Design and Implementation of Efficient Techniques for All-SAT

Dual Degree Dissertation

Submitted in partial fulfillment of
the requirements for the degree of

Bachelor of Technology
(Electrical Engineering)
&
Master of Technology
(Micro Electronics)

By
Tejas Nikumbh
(10D070019)

under the guidance of
Supervisor. Prof. Virendra Singh
Co-Supervisor. Prof. Shankar Balachandran

DEPARTMENT OF ELECTRICAL ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY, BOMBAY

JUNE 2015

Approval Sheet

This dissertation entitled '**Design and Implementation of Efficient Techniques for ALLSAT**' by **Tejas Nikumbh** is approved for the degrees of **Bachelor of Technology** and **Master of Technology** (Dual Degree).

Examiners

_____

_____

Supervisor

_____

_____

Chairman

_____

Date: _____

Place: Indian Institute of Technology Bombay, Mumbai

Declaration

I declare that this written submission includes my ideas in my own words and where others' ideas or words have been included, I have adequately cited and referenced the original sources. I also declare that I have adhered to all principles of academic honesty and integrity and have not misrepresented or falsified or fabricated any idea/data/source/fact in my submission. I understand that any violation of the above will be cause for disciplinary action by the Institute and can also evoke penal action from sourced which have thus not been properly cited or from whom permission has not been taken when needed.

_____

Tejas Nikumbh
10D070019
Dept. of Electrical Engg.
IIT Bombay

Date : 14th June 2015

Acknowledgements

I would like to thank Prof. Virendra Singh for his guidance and motivation throughout the project. I also thank him for being a very considerate professor and someone who actually cares for his students.

I would also like to thank Prof. Shankar Balachandran, for giving the right insights and directions at various stages of the project. He has been a mentor and this thesis would not have been possible without his continued support.

I would like to thank my parents who have made innumerable sacrifices for making me capable enough of reaching here.

Last, but definitely not the least, I would also like to thank Prof. Janak H. Patel and all the people in the CADSL group at IIT Bombay, who have been instrumental in providing valuable inputs from time to time.

Tejas Nikumbh

Abstract

The All-SAT problem is defined as the task of finding all satisfying assignments for the variables involved a given propositional logic formula. This problem has great importance from point of view of Formal Verification, Unbounded Model Checking and other applications in the VLSI domain. There are two ways to approach this problem, with BDDs (which are graphical structures that represent all solutions to a SAT instance in parallel) and Sequential All-SAT solvers. We focus on Sequential All-SAT solvers. We particularly implement 'All-SAT Using Minimal Blocking Clauses', a novel technique based on complexity theory for All-SAT solving. We then profile the Implementation and identify issues related to the implementation. We suggest solving the issues using predictive techniques for SAT, that are applicable to the solver and beyond.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The Boolean Satisfiability problem, more popularly known as SAT is the problem of finding an assignment to a set of Boolean variables such that a particular propositional logic formula or SAT Instance of these variables is evaluated to true. The variable assignment is then called a 'satisfying solution' that satisfies the given propositional logic formula. The All-SAT problem is the problem of finding all such satisfying solutions.

SAT is one of the first problems that was proven to be NP-Complete. [Non Deterministic Polynomial Time] These are set of problems that are both NP and NP Hard. There is no known algorithm that could solve the SAT problem, let alone All-SAT in polynomial time. The All-SAT problem has a number of applications in Formal Verification, Unbounded Model Checking and other applications in the VLSI Domain. There are two ways to approach this problem.

The first of them is BDD's or Binary Decision Diagrams which we discuss later. These are graphical structures that represent all satisfying solutions in parallel. The second, more traditional approach is to use Sequential SAT solvers, many of which use the DPLL Approach as discussed later. In the first Chapter, we go on to discuss the motivation for thesis, following which we discuss some related work and give a general organizational overview of the thesis in the last section.

1.1 Motivation

With the growing applications of SAT, it is becoming very evident that there exists an urgent need to design efficient and fast techniques for finding all satisfying solutions of a SAT Instance. In this thesis, we try and build upon the recent most novel techniques that have made use of intuition from complexity theory to develop clever algorithms for All-SAT. An in depth exploration of such techniques and ideas would not only better out understanding of the structure in the problem, thereby helping us implement them better , but also point us in the right direction with regards to improvement in the algorithm. It is with this motive in mind that this thesis was written and the project was undertaken.

1.2 Related Work

As mentioned earlier, there are primarily two ways to deal with the All-SAT problem. The first one of these is using Binary Decision Diagrams. Binary Decision Diagrams or BDDs are graphical structures that represent all solutions of a Boolean propositional logic formula in parallel. One major drawback of BDD's is that they require a large amount of memory. This is primarily due to the exponential complexity of number of nodes (exponential in the number of variables) required to represent the BDD and also the Hashing requirement. Hence, efforts towards developing an approach that helps make BDDs require less memory are needed for large scale simulation. From the time complexity point of view, due to hashing techniques being used, BDDs provide constant time checking for Tautology and Linear Time Checking for satisfying assignments. This is one benefit they have over Sequential SAT solvers. The time complexity for declaring that a particular Boolean Formula is UNSAT (no satisfying assignment can be found), is, however, exponential. We give an overview of BDDs in Chapter 2.

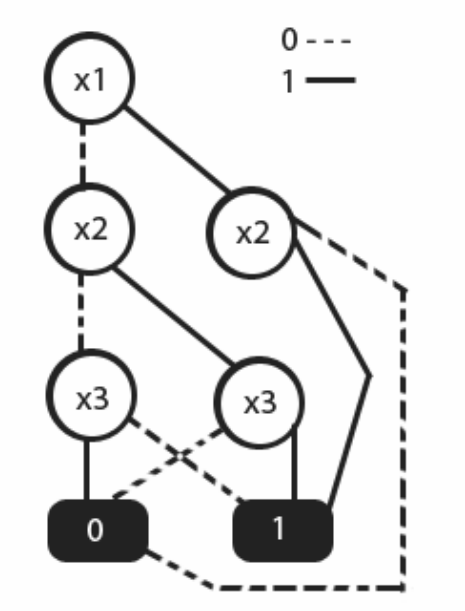


Figure 1: ROBDD Example

The basics of construction of a BDD is very well described in Anderson's Notes [1] .A well thought out blueprint that covers all aspects of a good BDD package is described in the paper – 'Efficient Implementation of BDD Package'[2]. The basis for the current State of the Art BDD Package – CUDD is described pretty well in the paper by Somenzi, Fabio[3]. There have been certain efforts to contain the memory tradeoff of the BDD Approach. One such effort is the Streaming BDD[4]. Although it has it's limitations, it makes for a good read if one wants to pursue that direction of research.

The crux of this dissertation however centers on the more traditional second approach to All-SAT, the Sequential All SAT solvers. Sequential All SAT solvers are better than BDDs because they don't have the memory constraint. They don't have to store all solutions in the stack memory in parallel. In fact, since each solution/set of solutions is produced one at a time, the solutions that these solvers produce can also be stored in a file as they are deduced by the program. Sequential All SAT solvers have to use an underlying SAT Solver as a basis, and there are many of them. One such solver that is popular in academia is MiniSAT [5] and we use it in our implementation and experiments.

Harnessing the power of the underlying Solver, a traditional approach at a Naïve All-SAT Algorithm template is first detailed. A very recent novel technique based on Complexity theory that dramatically improves the performance of the All-SAT Solver can be found in the paper by Yu,Yinlei et al[6]. Finally we make use of heuristics to predict whether a SAT Instance would be solvable or not, as a part of avoiding timeout and performance enhancement to Novel Techniques. The Heuristics we use are described well in the paper on SATZilla by Xu ,Lin et al[7] The classification techniques that we use as a part of the prediction can be found in references [8], [9] , [10] , [16] and [17].

1.3 Organization of the thesis

1.3.1 Organization Diagram and Focus Area

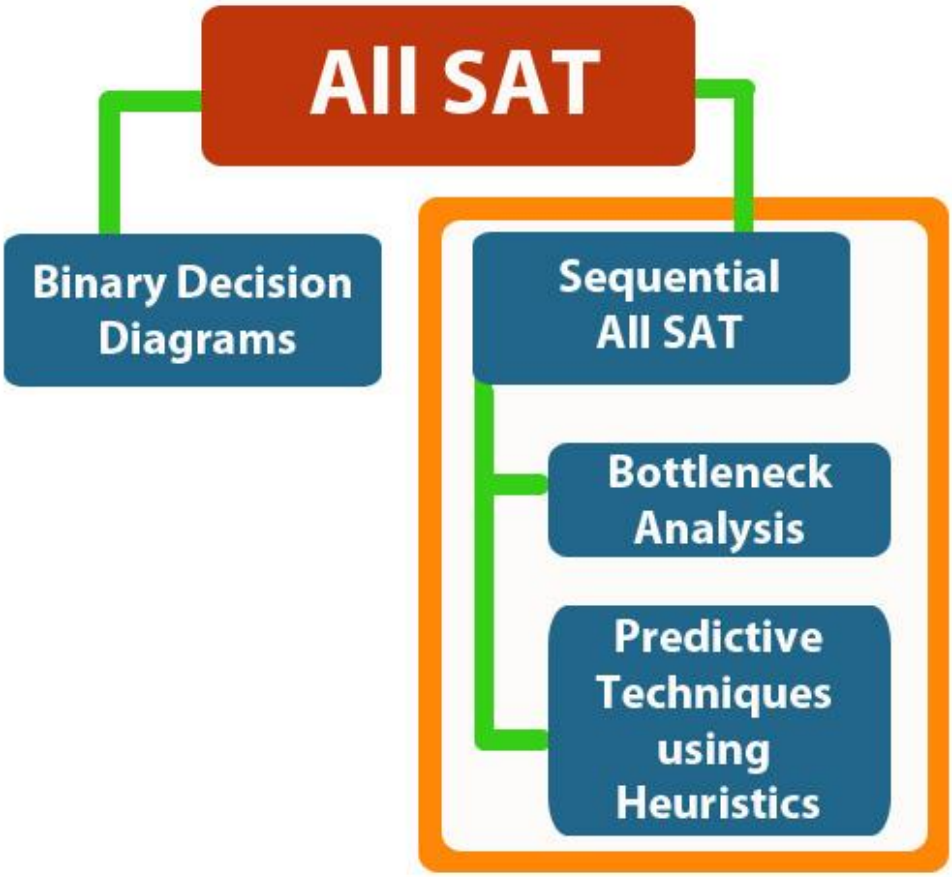


Figure 2: Thesis Organization

There are essentially two ways to approach the All SAT Problem, as mentioned earlier. They are BDD's and Sequential SAT solvers. The more traditional approach is Sequential SAT Solver. We first get an overview of the BDD approach, and make certain observations as well as conclusions. We also provide pointers to the inquisitive reader to explore more. The focus area of this dissertation is however, the Sequential SAT Solving techniques, in which we study a very recent ALL SAT Technique using Minimal Blocking Clauses that is providing promising results. We create an implementation of the technique in Python [there is no publicly available implementation of the technique, to the best of the author's knowledge, at the time of writing of this thesis] and run simulations on standard SAT Benchmarks. We then figure out issues related to the technique and analyze it's performance bottlenecks. We suggest improvements to the technique by using predictive techniques to resolve the timeout issues we encountered. We now provide a chapter wise overview of the dissertation.

1.3.2 Chapter Wise Overview

We begin by looking at Binary Decision Diagram's in Chapter 2. We explore the notion of BDD's and how a Boolean Formula can be represented as a Binary Decision Diagram. We first draw the relation between a BDD and finding All-SAT Solutions. Later, we describe the basic procedures followed in creating a BDD from a given Boolean Formula and go into the finer details in implementing a BDD Package. We then move on to point to further research in the area for the inquisitive reader.

In Chapter 3, we explore the Sequential All-SAT Solver techniques. We first introduce the reader to the CNF Form, a form which is popularly used by benchmarks for SAT. We then describe how a SAT problem is typically solved using the DPLL approach, a template for the algorithm that many SAT solvers today use for computing one satisfying solution of a given SAT Instance. We go on to describe MiniSAT, a minimalistic SAT Solver popular in academia because of it's ease of understanding and implementation. Post this we explore the Algorithm template for finding all satisfying solutions of a given SAT Instance. We then move

on to study the Novel Technique based on Complexity Theory that improves upon the Algorithm Template and discuss the intuition behind it. This chapter provides a head start to the real implementation of the technique which we describe in greater detail in Chapter 4.

In Chapter 4, we describe the implementation details for the All-SAT Solver package that was built as an important part of this project. We then move on to discussing the experimental setup and benchmarks used in the simulations.

In Chapter 5, we describe the results obtained as a result of running the simulations on the benchmarks. We identify issues with the solver, and discuss the bottlenecks. We then propose to make use of a SAT Classifier based on prior research to better solve the identified issues. We build 5 different Classifiers for all 3 categories of SAT problems as described in the benchmarks. We also introduce an additional benchmark category. Furthermore, we suggest a strategy to make the classifier more robust to changes in the algorithm and improve it's performance. We then discuss the results obtained by using the strategy. We call this strategy the Majority Rule.

Finally, in chapter 6, we draw conclusions about the project and explain scope for future work in the area. We explain further use of the classifier's we developed in different SAT solvers. We also explain scope for parallelization in the project. We finally conclude by mentioning all the references that were helpful in the project.

Chapter 2

Binary Decision Diagrams

2.1 Introduction to BDD's

Binary Decision Diagrams or BDDs are Data Structures based on the INF or If Then-Else logic paradigm. They are essentially graphical representations of a Boolean Formula. To construct a BDD from a Boolean Formula, we need to specify variable ordering on the Boolean Formula. Once this order is specified, we proceed to construct the BDD in the following way. We take the variable with the first priority in the order, and treat it as a node. Let's call this node root. Once this node is stored in memory, we construct two more nodes, one of which is selected 'if' the assignment of the prior node is 0, the other, if the assignment is 1. Both of these nodes are representative of Boolean Formula's procured by replacing the Boolean Formula represented by root with 0 and 1 respectively.

It is clear from the above explanation that the BDD will essentially create a tree-like structure. The leaves of this tree are always 0 or 1, which is self-explanatory since if all the variables in a Boolean Formula have an assignment, the Boolean Formula will evaluate to either 0 or 1. Every node apart from the leaf nodes in a BDD is representative of a Boolean Formula. Also, it follows all recursive properties like that of a tree. Figure 3 illustrates a graphical BDD representation of the Boolean formula

$$f(x_1, x_2, x_3) = \overline{x_1}\,\overline{x_2}\,\overline{x_3} + x_1x_2 + x_2x_3$$

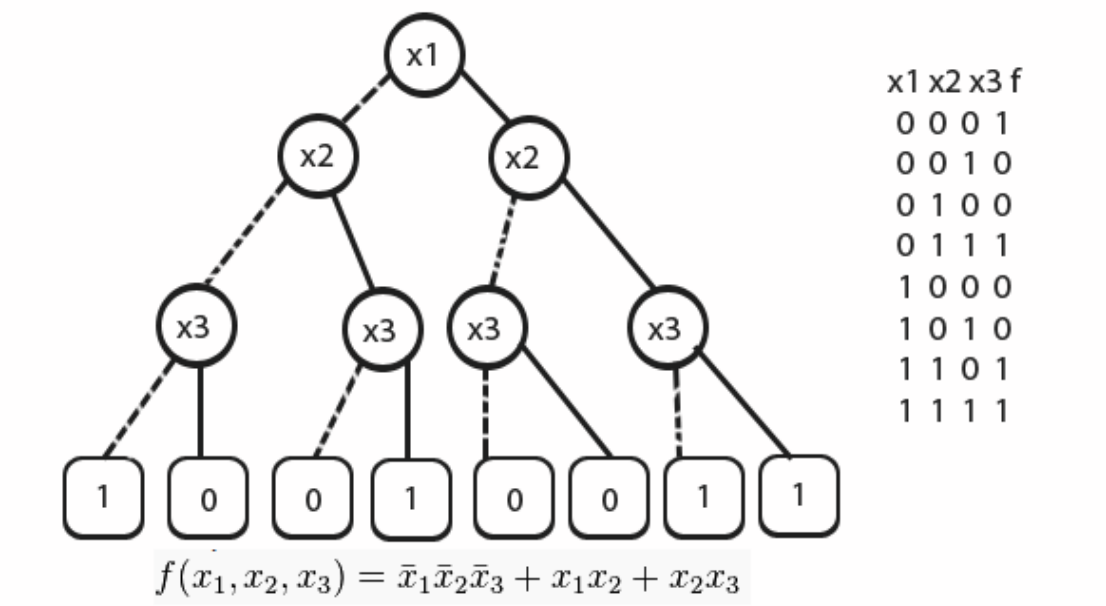


Figure 3: OBDD Example 2

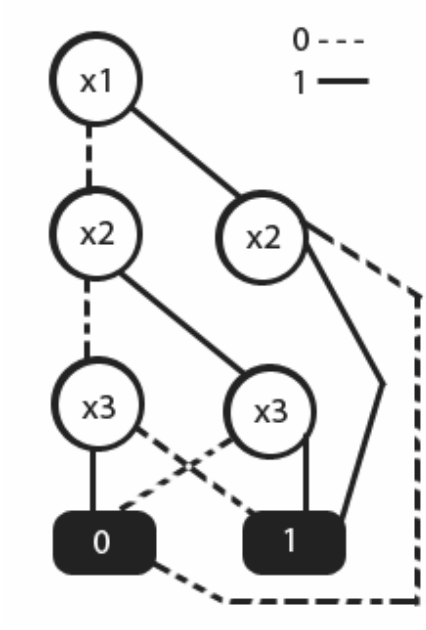


Figure 4: ROBDD Example 2

This is a very naïve explanation of how a BDD is constructed from a Boolean formula, in program memory. Figure 3 essentially represents what is called an Ordered Binary Decision Diagram. There are many nodes that are redundant in the Ordered Binary Decision Diagram, nodes that represent the same Boolean Formula. Such nodes are clubbed together to form what is called a reduced Binary Decision Diagram. A reduced version of the above BDD in figure 3 is given in Figure 4. A reduction process is essentially performed with the help of a table that hashes and stores nodes that are created as the BDD is constructed in memory, and each time we try to insert a node, checks if the node is already present in the Hash Table. If yes, it doesn't re-create the node but points to it instead. In this way, we avoid redundancy in nodes to create a ROBDD as in Fig 4

There are measures to avoid redundancies and make the BDD more efficient. Reference [1], Anderson's notes, particularly deals very well with giving the reader a head start on BDD's. Particularly important operations on BDD's as described in the paper are: -

1. Build Operation : - For constructing BDD from a given Boolean Formula
2. Apply Operation : - For performing specified operation on given BDD's
3. Restrict : - For performing quantification operation on a BDD
4. SAT Related Operations : - These include methods that help compute number of SAT Solutions, one SAT Solution, and All SAT Solutions of the Boolean Formula represented by the Binary Decision Diagram

Anybody wanting to pursue research in this area is encouraged to go through them once. ROBDD's are described in greater detail in Reference [11].While dealing with BDD's it is practice to call ROBDD's as BDDs since we assume that the BDD we are dealing with is reduced.

2.2 Limitations of BDD's

BDD's, in spite of being all mighty and powerful, are not that popular in SAT Based Applications. The prime reason for this is humongous memory requirement for the storing the nodes in even ROBDD's. Hence, while we get all solutions in parallel and testing for Tautology is rapid [since tautology would represent only one node labelled '1'], construction of BDD's in memory is a problem.

There have been some efforts from point of view of limiting the size of the BDD in memory by usage of a streaming model. However, even this approach has it's cons in the form of 'Inability of Partial Assignment', 'Dependence on BDD structure' etc. The inquisitive reader is suggested to refer to Reference [4]

Chapter 3

Sequential All-SAT

Having seen the limitations of the BDD Approach, we now move on to the crux of the dissertation, the Sequential All SAT Approach. This Approach has a number of benefits. Since it does not have to store all solutions at once in memory, memory requirement from that point of view is less. Also, we don't have to create nodes as in the BDD Approach for this approach to work. It is primarily due to this that there is no humongous memory requirement unlike BDDs.

In this chapter, we first give the reader a brief overview of the basis of SAT Solving techniques in general and related terminology. These are techniques for finding a single satisfying solution to a given Boolean Formula. We go over one of the most popular and basic approaches to solving SAT problems – the DPLL Algorithm. This would give the reader a good idea about how traditional SAT solvers like MiniSAT work. Post that, we describe MiniSAT itself– a minimalistic SAT Solver that utilizes DPLL as the underlying algorithm to obtain a satisfying solution to a Boolean formula. We do this since we use MiniSAT python bindings in our implementation. Following this, we introduce the traditional way Sequential All SAT solvers work, by describing the Naïve All SAT Algorithm. We also introduce the idea of disjoint and overlapping cubes. Finally, we conclude the chapter with an in depth explanation of the overlapping cubes approach used in

the paper on All SAT using Minimal Blocking Clauses[6] which lays the foundation for the Implementation work done in the next chapter.

3.1 Finding a single SAT Solution

There is one important thing to discuss before we go on to discuss Sequential SAT solvers. Whenever dealing with solving a SAT problem, or finding all solutions to it in general, we are generally given the SAT Boolean formula in CNF form. CNF stands for Conjunctive Normal Form and basically means AND of ORs. Given below is an example of a Boolean Formula in CNF Form.

$$(x_1 \ V \ x_2 \ V \ x_3) \ \Lambda \ \left(x_2 \ V \ \bar{x}_1 \ V \ x_3 \right) \ \Lambda \ (\overline{x_4})$$

We assume the following problem definition: Given a Boolean Formula in CNF form, find all satisfying solutions to the Boolean Formula.

An important part in finding all satisfying solutions to a Boolean formula in CNF form is actually being able to find one satisfying solution to the Boolean formula. We study this problem first.

3.1.1 DPLL Algorithm

DPLL stands for Davis-Putnam-Logemann-Loveland. This Algorithm has a worse case time complexity of O(2^n) and worse case space complexity of O(n). The algorithm is essentially an enhanced form of the backtracking procedure demonstrated as follows.

Given a formula in CNF form, we select a variable and assign It a particular truth value. Then, substituting this value in the Boolean Formula, we obtain a partially assigned Boolean Formula containing the assignment of the first literal. Given that this obtained formula is a tautology or has already evaluated to 1, the search terminates. If not the same recursive procedure is carried out on the Boolean Formula obtained. If after exhaustive searching, the '1' branch evaluates

to false, we do the same process by assigning the first chosen literal the value of 0 and the process is recursively repeated.

The DPLL Algorithm builds up on this backtracking procedure primarily by introduction of two important phases, namely :-

Unit Propagation: Unit propagation corresponds to a clause containing only one literal, in which case we HAVE to assign that literal a truth value of 1, because there is no other way in which the clause can be satisfied. Also, since this literal is assigned the value of 1, all other clauses containing this literal are pruned since they also become true. Such a procedure at this beginning greatly prunes the search space in the CNF Boolean formula

Pure Literal Elimination: A pure literal is a literal that occurs in only one form throughout all the clauses. A value to this literal can always be assigned such that the clauses that contain it all evaluate to true. So, such clauses are removed and the search space is pruned.

Given below is the pseudo code for the DPLL Algorithm. Here **CNF** is the input Boolean formula in CNF form and the algorithm outputs a truth assignment for **CNF** that satisfies it. Reference: - Wikipedia

```
function DPLL (CNF):

    if CNF is consistent set of literals
        then return true;

    if CNF contains empty clause
        then return false;

    for every unit clause c in CNF
        CNF := unit_propagate(c, CNF)

    for every literal l that occurs pure in CNF
        CNF := pure_literal_assign (l, CNF)
    l := choose_literal(CNF)

return DPLL (CNF^l) or DPLL (CNF^not (l));
```

The algorithm as described on the last page returns only whether a given SAT instance in CNF form is satisfiable or not. In case we also want to know the set of assignments that made the CNF Form satisfiable, we need to return them in every iteration of the recursive calls of the function by returning them from the first if statement. We also need to add with them the unit clause literals and pure literals we assigned true. Hence the modified algorithm pseudo code would look somewhat like what is given below

```
function DPLL (CNF):

    if CNF is consistent set of literals
        then return [true, set_of_literals];

    if CNF contains empty clause
        then return [false, empty_set];

    for every unit clause c in CNF
        CNF := unit_propagate(c, CNF)

    for every literal l that occurs pure in CNF
        CNF := pure_literal_assign (l, CNF)
    l := choose_literal(CNF)

    [result_1, set_1] = DPLL (CNF^l)
    [result_0, set_0] = DPLL (CNF^not (l)))
    uc := set of all unit clauses in CNF
    pl := set of all pure literals in CNF

    if result_1 is true
        return [true, set_1 | uc | pl | l]
    else if result_2 is true
        return [true, set_0 | uc | pl | -l]
    else
        return [false, empty_set]
```

Note that in case both result_1 and result _2 are true in the second modified algorithm, we simply return the results of result_1, since such a scenario indicates that any assignment on literal l is producing a satisfying solution and we can choose any one of the resulting set of assignments.

Building up on DPLL, we now briefly discuss CDCL, which is Conflict Driven Clause Learning. The prime difference between the two techniques is the backtracking. We briefly outline the CDCL work flow.

CDCL consists of the following steps: =

1. Select a variable and assign it True or False. Call this the Decision state and remember this assignment.
2. Do Unit propagation as described above
3. Build Implication graph based on the unit propagation
4. If there's a conflict, based on analysis, backtrack to appropriate decision level
5. Otherwise continue from step 1 until all variables are assigned.

Information in greater detail about this modification can be found in reference [12] , however, details about inner workings of CDCL are not essential to understanding this project.

3.1.2 MiniSAT

For sequential SAT Solving, the most well documented, minimalistic and well supported open source solver is the MiniSAT. It is widely popular in academia and we would be using the same for our implementation as well. Following is the API and a brief discussion of the operations and methods that MiniSAT supports. DPLL and CDCL described above provide a basic overview of inner workings of MiniSAT. Efficiency tweaks in MiniSAT, however are not of prime importance. The inquisitive reader can examine this solver in greater detail on the official MiniSAT website.

The API of the MiniSAT Sat Solver as mentioned on the official website is as follows. We describe in short the operational utility of the prime methods mentioned in the API in order to give the reader a feel of the implementation details of using MiniSAT via a C/C++ Interface.

- `bool` **`addClause`**`(vec<lit> clause)`
  - From these variables, clauses are built and added using addClause. Trivial conflicts are detected and in case of conflicts like a literal and its negation, the addClause method returns FALSE indicating that the Boolean Formula is unsatisfiable

- `bool` **`simplifyDB`**`()`
  - This method simplifies the Boolean Formula by Unit Clause Propagation[11] Trivial conflicts can also be detected in this stage and this method is particularly called before calling the solve method described below

- `bool` **`solve`**`(vec<lit> assumptions)`
  - This method is called when we wish to solve the Boolean Formula, and is typically passed an empty list of assumptions in the beginning. It returns TRUE if the Boolean Formula is satisfiable and FALSE otherwise. In case of TRUE, the public vector<bool> model contains the satisfying truth assignment.
  - In case the solve function returns satisfiable, we can add additional constraints to the solver by using the add…(…) function and running solve again.
  - Also, in case we wish to pass assumptions to the solve function, we need to pass them as a vector of unit literals and run simplifyDB() mandatorily to detect any trivial conflicts and do unit propagation. Calling simplifyDB() before solve() is mandatory as per the API Assumption here is indicative of a partial assignment

This implementation of MiniSAT uses CNF as the underlying representation for the Boolean Expression or Formulae in question. Also, Improvements over this package have been done in papers proposed after 2003. Also, using the improved versions of the theory, a SAT Solver named MiniSAT+ is also available on the official MiniSAT Website

As far as this project is concerned, the only purpose the above description serves is giving the reader an idea about the methods in the original MiniSAT package. Since we would be using a python binding of the above API, the methods that we use would be a bit different in terms of syntax. However, conceptually, the above methods should give the reader a good idea about the upcoming Python MiniSAT API that we would be using in case of our implementation.

3.2 All-SAT Algorithm Template

3.2.1 Naïve All-SAT Algorithm

Now that we know about finding a single SAT Solution to a SAT Instance, we would describe an algorithm template that describes how to obtain all solutions of a particular SAT Instance. The pseudo code for the algorithm is outlined below

```
Input: CNF formula
Output:  Set of all solutions of CNF
function naïve_ALLSAT (CNF)
    solution_set := empty_set
    while(unsat(CNF))
        solution := solve(CNF)
        solution_set.add(solution)
        blocking_clause = negate(solution)
        CNF := CNF ^ blocking_clause
    end
return solution_set
```

The algorithm is pretty straight forward. We have, as input the CNF formula labelled here as CNF. Our goal is to output all satisfying solutions of the CNF formula. The way we achieve this is using the algorithm described above. While the problem becomes unsat, we solve it to get a solution and it to our solution set. We then produce a blocking clause, which is essentially a negation of the current solution and add this clause to CNF. This is done so that the current solution is not procured again by the solver. This exhaustive search up until the problem CNF becomes unsat leads us to get all satisfying solutions of the problem CNF. Note that CNF here denotes the problem itself.

This in itself is however, not a very elegant solution to the All SAT problem. We will study one important improvement to this algorithm before advancing on to the novel algorithmic technique that we study in the next section.

3.2.2 Greedy Cube Cover

The solution generated by a solver in the algorithm described above involves an assignment for all the variables that make up the Boolean formula. A variable assignment being a solution essentially means that the variable assignment is able to cover the entire set of clauses of the CNF form provided to us. However, it is indeed very possible that only a subset of the variables need to be assigned certain values in order to cover all the clauses of the formula in the CNF form. Such a set of literals is said to 'cover' the Boolean formula. The advantage of such a cover is that it enumerates many solutions at once. The missing variables in the cover can be assigned any value , and hence, if n variables are missing from the cube cover, these n variables can be assigned a combination of 2^n different values and hence the cover enumerates 2^n different solutions. This is particularly useful in the above algorithm for all sat.

Let us say we get one solution from our solver. If we could generate a cover spanning this solution, and add a blocking clause negative this cover instead, we would block all solutions represented by the cover and also add all solutions represented by the cover to the solution set. It is this idea that leads to

modification of the above algorithm to introduce a get_ cover method. The modified algorithm, however, would also need a modified output. We now output set of covers. Note that these covers are all mutually disjoint since we add a blocking clause after computation of each cover. Each cover enumerates a certain number of solutions. Another way to represent the output is represent them as a sum of products, the products being cube covers. Since this formula is '1' even if any one of the cubes is 1, this is an equivalent form of the initial CNF formula. Also, here we have all covers, which essentially means we have all solutions listed. We can also reformulate the problem of finding all solutions of a SAT problem as finding it's DNF equivalent.

The modified algorithm is described below.

```
Input: CNF formula
Output:  Equivalent DNF formula
function cover_ALLSAT (CNF)
    DNF := null
    while(unsat(CNF))
        solution := solve(CNF)
        cover := get_cover(solution,CNF)
        blocking_clause = negate(cover)
        DNF := DNF | cover
        CNF := CNF ^ blocking_clause
    end
return DNF
```

The | symbol in the above algorithm indicates an OR operation, essentially creating the DNF form cover by cover. The blocking clause is obtained as previously by negating the cover. Adding the blocking clause to the CNF ensures that we don't produce the currently produced cover again. We now go on to describe how we actually compute the cover , i.e , the working of the `get_cover(solution,CNF)` method. The explanation for the algorithm as well as it's pseudo code have been provided on the next page.

```
function get_cover(solution,CNF)
    essentials :=  get_essentials(solution,CNF)
    CNF := prune_essential_cover(essentials,CNF)
    literals := greedy_cover(CNF)
return essentials ^ literals
```

The first step in the above algorithm, get_essentials corresponds to getting the essential literals from the literals in the solution. What this means is, the solution might contain certain literals that are the sole literals that make certain clause in the CNF to evaluate to true, and therefore them being assigned the truth value of 1 is absolutely necessary. Such literals are called essential literals and are extracted using get_essentials method.

There is now no point in keeping the clauses covered by these literals, hence we prune the clauses covered by these essential literals using the prune_essentials_cover method shown above. The resulting pruned CNF formula is then assigned to the previous CNF formula.

Lastly, we take up a greedy strategy to computing the least set of literals that completely cover the clause set or updated CNF form. Although this strategy does not guarantee optimum results, it often gives results of the order of O(logn) approximation to optimum. What we do here is greedily select the literal that covers the most clauses, eliminate clauses covered by it, and add it to our literals selection. We do this repeatedly until the entire CNF form is covered.

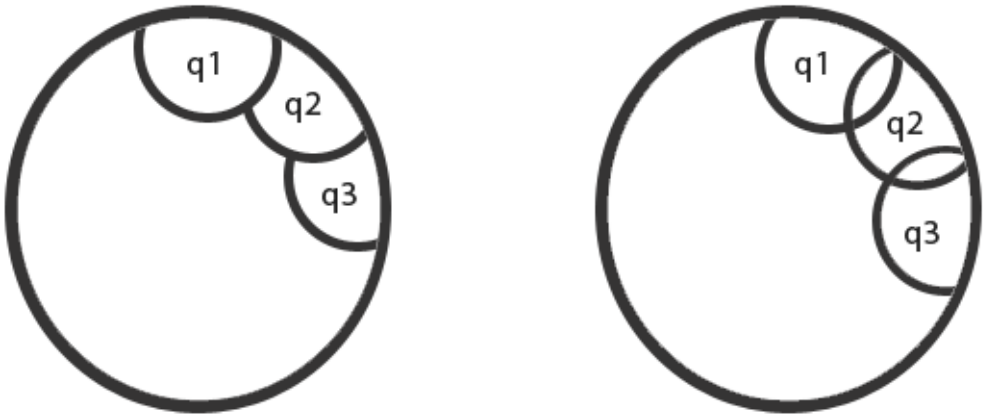
The resulting literals set and essentials set is then unified and a combined cube is then returned as a cube cover to the problem. We now go on to study a very novel technique based on complexity theory that makes a minor modification to the algorithms we've seen so far. This is based on the paper 'All SAT using Minimal Blocking Clauses' and forms the crux of the next section.

3.3 All-SAT using Minimal Blocking Clauses

We now go on to discuss the novel technique for the All SAT Solver that we have implemented and studied. This technique is called All SAT using Minimal Blocking Clauses. We first start with the Intuition for All SAT using Minimal Blocking clauses as described by the paper by Yu, Yenlei et al [6] and later on describe the crucial changes in the pseudo code for the algorithm that we described in the previous section.

3.3.1 Intuition

The intuition for All SAT using Minimal Blocking clauses can be best described using a diagram. Consider Figure 7.



(a) Non Overlapping Cubes
Naive and All Clause

(b) Overlapping Cubes
Non Disjoint Algorithms

Figure 5: All Clause and Non-Disjoint Algorithms

The Algorithms we saw in the previous section, viz. the Naïve Algorithm and the All Clause Algorithm produce, during each iteration, solutions that are mutually exclusive. What we mean by this is that each cube cover produced by

the All Clause Algorithm enumerates mutually disjoint or distinct solution. In the Naïve case, since all solutions are different, they are by definition disjoint. This scenario is diagrammatically represented in Figure 7(a) which shows mutually disjoint cubes produced by the Naïve as we as All Clause Algorithms.

However, from complexity theory, counting the number of satisfying assignments to a Boolean formula is *#P Complete* in time complexity. This can be done in Polynomial time given a set of disjoint cubes in DNF Form, since disjoint cubes ensure that no minterm is counted twice. However, most problems related to computing a DNF cover are of Σ_2^P complexity. This suggests that the All Clause Algorithm is doing more work than necessary. [6]

3.3.2 Modification and Pseudo Code

Using the above insight, one thing we could try doing is letting the cubes overlap. That is, not providing a strict constraint that forces the cubes to be mutually disjoint. This is exactly what was tried in the paper by Yu, Yenlei et al[6]

In order to do this, we need to make one modification in the algorithm described on the previous page. We now maintain a record of the original CNF problem and compute covers of the solutions on this CNF problem's search space rather than the updated CNF. Rest everything remains the same.

```
Input: CNF formula
Output:  Equivalent DNF formula
function cover_ALLSAT (CNF)
    DNF := null
    CNF_Init := CNF
    while(unsat(CNF))
        solution := solve(CNF)
        cover := get_cover(solution,CNF_Init)
        blocking_clause = negate(cover)
        DNF := DNF | cover
        CNF := CNF ^ blocking_clause
    end
return DNF
```

Note that in the algorithm on the last page, we still update the CNF form in order to find newer solutions in the disjoint search space, but we compute covers on the Initial CNF Form called **`CNF_Init`**

We reproduce the `get_cover` method here for brevity.

```
function get_cover(solution,CNF_Init)

   essentials :=  get_essentials(solution,CNF_Init)

   CNF:= prune_essential_cover(essentials,CNF_Init)

   literals := greedy_cover(CNF_Init)

return essentials ^ literals
```

We now go on to explain more about our implementation of All SAT Using minimal blocking clauses in the next chapter - Implementation

Chapter 4

Implementation

Having looked at the intuition behind All SAT Using Minimal Blocking Clauses and the proposed modification to the pseudo code for obtaining all solutions to a SAT Instance, we now look at the foundation for our research, the implementation of ALL SAT using Minimal Blocking Clauses.

4.1 Implementing All-SAT Using Minimal Blocking Clauses

The implementation of the project was done in Python 2. In this section, we first describe the Core Algorithm Templates used in the Implementation, by presenting their pseudo code. Post that we specify important implementation details necessary for the Algorithm Templates. Next, we specify the input format for the SAT Instances that we process and find all solutions for. We finally conclude the section by binding all the pieces together in the 'Putting it all together section'.

4.1.1 Core Algorithm Templates

We need an algorithm template in order to provide a solution for the problem of converting a given SAT Instance in CNF form to a SAT Instance in DNF Form. We have previously described the All Clause Algorithm template and the modified version of it using minimal blocking clauses. Just for the sake of completeness, we would be re-producing the Algorithm Templates here.

Algorithm Template 1: All SAT Using Minimal Blocking Clauses

```
Input: CNF formula
Output:  Equivalent DNF formula
function cover_ALLSAT (CNF)
    DNF := null
    CNF_Init := CNF
    while(unsat(CNF))
        solution := solve(CNF)
        cover := get_cover(solution,CNF_Init)
        blocking_clause = negate(cover)
        DNF := DNF | cover
        CNF := CNF ^ blocking_clause
    end
return DNF
```

Algorithm Template 2: Greedy Cover

```
    Input:  One satisfying assignment(solution), also
called minterm

    Output: Cube Cover for minterm on the provided
search space(CNF_Init)

    function get_cover(solution,CNF_Init)

       essentials :=  get_essentials(solution,CNF_Init)

       CNF:= prune_essential_cover(essentials,CNF_Init)

       literals := greedy_cover(CNF_Init)

    return essentials ^ literals
```

As far as pseudo code is concerned, this is what we need to do. Implementation wise, in Algorithm Template 1, we need to specify how we are going to get a solution to the evolving CNF problem. In Algorithm Template 2 there are important implementation details that we need to specify, like how are we going to get the essential literals? How are we going to get the clauses covered by them? Do we need any additional data structures to represent the information? How do we compute the greedy cover? All of these questions will be answered in the next two parts.

4.1.2 MiniSAT using Python Bindings

In order to implement Algorithm Template 1, we need to specify how we are going to actually solve the evolving SAT Instance, since that is something that is important to the implementation. As the authors suggest[6], we use MiniSAT, a minimalistic and simple to implement SAT Solver written in C/C++. However, since our implementation is inn Python, we need to use a python bound SAT Solver based on the MiniSAT. Fortunately, there is an open source Python binding available called PyMini solvers, cited in reference [13].

The code written as a part of the project includes a README file which links to the Full API for PyMini solvers. We first describe methods relevant to our implementation and then cover a basic usage example for the reader to get acquainted with the basics of the solver.

Initializer methods:

We need to use them first to initialize our Solver instance with a given SAT problem. This is typically done before doing any operation on the CNF problem instance.

`MiniSATSovler()`

This method acts as a constructor for instantiating a MiniSAT based solver instance. This constructs the object to which we can deliver our CNF instance using it's methods and do other SAT operations.

`.new_var()`

This method is used to inform the solver about the number of existing variables in our CNF problem. We need to call this method 'n' number of times in the beginning if there are 'n' variables in our CNF problem. This method is typically called first to initialize the Solver with the particular CNF problem instance.

`.add_clause(<list>)`

We now pass clauses to the solver to initialize it for our particular CNF problem. This list is typically a list of integers denoting our literals. Negative literals are integers with '-' before them.

Other methods:

Now we describe the methods used to solve the SAT Instance that has been initialized by using the Initializer methods and getting a particular solution.

`.solve()`

This method solves the CNF problem instance associated with the solver. It returns True if the CNF problem is satisfiable and false otherwise

`.get_model()`

This method returns one satisfying assignment in case the CNF problem instance currently associated with SAT Solver is satisfiable and solve has been called on it. Note that we need to call solve before we call this method. Also usually, it is good practice to cast the return type of this method to a list as we then get a list of '0' or '1' assignment to the variables of the CNF problems in order.

We now describe a simple example using the above methods. This example is the same as that in the PyMini solvers' README file. We first cover the pseudo code and then go on to explain the example.

```
from mini solvers import MinisatSolver

// initializing a SAT Solver Instance
S = MinisatSolver()
```

```
// Creating new variables as per given problem
// Let's consider we have 4 variables here
for i in range(4):
    S.new_var()

// Adding clauses involving variables
for clause in [1], [-2], [-1, 2, -3], [3, 4]:
    S.add_clause(clause)

// Solving the instance
S.solve() // Returns True

// Method that returns the assignment to variables
list(S.get_model()) // Returns [1,0,0,1]

// Adding a clause to make it unsatisfiable
S.add_clause([-1, 2, 3, -4])

// Returns False since instance now unsatisfiable
S.solve()
```

The example is pretty simple. We initialize a sat instance, create 4 variables and add 4 clauses. We then solve the instance and since it returns true, we know that the solver has found a satisfying instance. We then go on to add a blocking clause, which if you observe is a negation of the variable assignment we received previously from `list(get_model)`. This makes the CNF problem unsolvable, which is why the `solve()` method returns False on the last line.

4.1.3 Additional Data Structures

In order to implement the `get_cover` method described in Algorithm template 2, we need to specify a few more details. We need a data structure to efficiently compute the essential literals, clauses covered by them and implement the greedy strategy. We use a matrix which we call the membership matrix to serve our purpose.

The membership matrix is common to the unate covering problem and is also the one that is used by the authors in reference [6]. In order to construct this matrix, given a particular minterm as the solution and a search space, what we do is the following. Let us say that the number of clauses in given CNF instance is m. Let's also consider the number of literals in the minterm as n. Our job now is to construct a mxn matrix each term of which is either '0' or '1'. Each row of this matrix represents a clause in the provided CNF. For each row, every column entry is 1 if that particular literal of the minterm occurs in the clause and 0 otherwise. An example of a CNF form, a minterm is shown in the table below.

CNF Form: `(1).(-2).(-1 2 -3).(3 4)`

Solution or Minterm: `(1 -2 -3 4)`

Membership Matrix for Minterm:

	Literal 1	Literal 2	Literal 3	Literal 4
Clause 1	1	0	0	0
Clause 2	0	1	0	0
Clause 3	0	0	1	0
Clause 4	0	0	0	1

Table 1: Membership or UCP Matrix Example

In the above example we check each literals membership in each clause and construct the matrix. Clauses and solution literals from which this matrix was obtained have also been specified at the top.

4.1.4 Input Format

We have already specified the format of the input formula to be CNF which is basically a conjunction of clauses which are disjunctions. However, we need to specify one more detail about the input format. The standard format for Benchmarks used by us as well as the SAT community as a whole is DIMACS. We describe the specifications of the DIMACS format for Input CNF benchmarks on the next page.

Quoting the DIMACS format directly from the website, -

"An input file starts with comments (each line starts with c). The number of variables and the number of clauses is defined by the line p cnf variables clauses

Each of the next lines specifies a clause: a positive literal is denoted by the corresponding number, and a negative literal is denoted by the corresponding negative number. The last number in a line should be zero. For example,

c A sample .cnf file.
p cnf 3 2
1 -3 0
2 3 -1 0 "

We use a function `parse_SAT` in our implementation to parse a given CNF SAT Instance in DIMACS format and instantiate a full SAT Solver instance by adding variables and clauses to it.

4.1.5 Putting it all together

Having specified all the pieces together, we now specify the organizational flow of the implementation of the All SAT Using Minimal Blocking Clauses in Python.

The process is diagrammatically represented in Figure 9. We first take in the CNF problem in DIMACS format. This problem goes through the `parse_SAT` function in the implementation and returns a SAT Instance as well as clause list of the original CNF SAT Instance.

Post this process, we implement Algorithm Template 1 which iteratively computes overlapping cube covers until the problem becomes unsatisfiable. It makes use of minisat to compute solutions. The covers for these solutions are computed using Algorithm Template 2, which uses the membership matrix data structure to to operate on the data provided to it.

The final result is then the desired Boolean Formula in DNF format

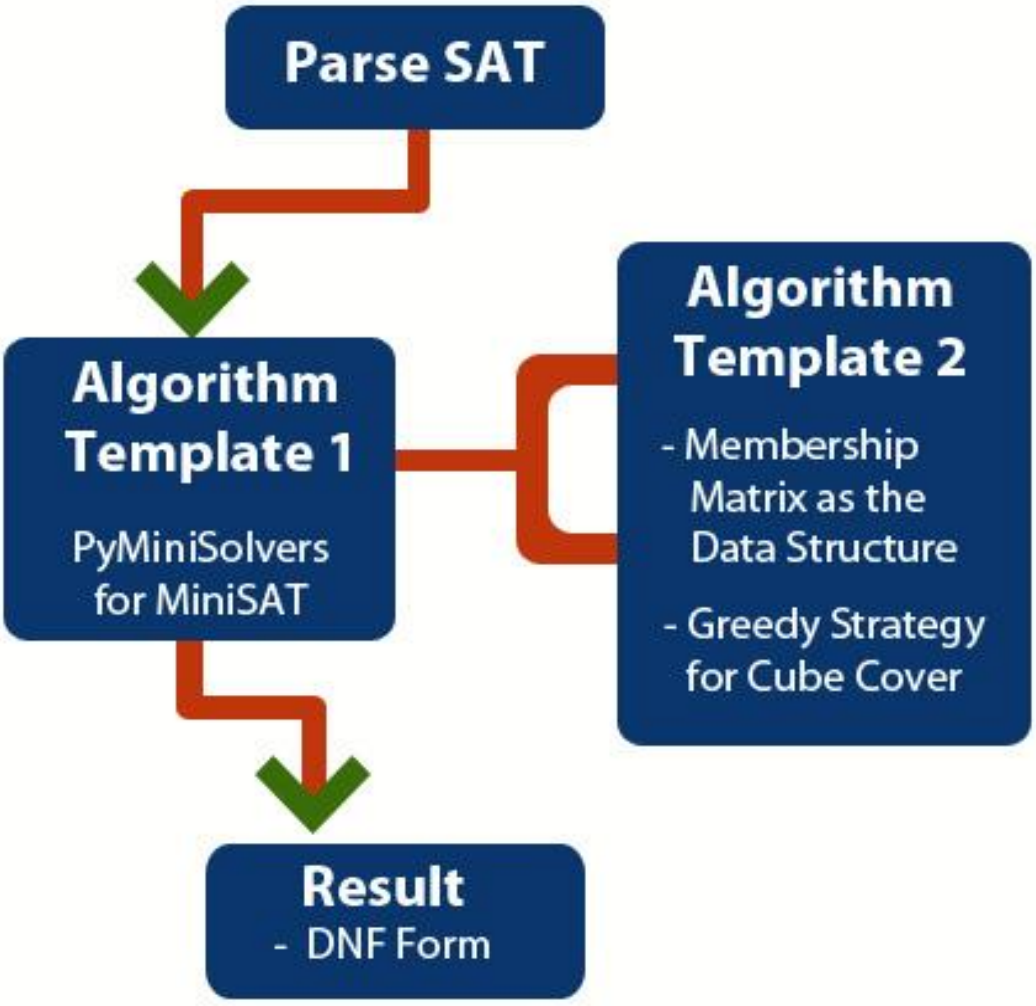


Figure 6: Implementation Flow

Chapter 5

Results and Discussion

Having implemented the All SAT Solver using Minimal Blocking clauses in the last chapter, we now run some simulations on the solver. We first specify experimental setup and benchmarks used. Then we move on to present profiling results for the solver. Based on these results, we draw certain conclusions and suggest some additional improvements that could help improve the solver performance. We propose using predictive classification techniques that can be used in case of Hard SAT & UNSAT problems that are tough even for MiniSAT. We test these techniques and present some interesting insights. Finally we conclude with application specific strategies that build up on these techniques and boost their performance.

5.1 Experimental Setup

We now describe the experimental setup which includes the machine specs as well the descriptions about the benchmarks that we used for our experiments.

5.1.1 Machine Specifications

We used an Intel(R) Core(TM) i5 CPU with an operational clock of 2.67 Ghz and memory cap of 4GB. We kept an execution time out cap of 1.5 hours for obtaining All SAT Solutions for the instances. We now proceed on to describing the benchmarks that were used for obtaining the profiling results on the next page.

5.1.2 Benchmarks

We used standard benchmarks for our evaluation. Our benchmarks were a subset of the benchmarks listed on the satlib.org official website. We also used benchmarks of SAT Competitions from 2007 and prior to 2007 for evaluation of other techniques which we implemented in order to try and improve the performance of the solver.

Benchmarks from these sources are classified into 3 prime categories, which we think are important to specify from point of view of further discussion.
The three categories are: -

5.1.2.1 Random

Random problems:

These are randomly generated instances of sat problems. They have a subset called Random 3SAT which strictly have only 3 literals in all clauses. We have used Random 3SAT ones for evaluation.

5.1.2.2 Crafted

Crafted problems:

These are problems that are crafted specifically to make it hard for the solver to solve them. These are also called handmade sat problems

4.2.2.3 Industrial

5.1.2.3 Industrial

Industrial problems:

These are SAT problems that are related to industrial grade SAT problems occurring in domains like VLSI Verification amongst others. This concludes the discussion of our Experimental setup.

5.2 Results of Implementation

Having described the Experimental setup and benchmarks, we now go on to describe the results obtained as a result of the simulations carried out on a subset of the benchmarks described above.

5.2.1 Identified Categories of Problems – I & II

We tried running the All SAT solver on 3 sets of Industrial Problems, 2 sets of crafted problems and 4 sets of Random problems. The problems were selected such that they were large enough in size to be of significance.

On running the simulations, we observed that there were broadly two categories into which the problems from all 3 categories could be classified. These categories were named Category I and Category II. We provide detailed description about them below.

Category I: These were the problems the solver was able to solve without a memory insufficiency or timeout. We include a timing analysis of different functions for such problems below. For such problems, the program was correctly able to generate an equivalent DNF Cover of the given CNF Form and thus enumerate all satisfying solutions.

Category II: These were the problems that the solver did not generate solutions for. There were two prime reasons identified for this failure. One of which was timeout and the other was memory insufficiency.

We include a timing analysis of both kinds of problems, in our next section – Timing Analysis for Problems.

5.2.2 Timing Analysis for Problems

5.2.2.1 Category I

These problems, as described above, were the problems for which the solver could successfully obtain a DNF cover. We analyzed the timing split for various functions in the implementation for such problems. We present the relative timing split in percentages across major functions across all three categories on the next page. We also present a bar graph for a good visual comparison of the results in Figure 9

	Membership Matrix	Essential Literals	Clauses Covered
Random	50.41%	9.12%	36.02%
Crafted	47.60%	7.87%	32.71%
Industrial	48.68%	7.80%	41.78%

Table 2A: Timing Analysis for Category I [In Percentage of Total Time]

	Greedy Cover	Parsing Time	Other Functions
Random	0.7%	0.0001%	3.74%
Crafted	0.7%	0.0001%	11.12%
Industrial	0.0002%	0.0001%	1.73%

Table 2B: Timing Analysis for Category I [In Percentage of Total Time]

As we can observe from the above results that in case of problems that are solvable by the solver, i.e Category I, majority of the time for all three categories is split between the membership matrix construction, essential literal determination and clause cover determination(of the essential literals). This suggests that in case of problems where the solver successfully determines all satisfying solutions to the problem, these three functions form the bottlenecks.

We can also observe that the parsing time is negligible in all three cases. We can also observe that greedy cover time reduces dramatically in case of Industrial problems. We also see that determining clauses covered by the essential literals takes more time in case of the Industrial problems. Both of these results are indicative of the fact that in case of Industrial problems, essential literals cover more clauses than other two categories.

Figure 9 represents a bar graph of timing split across categories for better visualization. We only represent the major bottlenecks in the bar graph, wrapping greed cover, parsing and other functions into just – 'other functions' for brevity and better understanding.

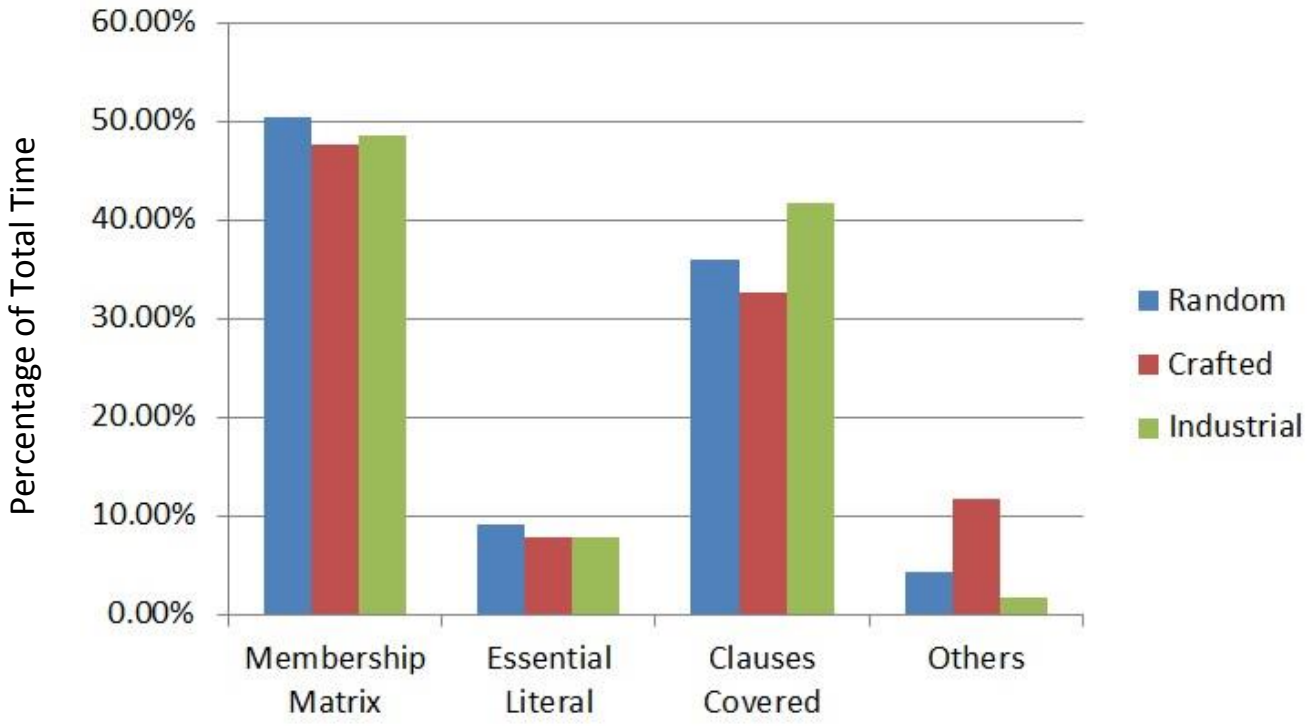


Figure 6: Timing Analysis for Category I Problems

5.2.2.2 Category II

For these categories of problems, the major bottleneck was the solve function. This typically occurred in case of large UNSAT or SAT problems. For such problems, in the UNSAT case, it is typically hard to conclude that the problem is UNSAT unless the entire exponential search space is explored. Such problems that are hard to prove UNSAT and most solvers time out on such instances. For the SAT case, hardness of a problem generally means that it is hard for the solver to generate a satisfying solution to the problem In general. As it takes a long time to generate even one single satisfying solution, the time taken to generate all satisfying solutions, even using the novel technique described earlier, is gigantic.

From a In-Practice point of view, the second more frequently occurring problem in such problems is memory insufficiency due to which the solver cannot

solve the problem. In our simulations, it was also observed that memory insufficiency typically occurred in case of problems that took a long time to solve.

For such problems, the major bottleneck observed was the solve() function of the MiniSAT Api, which is not surprising since we know that for such problems, it is difficult to obtain a satisfying solution in the SAT case and difficult to conclude UNSAT in the UNSAT case.

From an applicability point of view, the issue of large SAT and UNSAT problems being discarded by the standard solvers due to their hardness is a pretty challenging problem to tackle. But since this problem is NP-Hard, there is no known algorithm to solve this problem in polynomial time. However, based on prior research, we figured that a predictive technique, based on heuristics that estimate the hardness of a problem, might be the right way to go. We describe this in more detail in the section 5.2.3.1 – Desired Characteristics of Prediction.

Such a technique would not only benefit our ALLSAT Solver but also be applicable to a variety of other domains. A concrete example would be a solver that needs to know the probability of a SAT Instance being satisfiable after a particular variable assignment branch has been taken.

5.3 Predictive Techniques for Category II

In this section, we first describe the characteristics of the predictive techniques that make them useful for our application.

We then build up predictive techniques that predict whether a problem is SAT or UNSAT. We test their accuracies on standard benchmarks. We refer to the work done in Reference [14] and build application specific classifiers of our own.

5.3.1 Desired Characteristics of Prediction

As previously discussed, in case of Category II problems, the All SAT Solver terminated it's operation either due to (a) timeout or (b) memory insufficiency without giving a conclusive result about the problem. In order to reduce the magnitude of uncertainty regarding such problems, we figured that it would be

better if we could predict, with some confidence if a problem was satisfiable or not. If UNSAT, the solver would skip such a problem and if SAT, would increase the stack memory and try to solve the problem.

For such an application, it is necessary that the probability of a problem being SAT and classified as UNSAT be as less as possible, since this would mean we are totally misclassifying that problem all together. On the other hand, if we happen to misclassify an UNSAT problem as SAT, the solver would try to solve it , and in the end after exploring the entire search space, would conclude that the problem is unsatisfiable. In this manner the error of UNSAT as SAT is caught whereas the error of SAT being classified as UNSAT is not. It is therefore, in our interest, to make the probability of SAT being classified as UNSAT as less as possible. We now start to build classifiers of our own to predict whether or not a particular instance is satisfiable. Before doing that, we go on to describe the feature set that we extract from a particular problem that will later be useful for classification of the problem. We do this in brief. The feature set chosen was the same as that of SATZilla[15] and the inquisitive reader is advised to refer to the paper. We also describe the feature extraction time required to extract these features for various categories of problems.

5.3.2 Choosing the Feature Set

We briefly discuss the features chosen for the classification. These features are the same as that of the features chosen by the SATZilla[15] solver. The features can roughly be categorized into 9 categories. The first category of features provides an estimate of the problem size. It includes variables, clauses, their ratio and cubes and squares of the ratio as well. The next 3 groups consist of statistics involving graphical representations of the SAT instance. A SAT instance can be represented as a variable clause bipartite graph, variable graph or clause graph. Statistics for each of these graphs, such as diameter, clustering coefficient etc. are calculated and set as features in this group. The fifth category is of balance features. These features provide information about the number of unary, binary and ternary clauses in the SAT Instance, balance of positive to negative literals

etc. The sixth group provides information about proximity to Horn Formula as this is an important aspect in SAT Solving. The seventh set provides information by solving a linear programming relaxation of an integer program representing the current SA T instance. This is important since it provides information about unit propagation depths until contradictions are found after running the solver on the relaxation for a stipulated amount of time. Lastly, the last two groups provide information about local search parameters. These are averaged across many runs and the algorithms used are stochastic local search algorithms called GSAT and SAPS.

Although the features are an important part of the problem of classification, our focus is more on utilizing these pre-calculated features for prediction rather than researching about more features that might be useful. This is taking cue from Reference [14] in which they have used these features for building classifiers.

Although, we do not go into the details about the feature set, we do mention about the time it takes to compute these features as this is important from point of view of actually knowing how much time is required for our improvement to function inside the All SAT Solver or any other application. We now outline the feature computation time for our benchmarks in Table 3.

	Average	**Min**	**Max**
Random	7.99s	0.44s	14.34s
Crafted	9.86s	0.68s	140.09s
Industrial	145.94s	0.25s	6217.97s

Table 3: Feature Computation Time

As we can see from the above table the average feature computation time for all three categories is not over 146s. This time is negligible compared to the days for which an instance can run on the solver without any conclusive result. Having detailed the feature computation times for our benchmarks, we now go on to describe the classifiers that we built, and their performance.

5.3.3 Benchmarks

We use the same 3 categories of benchmarks as before in addition to a fourth benchmark. We train and test our prediction models on Random, Crafted and Industrial problems along with a fourth category called Mixed. This category has a combined set of benchmarks for all three categories, Mixed together. This is done from point of view of training the model to accommodate a scenario where we don't necessarily know the kind of SAT instance we are dealing with.

We used the complete SAT'07 and before benchmark suite for our training and testing our classification algorithms.

5.3.4 Classifiers

We now describe the various classifier algorithms we used in order to classify the test set. In all classifiers we used 10-fold cross validation as a technique for estimation of the error. We describe Neural Net Configuration in detail since we did some experiments to obtain an optimum architecture for it to be able to classify the benchmarks in the best possible manner. Post that we describe the other classifiers that we have used very briefly. In the last part of this section, we give combined results for all 5 classifiers.

5.3.4.1 Neural Net Configuration

We now look at the first classifier we tried, a Neural Network. We thought that it would be a good start with a neural network as we can vary the Neural Network Architecture to obtain varying accuracies. The strategy was to obtain the optimum architecture from the point of view of SAT Classification. We now describe the process and the outcome.

We used a fully connected 3 layered Neural Network as shown in Figure 10. The number of neurons in the middle or hidden layer was varied from 1 to 45 in intervals of 5. Each of these Neural Network models was trained and tested on the feature data set of SAT benchmarks of SAT Competition 07 and earlier. We did a different analysis for Random, Crafted, Industrial and Mixed problem data sets.

We now discuss the results obtained from our simulations.

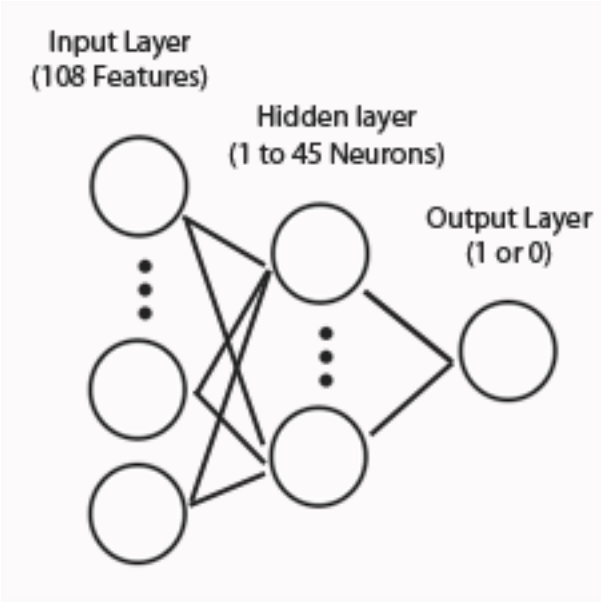


Figure 8: Neural Network Architecture

As shown in Table 4, as we swept the neurons from 1 to 45 in intervals of 5, the observed error rate roughly went down until 35. At 35, we see a dip in the error rate, which rises again beyond 35. We figured that having 35 neurons in the hidden layer is best since the error observed by choosing this architecture was least. This may be attributed to the bias variance tradeoff in machine learning which has the sweet spot at 35. We outline the results graphically in in Figure 9.

	1	5	10	15	20
Random	11.3646	11.4104	11.959	11.197	10.7568
Crafted	21.646	18.206	17.454	19.863	20.128
Industrial	17.442	16.772	14.896	15.586	13.844
Mixed	24.849	22.193	18.702	19.051	18.567

Table 4A: Error in % as neurons are varied

	25	30	35	40	45
Random	13.3108	13.4244	10.203	9.248	15.5598
Crafted	21.248	15.521	15.449	14.715	16.836
Industrial	14.058	14.314	12.795	15.34	15.676
Mixed	18.878	18.117	17.718	18.804	16.707

Table 4B: Error in % as neurons are varied

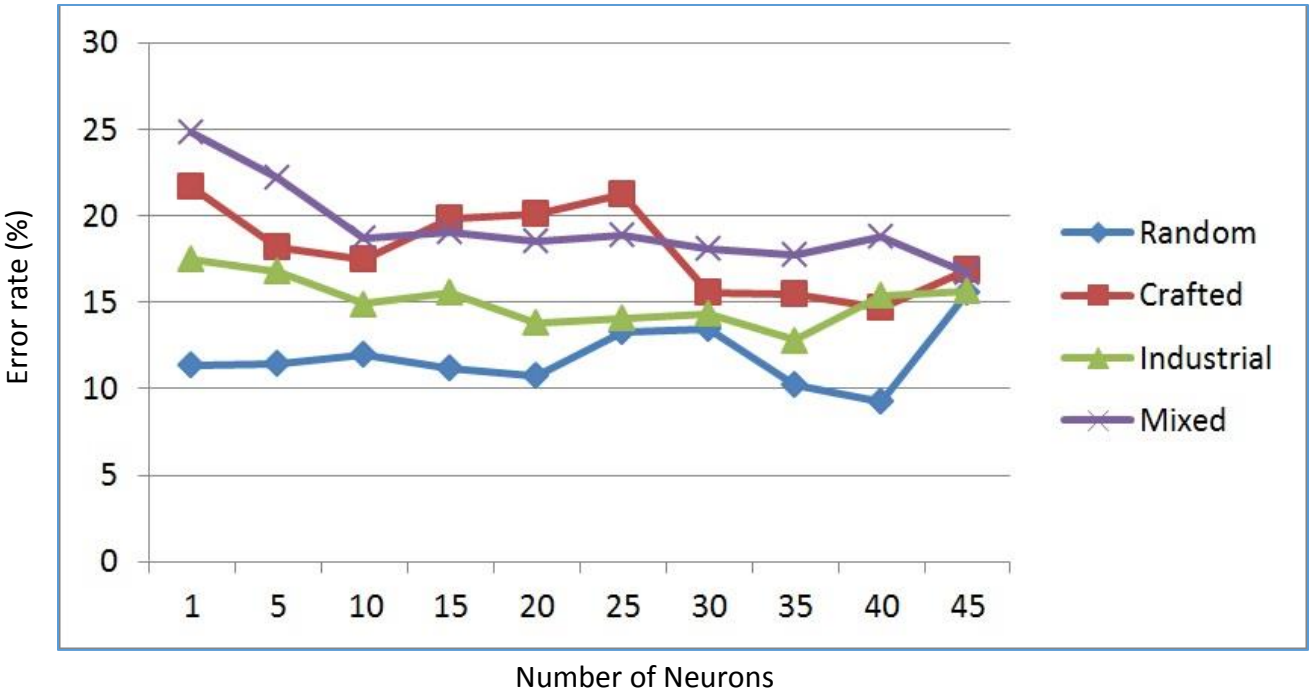


Figure 9: Error rate versus Neurons for all 4 categories

5.3.4.2 Other Classifiers

We also tried 4 other classifiers in order to get a better picture of the classification task at hand. We did not have to fine tune these classifiers as API's for them were readily available. These classifiers were Classification Tree, Naïve Bayes, Discriminant Analysis, and 1 Nearest Neighbor. As specified before, we followed the 10-fold cross validation technique to estimate the error. We encourage the inquisitive reader to go through these techniques in detail in references [8], [9], [10], [16] and [17].

5.3.4.3 Results

We now outline the classification accuracy results obtained using these classifiers in Table 5. As we can observe, Naïve Bayes gives the worst results. Classification Tree algorithm turned out to be the best having the best accuracies across all 4 categories, followed by a near second Neural Network. We obtained

accuracy in excess of 87% in case of a mixed set of data and as high as approx. 90% in the Random category.

Although these accuracies are good, as we previously specified, our application has to have pretty low error rate for SAT problems being misclassified. Some error rate for UNSAT problems being misclassified is permissible.

	Neural Network	Discriminant Analysis	1 Nearest Neighbor	Classification Tree	Naïve Bayes
Random	**89.80%**	89.77%	87.82%	**89.80%**	69.88%
Crafted	84.55%	80.15%	81.77%	**87.13%**	64.22%
Industrial	87.21%	87.91%	88.41%	**88.84%**	74.21%
Mixed	85.40%	78.47%	85.85%	**87.48%**	55.05%

Table 5: Classification Accuracies for various classifiers

In order to make certain rectifications that ascertain this, we need to observe the SAT correctly classified as SAT accuracies for the benchmarks that we just tested on. We did just that, and outline the results for SAT as SAT, as well as UNSAT as UNSAT in Table 6 and Table 7 respectively.

	Neural Network	Discriminant Analysis	1 Nearest Neighbor	Classification Tree	Naïve Bayes
Random	**95.8%**	95.26%	91.27%	93.03%	63.36%
Crafted	86.9%	72.65%	72.65%	83.37%	**95.63%**
Industrial	87.2%	85.66%	87.32%	**87.87%**	56.25%
Mixed	87.9%	85.71%	87.11%	**88.52%**	83.79%

Table 6: SAT correctly classified as SAT

The observed results from Table 6 indicate that we observe good accuracies for our application, with Classification Tree again yielding highest accuracies in case of mixed SAT data correctly classified as SAT. We also observe that Neural Networks give highest accuracy for Random SAT data correctly classified as SAT. In case of Crafted SAT data correctly classified as SAT, we see that Naïve Bayes is

the best. And lastly, in case of Industrial data, Classification Tree again does the best job of correctly classifying SAT instances as SAT.

	Neural Network	Discriminant Analysis	1 Nearest Neighbor	Classification Tree	Naïve Bayes
Random	74.7%	74.26%	79.08%	81.74%	**88.38%**
Crafted	**89.6%**	84.94%	77.59%	89.54%	45.22%
Industrial	**93.4%**	89.87%	89.36%	89.68%	89.52%
Mixed	82.2%	69.49%	84.12%	**86.06%**	45.95%

Table 7: UNSAT correctly classified as UNSAT

In case of UNSAT data correctly classified as UNSAT, we again see that Classification Tree dominates in case of Mixed data, whereas in case of Crafted and Industrial data, Neural Networks is the preferred Classifier. If the data happens to be Random, then Naïve Bayes does a good job of classifying the UNSAT data correctly.

5.3.5 Majority Rule

As we previously mentioned, we are looking at maximizing the SAT correctly classified as SAT accuracies. In order to do this, we decided that we could do something that makes it tougher for the classifiers to wrongly predict SAT problems. We explain our idea in the next section.

5.3.5.1 Idea

We figured that all 4 classifiers together wrongly predicting the SAT instance as SAT is highly improbable. That is, if all 4 classifiers say that the instance is SAT, only then classify it as SAT else, classify it as UNSAT. To explain this in a tabular way, refer to Table 8. Here, we give output from all 4 classifiers and corresponding output from our Majority Rule. We can clearly see that we predict UNSAT, even if one of the classifiers says it's UNSAT. So this is going to bring down our UNSAT correctly classified as UNSAT accuracy.

Classifier 1	Classifier 2	Classifier 3	Classifier 4	Majority Rule Output
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	0
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	0
1	0	0	0	0
1	0	0	1	0
1	0	1	0	0
1	0	1	1	0
1	1	0	0	0
1	1	0	1	0
1	1	1	0	0
1	1	1	1	1

Table 8: Prediction Table - Majority Rule for SAT correctly classified as SAT

We tried the above strategy on our benchmarks and obtained high accuracies for SAT correctly classified as SAT. We outline the results in the next section.

5.3.5.2 Results

We decided to choose Neural Network, Classification Tree, Discriminant Analysis and 1 Nearest Neighbor as 4 classifiers. We did this since we thought including Naïve Bayes would only pollute the results, since it turned out to be a not-so-good classifier. We now outline the results obtained by applying our Majority Rule idea to the benchmarks. Table 9 represents the results. We see that applying this rule. We followed the same procedure of 10-fold cross validation as used on the individual classifiers earlier.

We can see in Table 9 that we obtained very high accuracy (as high as 98.23% for Mixed data) for SAT correctly classified as SAT problems by applying

the Majority Rule. We also observe that the UNSAT as UNSAT accuracy goes down significantly. However, in case of Industrial benchmarks, this accuracy was observed to go down relatively less dropping down to 80%, which is acceptable in our application.

	Overall	SAT as SAT	UNSAT as UNSAT
Random	88.04%	**98.95%**	57.15%
Crafted	78.45%	**94.53%**	69.21%
Industrial	87.74%	**96.88%**	80.03%
Mixed	81.44%	**98.23%**	59.23%

Table 9: Accuracy split for Majority Rule – SAT as SAT

We also tried a variation of the Majority Rule for UNSAT correctly classified as UNSAT and obtained the results outlined in Table 10. We just made a little rectification in the idea. We , in this case, classify as UNSAT only if all 4 classifiers say UNSAT , else we predict SAT.

	Overall	SAT as SAT	UNSAT as UNSAT
Random	87.64%	85.64%	**93.19%**
Crafted	78.53%	49.36%	**97.77%**
Industrial	84.95%	71.60%	**96.88%**
Mixed	82.44%	74.33%	**95.05%**

Table 10: Accuracy split for Majority Rule – UNSAT as UNSAT

We observe that the SAT as SAT accuracy is 74.33% for Mixed problems in this case. This might be acceptable for a trade off in some applications.

For future work on this, we suggest modifying the Majority Rule to predict SAT if all 4 classifiers as SAT, predicting UNSAT if all 4 classifiers predict UNSAT,

and predicting Unknown in all other cases. This would ensure high accuracies for both SAT as well as UNSAT problems, although we would not classify some of the problems. The prediction table for such a case is shown in Table 11.

Classifier 1	**Classifier 2**	**Classifier 3**	**Classifier 4**	**Majority Rule Output**
0	0	0	0	0
0	0	0	1	U
0	0	1	0	U
0	0	1	1	U
0	1	0	0	U
0	1	0	1	U
0	1	1	0	U
0	1	1	1	U
1	0	0	0	U
1	0	0	1	U
1	0	1	0	U
1	0	1	1	U
1	1	0	0	U
1	1	0	1	U
1	1	1	0	U
1	1	1	1	1

Table 11: Prediction Table for Modified Majority Rule

Chapter 6

Conclusions and Future Work

6.1 Conclusions

In this chapter we outline the conclusions obtained as a result of our experiments and provide a guideline for solid future work in this direction.

As we discussed, we obtained two categories of problems after running simulations. Category I was of the problems that the solver could produce All satisfying solutions for in limited time without running out of memory. Category II consisted of hard problems that either the standard solver timed out on, or ran out of memory. We outline our conclusions with regards to both kinds of problems.

6.1.1 Category I – Identified Bottlenecks

Having simulated across standard benchmarks like that of Reference [6], we come to the following conclusions about Timing Analysis for Category I problems.

Our implementation of All SAT using Minimal Blocking Clauses, to the best of our knowledge, has the following bottlenecks –

- Membership Matrix computing function
- Essential Literal Getting Function
- Clauses Covered function that computes clauses covered by these essential literals

We also observed that for Industrial benchmarks, the time taken for greedy cover was significantly less that the Random and Crafted benchmarks. The graph

representing the Timing Split is reproduced here again, for the sake of brevity.

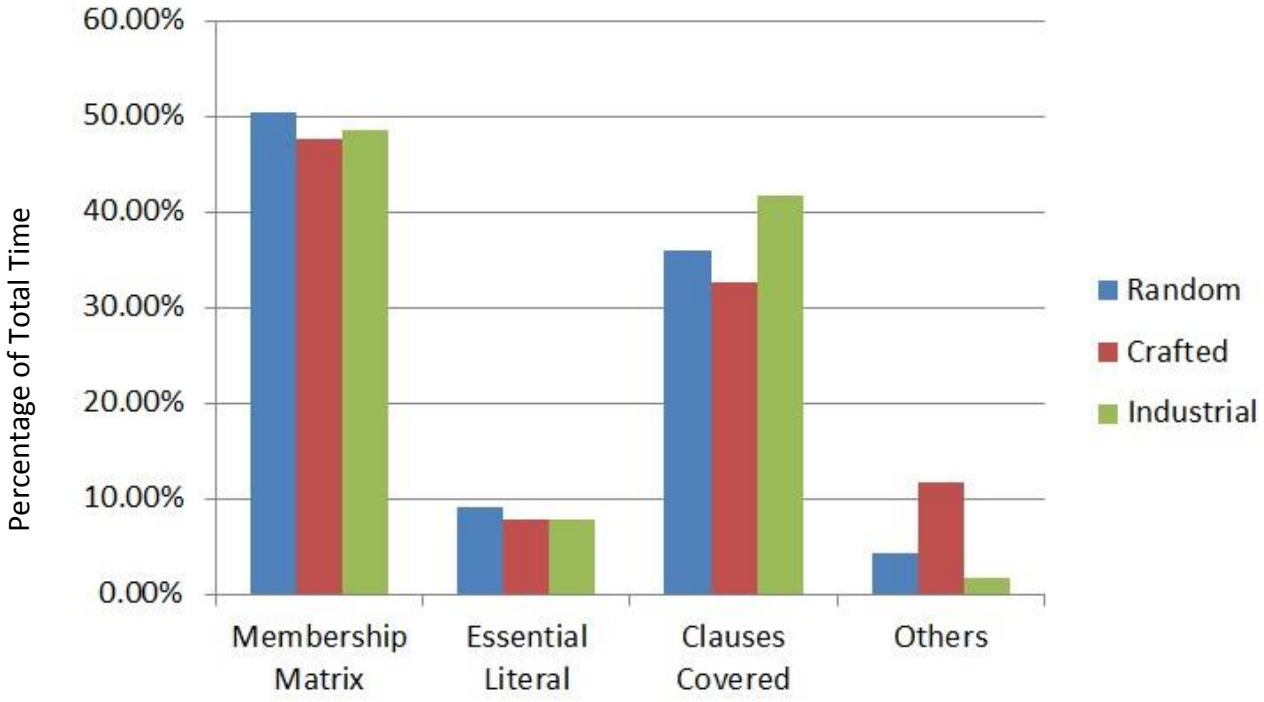


Figure 10: Timing Analysis for Category I problems

6.1.1 Category II – Prediction and Majority Rule

We also saw that utilizing prediction in case of Category II problems, where standard solvers fail, can be a good strategy with us obtaining results as high as 88.52%. We make the following conclusions about prediction as well as Majority Rule.

- In case of Prediction, Classification Tree turned out to be the best classifier giving accuracy as high as 88.52% in case of Mixed Benchmarks.
- Accuracies as high as 98.23% for SAT specific applications can be achieved using the Majority Rule for SAT
- Accuracies as high as 95.05% for UNSAT specific applications can be achieved using Majority Rule for UNSAT

6.2 Future Work

We now outline some solid guidelines for future work in this area. We have primarily identified three prime verticals upon which someone who wishes to continue this work further can build upon. We outline them below.

6.2.1 Modified Majority Rule

As specified previously, the Majority Rule can be modified to obtain high accuracy for SAT as SAT as well as UNSAT as UNSAT simultaneously. This can simply be done by modifying the prediction table described earlier in the last chapter to the form of Table 10. Having done that, some problems will be left un classified, however, accuracy on problems that are classified in either categories would be much higher. We re-produce Table 10 here for brevity.

Classifier 1	Classifier 2	Classifier 3	Classifier 4	Majority Rule Output
0	0	0	0	0
0	0	0	1	U
0	0	1	0	U
0	0	1	1	U
0	1	0	0	U
0	1	0	1	U
0	1	1	0	U
0	1	1	1	U
1	0	0	0	U
1	0	0	1	U
1	0	1	0	U
1	0	1	1	U
1	1	0	0	U
1	1	0	1	U
1	1	1	0	U
1	1	1	1	1

Table 12: Prediction Table for Modified Majority Rule

6.2.2 Prediction as a Heuristic for SAT Solving

The classifiers and the Majority Rule that we have built here have applications well beyond just SAT prediction in case of All SAT solvers. From Reference [14] as well as insight, we can use these predictors to determine the probability of a problem being SAT, when a particular variable assignment is done by a SAT solver such as MiniSAT. We can do the branching that a SAT solver does based on the probability of the resulting problem being satisfiable. This can help us get to the satisfying solution much faster.

6.2.3 Parallelizing Bottlenecks

Lastly, bottlenecks identified for Category I problems can also be parallelized although due to large number of function calls, the effect that this parallelization will have remains something to be experimented with.

Bibliography

A lot of prior work was particularly useful in conducting research on this topic. We would like to acknowledge and cite all the references which have been used in the development of this dissertation.

[1] Andersen, Henrik Reif. "An introduction to binary decision diagrams." *Lecture notes, available online, IT University of Copenhagen* (1997).

[2] Brace, Karl S., Richard L. Rudell, and Randal E. Bryant. "Efficient implementation of a BDD package." *Proceedings of the 27th ACM/IEEE design automation conference*. ACM, 1991.

[3] Somenzi, Fabio. "CUDD: CU decision diagram package release 2.3.0."*University of Colorado at Boulder* (1998).

[4] Minato, Shin-ichi, and Shinya Ishihara. "Streaming BDD manipulation for large-scale combinatorial problems." *Proceedings of the conference on Design, automation and test in Europe*. IEEE Press, 2001.

[5] Sorensson, Niklas, and Niklas Een. "Minisat v1. 13-a sat solver with conflict-clause minimization." *SAT* 2005 (2005): 53.

[6] Yu, Yinlei, et al. "All-SAT Using Minimal Blocking Clauses." *VLSI Design and 2014 13th International Conference on Embedded Systems, 2014 27th International Conference on*. IEEE, 2014.

[7] Xu, Lin, et al. "SATzilla-07: the design and analysis of an algorithm portfolio for SAT." *Principles and Practice of Constraint Programming–CP 2007*. Springer Berlin Heidelberg, 2007. 712-727.

[8] Leung, K. Ming. "Naive bayesian classifier." *Polytechnic University Department of Computer Science/Finance and Risk Engineering* (2007).

[9] Fraley, Chris, and Adrian E. Raftery. "Model-based clustering, discriminant analysis, and density estimation." *Journal of the American statistical Association* 97.458 (2002): 611-631.

[10] Specht, Donald F. "Probabilistic neural networks for classification, mapping, or associative memory." *Neural Networks, 1988., IEEE International Conference on*. IEEE, 1988.

[11]Sieling, Detlef, and Ingo Wegener. "Reduction of OBDDs in linear time."*Information Processing Letters* 48.3 (1993): 139-144.

[12] Zhang, Lintao, et al. "Efficient conflict driven learning in a boolean satisfiability solver." *Proceedings of the 2001 IEEE/ACM international conference on Computer-aided design*. IEEE Press, 2001.

[13] https://github.com/liffiton/PyMini solvers

[14] Devlin, David, and Barry O'Sullivan. "Satisfiability as a classification problem."*Proc. of the 19th Irish Conf. on Artificial Intelligence and Cognitive Science*. 2008.

[15] Xu, Lin, et al. "SATzilla: portfolio-based algorithm selection for SAT." *Journal of Artificial Intelligence Research* (2008): 565-606.

[16] Wong, M. Anthony, and Tom Lane. "A kth nearest neighbour clustering procedure." *Computer Science and Statistics: Proceedings of the 13th Symposium on the Interface*. Springer US, 1981.

[17] Hansen, Matthew, R. Dubayah, and R. DeFries. "Classification trees: an alternative to traditional land cover classifiers." *International journal of remote sensing* 17.5 (1996): 1075-1081.