V_{k+j}

Step3: Let $T_2(V_{k+j})$ be the occurrence for the Positive literal

for the 2SAT clauses

Step4: Choose randomly $T_2(V_{k+j})$ clauses from 2SAT list

and append the literal

add to the 3SAT list

Step5: update the solution space and repeat step 3 and 4 for negative literal V'_{k+j}

Step6: let $T_1(V_{k+j})$ be the occurrence for the positive literal V in 1 SAT clauses

Step7: Choose randomly $T_1(V_{k+j})$ 1SAT clauses and append the literal

add to the 2SAT list

Step8: Repeat Step6 and Step7 for the negative literal V'_{k+j}

Step9: If number of solution $\geq$ desired Solution $\pm$ tolerance

add new 3literal clauses based on T_3

Step9: repeat steps 2-8 until all the variables are added

Figure 3.2: Growing the random core - Phase II

3.3 Random Instances with Random Solutions

The uniform random 3-SAT model does not guarantee that the solutions are randomly distributed. One evidence is the so-called *backbone*, variables that remain constant in all the solutions (Singer, Gent, and Smaill 2000).

To get a quantitative measurement of the solution randomness, we solve the SAT formula for all the solutions and then compute the Hamming distance between each pair of solutions. Figures 3.3,3.4,3.5 plot the average number of pairs with the same Hamming distance for 100 sets of solutions, with each set contains 10-64 solutions, over three different families. In Figure 3.3, the solutions are selected randomly and we see the perfect bell shape of the normal Gaussian distribution. Figure 3.4 is based on uniform random 3-SAT formulas, where we need about 1,000 instances to find 100 instances with solutions in the range of 10-64. Figure 3.5 is drawn based on instances we generate to have the 10-64 solutions "randomly" distributed. Figure 3.5 looks much more similar to Figure 3.3 than Figure 3.4, which implies that the formulas we generate have random solutions in terms of the pairwise Hamming distance. Furthermore, the formula's randomness has also been tested as shown in Figure 3.8.

We start with k randomly selected solutions and then find *local clauses* that all the k solutions satisfy. By the term *local clauses*, we mean clauses consists of literals only from 10 pre-selected random variables. These clause will eliminate part of the non-solutions. We repeat this for about 100 times to ensure the

randomness of clauses and the elimination of all non-solutions. Next, we remove the redundancy from clauses. A clause $\mathcal{C}$ in formula $\mathcal{F}$ is *redundant* if $\mathcal{F} = \mathcal{F} \backslash \mathcal{C}$, the formula obtained by removing $\mathcal{C}$ from $\mathcal{F}$. Whether $\mathcal{C}$ is redundant can be determined by checking the satisfiability of formula $\mathcal{C}'(\mathcal{F} \backslash \mathcal{C})$. After removing all the redundancy, we observe that there are around 180 clauses left. We then gather the statistical information about this formula and add extra clauses to make the total number of clauses 218 and make the formula look as "random" as possible.

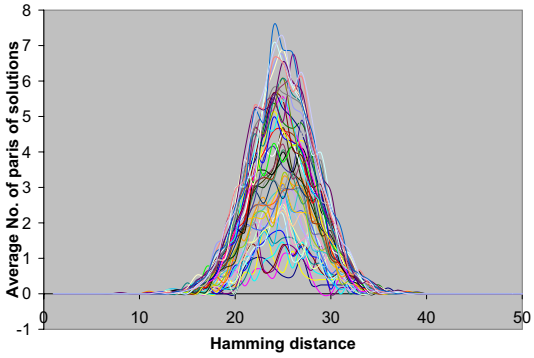

Figure 3.3: Solution randomness of randomly selected solutions.

The existence of solutions with extremely small Hamming distance (around 2-4) in Figure 3.5 is due to the fact that it becomes almost impossible to eliminate all the non-solutions when large number of solutions are selected randomly. We compensate this by selecting 10-20 random solution and including their neighbors as solutions if the desired number of solutions is large.

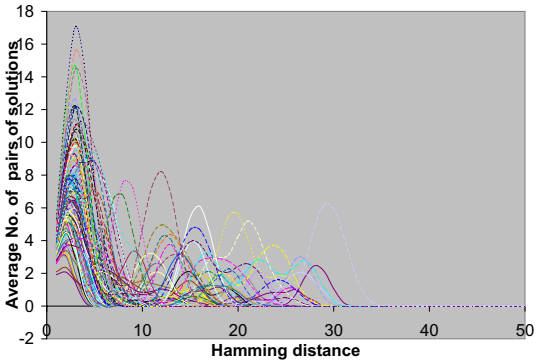

Figure 3.4: Solution randomness of uniform random 3-SAT instances.

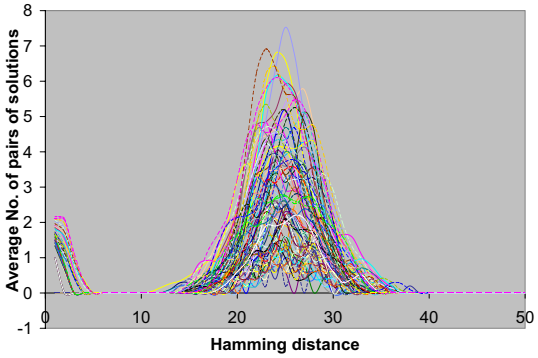

Figure 3.5: Solution randomness of instances with random solutions

Input: Number of variables - n,Number of solutions - S,

Output: 3SAT instance with $4.3n$ clause list F

```
Generate a set of r random solutions
Generate S − r random neighbors to r solutions
for ( m times ){
    Pick random k variables
    Select solutions assignments for the corresponding variables
    Obtain the non solutions for the selected solutions
    Let all the possible satisfying clauses C
    While ( non solutions and satisfying Clauses are not empty){
        choose a random clause from the list C
            if the clause kills at least one non solution
                add to the Final clause list F
            delete the clause from the list C
        }
    }
Select a clause randomly from list F
Check for redundancy and delete if redundant
Repeat until every clause is checked for redundancy
Output F
```

Figure 3.6: Generation of instances with random solutions

3.4 Generation of Random Benchmarks

In this section give the motivation for the need of Benchmarks with known solution space.

Finding all of the solutions for the randomly generated 3SAT instances is time consuming. Another key problem is only after finding all the solutions one can know exactly how many solutions exist. To find a relation or phase transition based on the number of solutions we require a fair number of instances for various number's of solutions. We generated 100,000 instances with random distribution of variables at the critical value, 4.3 clauses to variable ratio. Only 50 % of the benchmarks were satisfiable as expected at the critical value. We found all the solution for those benchmarks which had 5000 or lesser number solutions, to save time. For the benchmark which had more than 5000 solutions, we terminated the search since it took quite a long time. Around 1% of the instances has solutions more than 5000. We counted the number of benchmarks with the same number of solutions and plotted them against the corresponding number of solutions. We can see from figure 3.7, there are number of solution points which do not even have a single instance to represent. Also there is a lot more instances for fewer solutions than for the higher number of solutions. It won't be fair, if a conclusion is drawn based on this histogram of Instances. Our benchmark generator algorithm can generate benchmarks for a given number solutions. So we can generate as many benchmarks as need at various number of solutions.

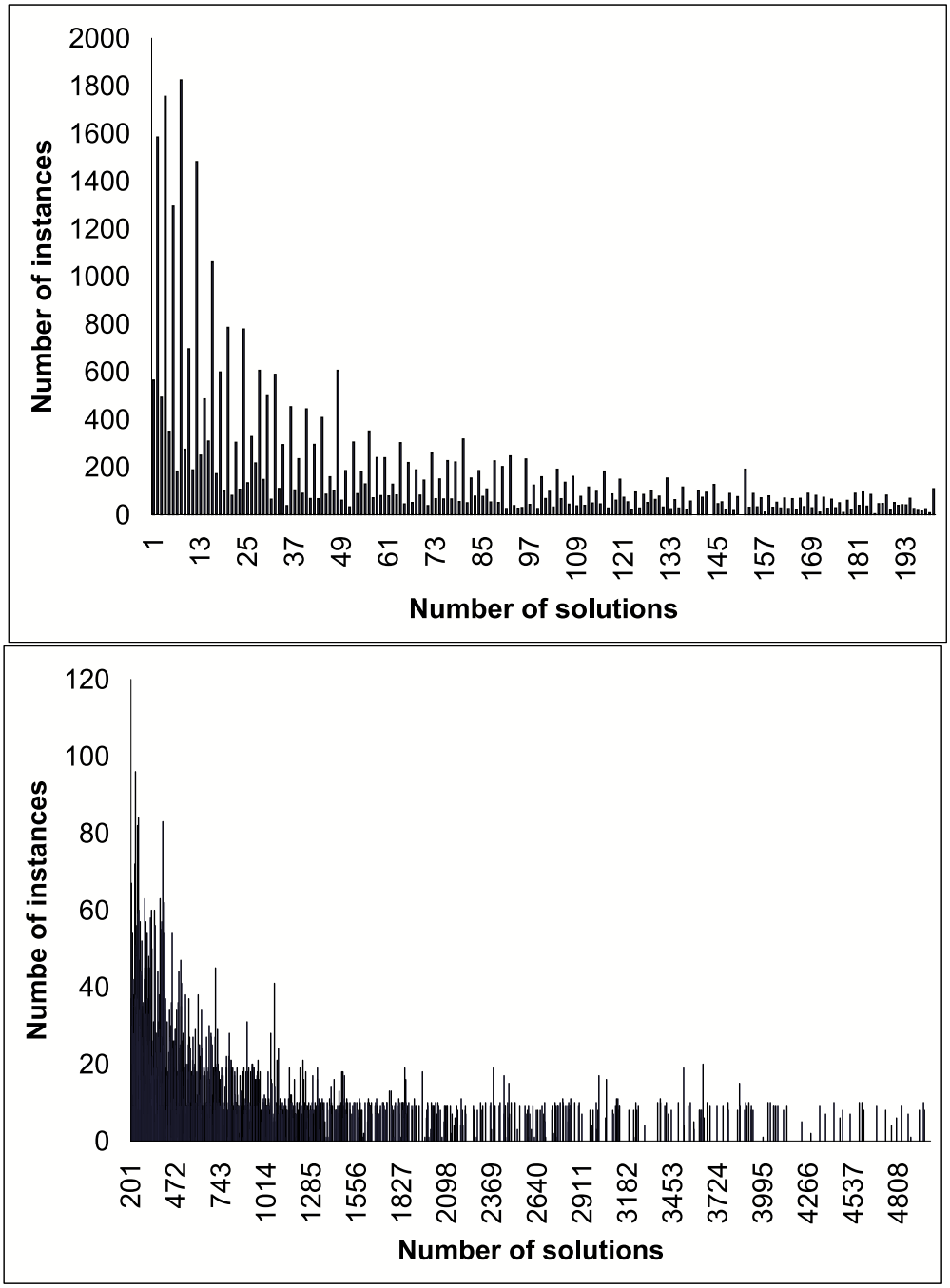

Figure 3.7: Distribution of randomly generated instances over number of solutions

3.5 Generation of Benchmarks with Solution Space

In this section we evaluate the properties claimed for the benchmark generator especially the randomness of the generated instances and the efficiency of the algorithm to generate instances for a given number of solutions within a tolerance range.

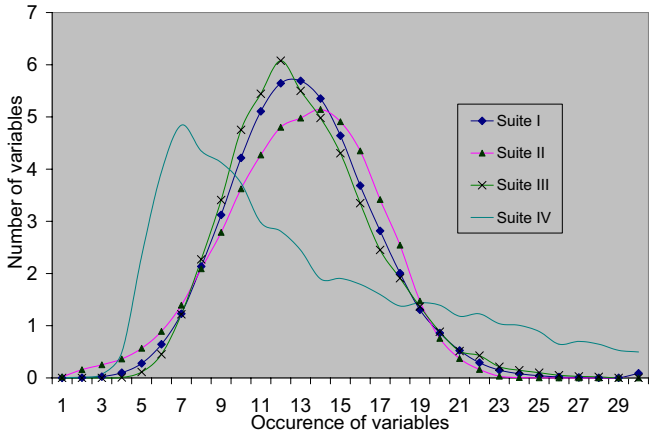

Figure 3.8: Clause randomness of 3SAT instances. Suite-I: Randomly generated instances, Suite-II: Instances with Specific number of solutions, Suite-III: Instances with Random solutions, Suite-IV: Not a Random 3-SAT

Randomness of the 3SAT instance is based on the distribution of variables; the number of times a variable appears in the instance, occurrence of pair of variables and the occurrence of triplets. The distribution of variables is evaluated by counting the number of variables whose occurrence is the same. We plot this

occurrence of variables to the corresponding number of variables. For example; consider a 50 variable 3SAT instance, where variables 3, 5, 19, 20, 38 occur 6 times. Then the number of variables which occur 6 times is 5. Occurrence of all the variables are thus counted and grouped respectively. In figure 3.8 the x axis represent the number of occurrence and the y axis represents the number of variables which has the corresponding occurrence. We compare the randomness of the generated benchmarks with the randomness of the pure random variables explained in the chapter 1. The curves represent the average occurrence over 100 benchmarks. we can see that the curve is pretty close with the randomly generated 3SAT clauses. Thus our generated benchmark resembles the randomly generated benchmarks.

The benchmark generator can be used to generate instance with solution space within a desired range. To test the benchmark generator we generated 3SAT instances with 50 variables and 218 clauses. The desired solutions range for these benchmarks are 100-200,250-350, 500-600, 750-850, and 1000-1200. Around 400 benchmarks were generated in these range. We counted the number of benchmarks within these range which had the same the number of solutions and plotted them again the number of solutions. Figure 3.9 shows the benchmarks generated for various number of solutions using our algorithm. We can peaks at these ranges showing that benchmarks were generated at these points. The run time for generating these benchmarks are far less than finding all the solutions.

Since we know the solution space, we can easily find the group or cluster

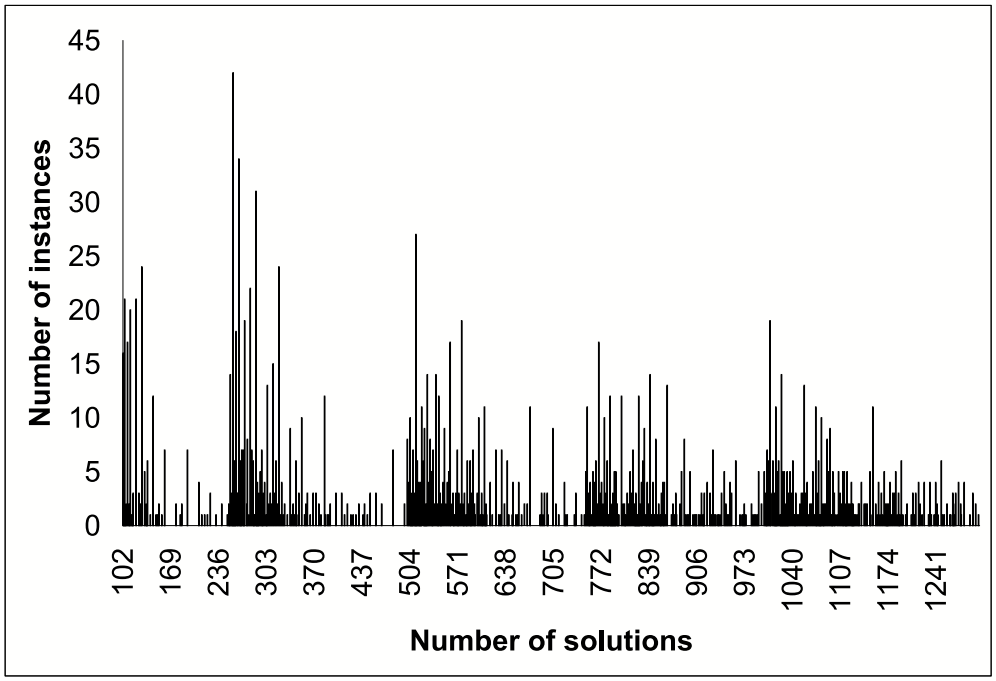

Figure 3.9: Distribution of instances generated by benchmark generator over number of solutions

information of these solution space. We used espresso tool to calculate the prime implicant (PI) information of these solution space. The number of prime implicants in a solution space corresponds to the groups of solutions. For the generated benchmarks, we calculated the PI of the solution space. The figure 3.10 shows the histogram of the number of groups obtained from the generated instances. x axis is the number of groups and the y axis is the number of instances for each group. Since the number of groups is not controlled by the algorithm, there is a lot of instances with fewer groups. As a future work we would like to consider the addition of groups as another constraint in generating

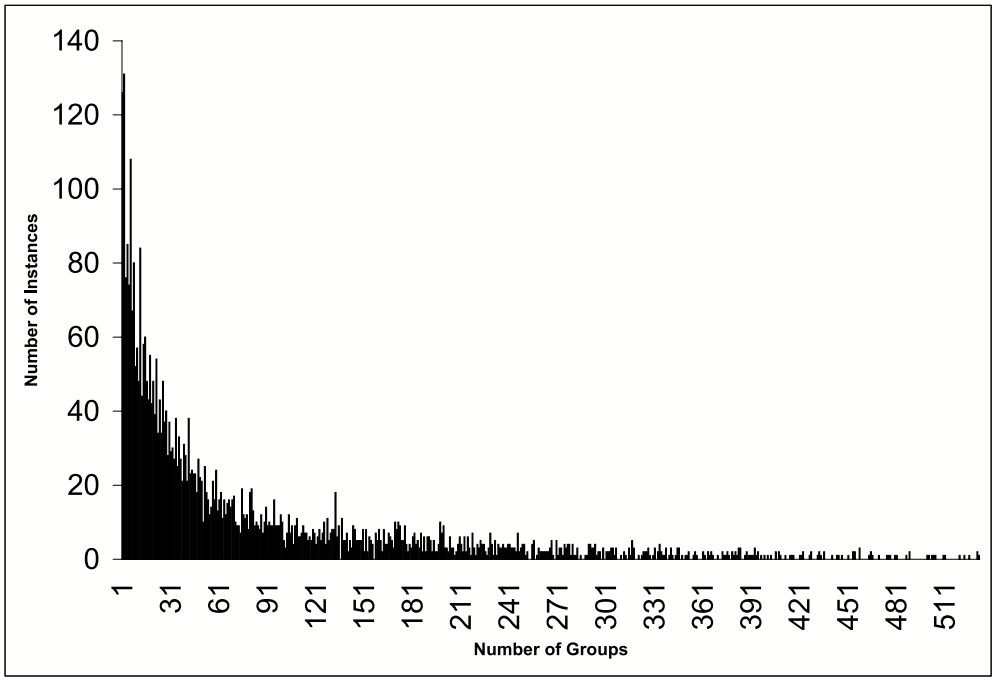

Figure 3.10: Distribution of instances generated by benchmark generator over number of solution groups

benchmarks.

3.6 Motivation for Hard benchmarks

Our motivation for this benchmark generator is to see if there could exist any relation between the hardness of an instance to the number of solutions. We generated instances with solution ranges 1-10, 1-100, 100-200, 250-350, 500-600, 750-850, 1000-1200, 2000-2200, 4000-4300 and 5000-5300. The benchmarks generated are made of 50 variables and 218 clauses and around 300 benchmarks for each range. All the benchmarks are in the critical region. We solved each of this benchmarks using zchaff. We used the number of decisions, to find the first solution as a metric to hardness. Figure 3.11 gives the average decision for a group of solutions space. The graph shows a trend of increase in hardness as the solution space decreases.

We also calculated the group information for these benchmarks and plotted them against the hardness. Figure 3.12 shows that as the groups tend to decrease there is an increase in the hardness of the benchmarks. From these preliminary results, it appears when the solution space is fewer and when the number of groups are fewer, the instances are relatively hard. To generalize this we need to do more experiments for various other number of variables. As a next step we would like to control the number and size of the solution groups that exist in the solution space. To find a phase transition or cross over point based on the number of solutions we required to do more extensive experiments. Now we can

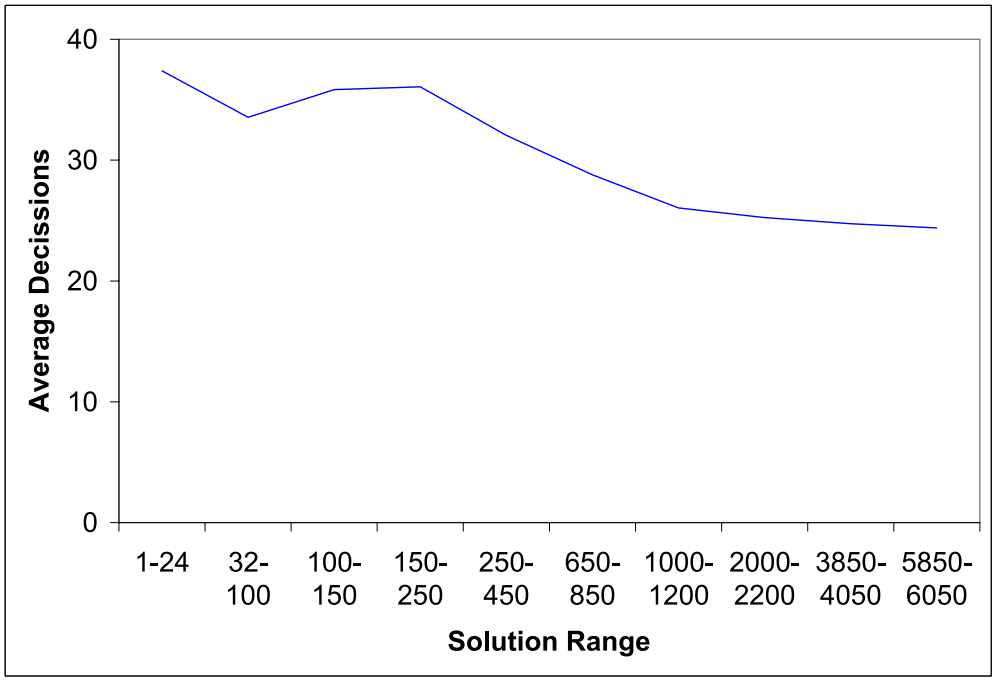

Figure 3.11: Hardness Vs Solution range

use this benchmark generator for that purpose and generate instances of required number of solutions and number of variables. Also we need test the hardness over the entire critical region, from clause to variable ratio 4 to 5, to explore the possibility of relating the hardness to the solution space. We believe that this generator algorithm will pave way to research the effect of solution space in satisfiability problem.

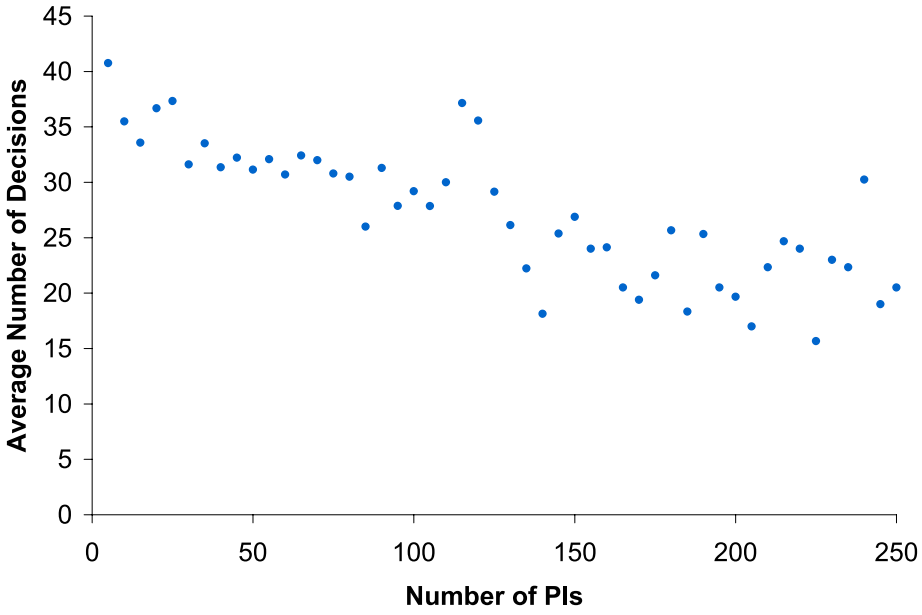

Figure 3.12: Hardness Vs Groups of solutions

Chapter 4

Conclusion And Future Work

For more than a decade the only metric to characterize hardness is the crossover point. We showed that the random 3SAT instances generated at this region have more number of easy problems. This shows that crossover point is not the only metric to characterize hardness. We conjecture that solution space can also be related to hardness. The key challenge for this conjecture is to provide the solution space information for the benchmarks. We present sampling techniques to estimate solution space and an algorithm to generate random 3SAT benchmarks with known solution space. We test the two estimation techniques on the satlib benchmark instances. The results show that they are fairly accurate. Using these techniques solution information can be obtained for available benchmarks. For an accurate study of solution space and its relation to hardness, we can use the benchmark generator algorithm. We show that our benchmark generator is capable of generating random 3SAT instances within a given solution range. Based on the benchmarks generated, we show that instances with fewer number of solutions are relatively harder. More experiments

need to be done to relate the hardness to solution space by generating more instances for various number of variables and number of solutions. As a future work, we would like include the group information as a constraint in our benchmark generator algorithm. Using the benchmarks generated for various solution range we can study the approach of solvers and improve their efficiency. Hence we believe that our benchmark generator will contribute to the study and understanding of the SAT better.

BIBLIOGRAPHY

[1] J.M. Crawford and L.D. Anton. Experimental results on the crossover point in satisfiability problems. In Richard Fikes and Wendy Lehnert, editors, *Proceedings of the Eleventh National Conference on Artificial Intelligence*, pages 21–27, Menlo Park, California, 1993. AAAI Press.

[2] P. Cheeseman, B. Kanefsky, and W.M. Taylor. Where the Really Hard Problems Are. In *Proceedings of the Twelfth International Joint Conference on Artificial Intelligence, IJCAI-91, Sidney, Australia*, pages 331–337, 1991.

[3] D.G. Mitchell, B. Selman, and H.J. Levesque. Hard and easy distributions for SAT problems. In Paul Rosenbloom and Peter Szolovits, editors, *Proceedings of the Tenth National Conference on Artificial Intelligence*, pages 459–465, Menlo Park, California, 1992. AAAI Press.

[4] M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of ACM*, 7(3):201–215, 1960.

[5] H. Hoos 1999. Satlib. A collection of SAT tools and data. See www.informatik.tu-darmstadt.de/AI/SATLIB.

[6] A. Urquhart. Hard examples for resolution. *Journal of the ACM*, 34(1):209–219, 1987.

[7] V. Chvatal and E. Szemeredi. Many hard examples for resolution. *Journal of the ACM*, 35(4):759–768, October 1988.

[8] Y. Asahiro, K. Iwama, and E. Miyano. Random generation of test instance with contolled attributes. In M.A. Trick D.S. Johnson, editor, *Cliques, Coloring, and Satisfiability*, volume 26 of *DIMACS Series in Discrete Mathamatics and Theoretical Computer Science*, pages 377–393, 1996.

[9] A. Horie and O. Watanabe. Hard instance generation for sat. In *Eight International Symposium on Algorithms and Computation*, pages 22–31, 1996.

[10] M. Motoki and R. Uehara. Unique solution instance generation for the 3sat satisfiability (3sat) problem. Technical report, Dept. of Mathamatical and Computing Sciences. Tokyo Institute of Technology, 1999.

[11] 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.

[12] J. Franco, and Y.C. Ho. "Probabilistic Performance of A Heuristic for the Satisfiability Problem", *Discrete Applied Mathematics*, Vol. 22, pp. 35-51, 1988.

[13] J.P. Marques-Silva and K.A. Sakallah. "GRASP: A Search Algorithm for Propositional Satisfiability", *IEEE Transactions on Computers*, Vol. 48, No. 5, pp. 506-521, May 1999.

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

[15] M.W. Moskewicz, C.F. Madigan, Y. Zhao, L. Zhang, and S. Malik. "Chaff: Engineering an Efficient SAT Solver", *38th ACM/IEEE Design Automation Conference*, pp. 530-535, June 2001

[16] 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.