ECE595Z COURSE PROJECT

BOOLEAN SATISFIABILITY SOLVER

PROJECT REPORT

RAUNAQ NANDY MAJUMDAR *ROHAN KUMAR MANNA*

Introduction

The aim of a SAT solver is to efficiently conclude whether a given Boolean CNF (Conjunctive Normal Form) formula in DIMACS format can be solved or not and if it can be solved, the SAT solver should state the values assigned to each of its variables. The term "efficiently" in the above sentence means that the SAT solver should infer as quickly as possible while utilizing less memory. For this project we utilized the concept of DPLL (Davis-Putnam-Logemann-Loveland) algorithm along with one additional heuristic to compare the performance and benchmark the program based on a database of CNF files. The report mainly consists of six parts namely motivation, reason for choosing the data structure, the algorithm, results obtained, analysis of the results, and summary with conclusion.

Motivation:

SAT is established as the first NP-Complete problem and has proved to be an indispensable component of many formal verification and more recently program analysis applications as well. The performance of common SAT-Solvers relies primarily on two issues, namely unit propagation and backtracking. Unit propagation involves inferring information from clauses that have a single unassigned literal and assigning the literal a value such that the clause is then satisfied. Backtracking involves returning to the point in a search where a literal was assigned a value leading to a contradiction and reassigning that literal another value. Since the last few years there has been enormous progress in the performance of SAT solvers. Despite the worst-case exponential run time of all known algorithms, SAT solvers are increasingly leaving their mark as a general-purpose tool in areas as diverse as software and hardware verification, automatic test pattern generation, planning, scheduling, and even challenging problems from algebra. Our main motivation for this project was to explore the complexity of the DPLL algorithm, and to implement a subset of the Dynamic Largest Individual Sum (DLIS) heuristic.

Data Structure:

The data structure initialized in datatype.h are as follows:

1. vector <vector <int> > V_Literals – This represents a vector of a vector where every variable is related to a vector of clause numbers that hold that particular value. When a variable is present in the uncomplemented form then the clause number must be positive and when that variable is in the complemented form then the clause number must appear to be negative. We utilize this data structure as a look up table for the following tasks-

- **Looking for Pure Literals** – These are Literals having a positive or negative clause number throughout all the clauses of the CNF Boolean function which entered in DIMACS format. It can either be a uncomplemented pure literal, where the literal only appears as a positive clause number or as a complemented pure literal, where the literal only appears as a negative clause number.

- **Allocating a value to a Literal** – The complemented or uncomplemented literals gets the corresponding values assigned to it in accordance to the vector of clause numbers which contains the positive or negative clause number corresponding to each literal.
- **Performing Backtracking** – In this step we undo the previously assigned literal values to remove the conflict. It can be thought of as the reverse operation of the step where we allocate a value to a Literal.

2. vector <int> Literal_unknown – This represents a vector comprising the list of all the currently unassigned literals among all the clauses.

3. typedef struct Clause{int count_done;}Clause; – This count_done represents an integer variable of the structure Clause and it stores the number of Literals having assigned a satisfied value.

4. vector <Clause> V_Lookup – This vector represents a lookup table where a copy of all the clauses and literals are kept and used for backtracking.

5. vector <Clause> V_Clause – This vector contains all the clauses present in the CNF Boolean function whose satisfiability we are trying to check.

6. vector <int> V_occur – This represents a vector, comprising a list of all the occurrences of each literal.

The Data Structures initialized in the file main_alltest-1.cpp are as follows –

1.vector<int> pLiterals – This represents a vector that stores all the pure literals found at the start.

2.vector<int> foLiterals – This represents a vector that stores the current Forced Literal decisions.

3.vector<int> frLiterals – This represents a vector that stores the current Free Literal decisions.

4.vector<int> dLiterals – This represents a vector that stores all the literals whose values have already been decided.

5.vector<int> sat_nClause – This represents a vector that stores the satisfied clause number for certain variables.

6.vector<int> frLiterals_Pos – This represents a vector that stores indexes of the free literals in the vector "vector<int> dLiterals".

7.vector<int>nUniClause – This represents a vector that stores the total number of unit clauses.

8.vector<int>nUniClause_Lit – This represents a vector that stores the uni-clause literal.

9.vector<int>chrono_test – This represents a vector used for non-chronological backtracking.

The Data Structures initialized in the file parser_dimacs.h are as follows –

1.vector<char> V_ch – This represents a vector that stores any character that can be found in the input file containing the CNF Boolean function in Dimacs format and performs the necessary operations for parsing the different types of characters.

2.vector<int> Clause_Literals – This represents a vector that stores all the clause literals found in the CNF Boolean function after parsing.

Pseudo Algorithm:

Before jumping into the core dpll algorithm code we will preprocess the data. What is meant by preprocess is that we will first check if there is a clause with only one literal, then we will check if there is a literal which occurs only in single polarity throughout the clauses.

After we have checked the above condition we will check the forced literals possibility due to implications of these corner cases. For example after we checked the pure literal condition, we lookup for forced literals and then we check for possible conflicts, so if conflict occurs the entire system is unsolvable as pure literal fixes the assignment of that literal.

The way the code has been implemented it follows the binary constraint propagation logic. After assigning a free_literal in a certain iteration, we are investigating its possible implications and when we get a forced literal we set it to the implied value and that value is carried on to the clauses where that literal is there.

We are using a decision heuristic to select the variable on which to branch. What we are doing is that we are keeping an array where we are putting the number of the occurrences of the literal (irrespective of the polarity) and then after all the literals has been visited we are sorting it in descending order. We are then populating the free literal vector based on the above sorted array, so we are setting the variable which has occurred most number of times followed by variables with lesser occurrence. We found that the decision heuristic gave improvement in timing over when we are not using heuristic.

Our code assigns 1 to every free literal at first, and might be assigned to 0 if needed (if conflict arises) unlike mini-sat which assign any new free literal as 0 at first.

When all the preprocessing has been done with the help of the decision heuristic described above we populate the free literal vector. Then we pull out one free literal, set it to 1, and check its implications on possible forced literal (where the '1' assignment to the free literal doesn't satisfy the clause), set the forced literal accordingly, propagate the Boolean constraint to the other untouched clauses and check their satisfiability and so on. After forced literal has been set to appropriate values, we check for possible conflicts in the clause list due to our past decisions. If there is conflict we backtrack to our earlier free literal assignment. If the free literal of interest had been assigned 1 earlier, we now assign it to 0 and again check for forced literal and conflicts again. If the latest assigned free literal has been set to 0 already then it means that we have already checked for its '1' assignment, therefore we need to roll back to the free literal which we assigned before this one. Like this we continue backtracking and checking for possible conflicts. If we have reached top of the tree and have exhausted both our options (tried assigning both 1 and 0 to the variable) then we can't satisfy the CNF formula for the circuit hence the result is unsatisfiable.

The parser function is in separate file and it does the following:

Parser(filename)

{

Read file

Populate the Clause data base (V_Clause)

Populate the Literal data base (V_literal)

}

The core of the sat solver algorithm is as follows:

```
#include datatypes.h
#include parser_dimacs.h
Vector pure_literal;
Vector free_literal;
Vector forced_literal;
Vector decision_literal;
{
 Populate_occurence_array(); //populates the number of occurrence of the literals
Sort_occurence_array(); //sort in descending order of occurrence of the variables
Search_uniclause();//preprocessor to find whether there is a clause with one literal only at start
Search_pure_literal();//preprocessor to find whether there is a lit with same polarity throughout.
While (pure_literal !=empty())
        {
            decision_literal.push(pure_literal.front());
           pure_literal.erase(pure_literal.begin());
            assign_literal (decision_literal.back());
         }
Search_forced_literal(); //look for unit clause due to pure lit implication
While (forced_literal !=empty)
        {
            Find_conflict();
           If(conflict_found)
              { print("not possible to satisfy")
                Exit}
        }
Search_free_literal()// opulate the free_literal vector as per the decision heuristic used
While(free_literal!=empty)
        {
            decision_literal.push(free_literal.front);
```

```
        free_literals.erase(free_literals.begin())
        assign_literal (decision_literal.back());
        Search_forced_literal(); //look for unit clause due to pure lit implication
       While(forced_literal!=empty)
    {
        Find_conflict();
       Backtrack();// recursive function
      If(decision_literal == empty)
            {
              // we are at the top of the tree and not possible to satisfy
             Print( "not possible to satisfy")
                Exit;
            }
     // after the temporary conflict has been resolved we need to assign the forced literal
          decision_literal.push(forced_literal.front);
      remove the above forced literal from free literal vector;
       assign_literal (decision_literal.back());

    }
    }
If(number of satisfied clauses = = total number of clause)
Print("satisfied and the assignments")
Else
Print ("unsatisfied");
}
```

The backtrack algorithm does the following:

```
Backtrack()
{
     //roll back to last free literal decision
    decision_literal.pop();
```

free_literal.push()

if(free_literal.back() != complemented)

complement the variable in the decision literal

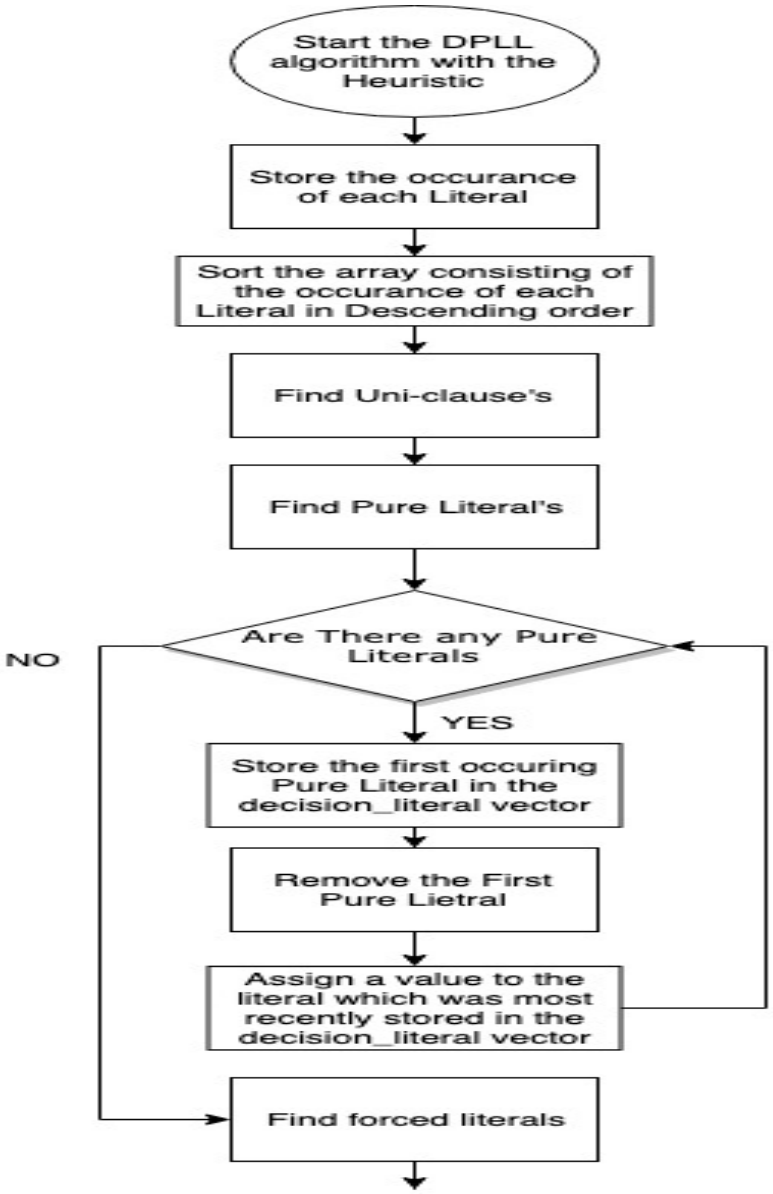
else

Backtrack();

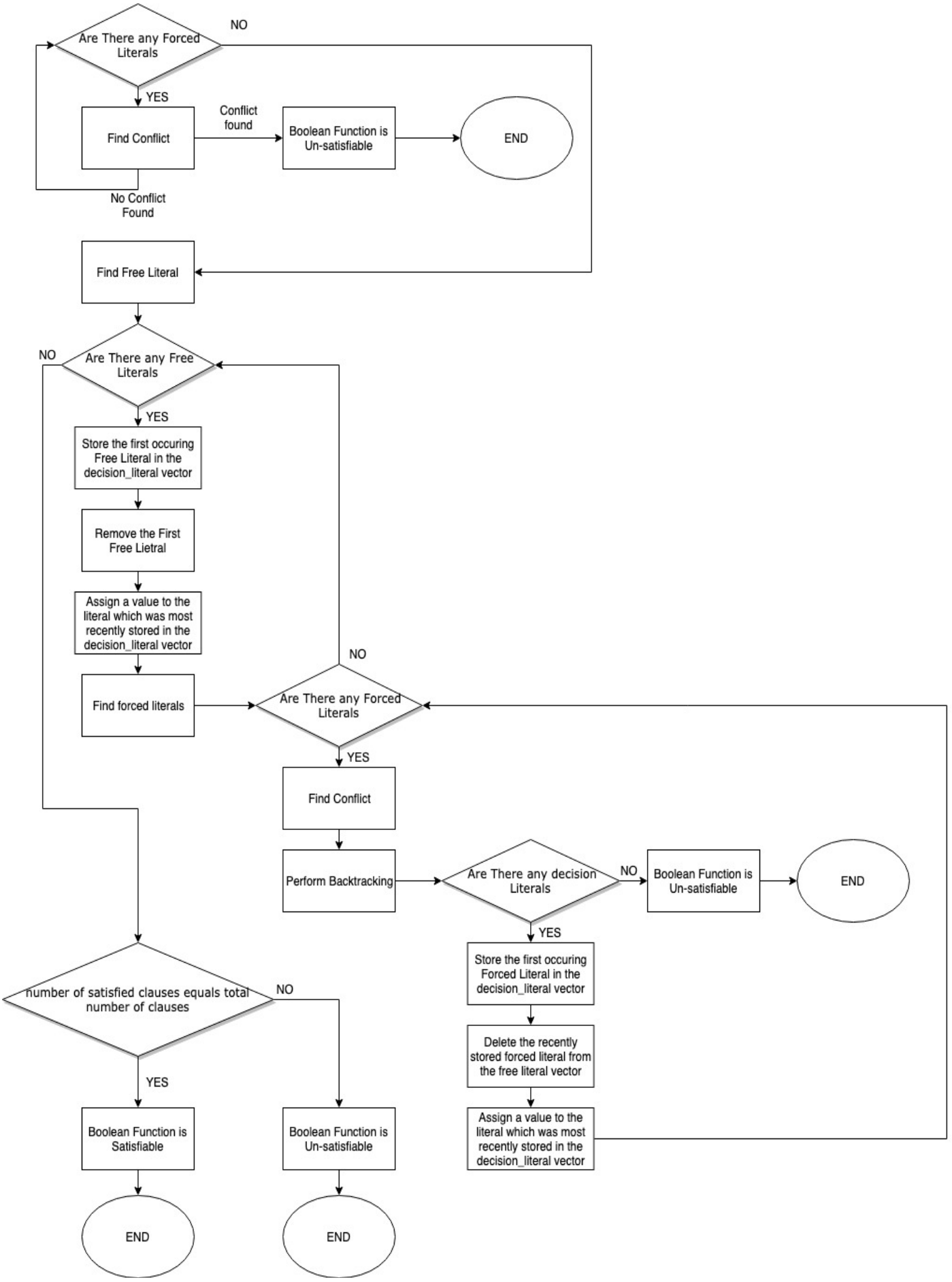
if(free_literals == empty) // reached at the top of the tree

exit

}



(NOTE - The Flow Chart continues on to the next page.)


Are There any Forced Literals
NO
YES
Find Conflict
Conflict found
Boolean Function is Un-satisfiable
END
No Conflict Found
Find Free Literal
Are There any Free Literals
NO
YES
Store the first occuring Free Literal in the decision_literal vector
Remove the First Free Lietral
Assign a value to the literal which was most recently stored in the decision_literal vector
Find forced literals
Are There any Forced Literals
NO
YES
Find Conflict
Perform Backtracking
Are There any decision Literals
NO
Boolean Function is Un-satisfiable
END
YES
Store the first occuring Forced Literal in the decision_literal vector
Delete the recently stored forced literal from the free literal vector
Assign a value to the literal which was most recently stored in the decision_literal vector
number of satisfied clauses equals total number of clauses
NO
YES
Boolean Function is Satisfiable
END
Boolean Function is Un-satisfiable
END

Results:

The result of the sat solver whenever a given input is satisfiable is printed out as follows:

RESULT: SAT

ASSIGNMENT:

1=0 2=0 6=0 3=1….45=1…..so on.

What this means is that as the minimum number of variables in our test benchmark is 50 we are naming each variable with its serial number and assigning it to 0 or 1;

For example from the above output sequence we get 1=0 which means variable 1 is assigned to 0; 45=1 means variable 45 is assigned to 1. The output sequence is not sorted, for example in the above sequence we can see the assignment for the 6^{th} variable (6=0) is done before 3^{rd} variable (3=1). But it can be using simple standard Vector sort function provided by C++ STL libraries.

When the CNF formula is not satisfied then it just prints the following.

RESULT: UNSAT

The following table summarizes the result. Timings are measured in seconds in all the measurement and timings are only measured across the core dpll algorithm which is called from the main function. The following table records the timing of the files listed for a certain set of number of variables and number of clauses when the decision heuristic we talked about earlier in the algorithm description section is **enabled**.

Set name	Numvar|NumClause	01.cnf	02.cnf	03.cnf
Sat_50_218	50|218	0.14	0.16	0
Sat_75_325	75|325	0.01	3.78	7.12
Sat_100_430	100|430	665.37	195.74	86.28
Sat_50_300	50|300	0	0	0
Sat_100_340	100|340	40.09	18.47	59.38
Unsat_50_218	50|218	0.14	0.2	0.18
Unsat_75_325	75|325	8.74	14.3	8.49
Unsat_100_430	100|430	1076.69	415.2	854.27
Unsat_50_80	50|80	1222.94	201.21	207.18

Table1: timing for benchmarks with decision heuristic enabled

The following table records the timing of the files listed for a certain set of number of variables and number of clauses when the decision heuristic we talked about earlier is **disabled**.

Set name	Numvar|NumClause	01.cnf	02.cnf	03.cnf
Sat_50_218	50|218	0.19	0.37	0
Sat_75_325	75|325	0.01	4.43	9.8
Sat_100_430	100|430	242.28	309.43	185.82
Sat_50_300	50|300	0	0	0
Sat_100_340	100|340	65.16	39.47	91.87
Unsat_50_218	50|218	0.19	0.2	0.27
Unsat_75_325	75|325	14.78	16.18	9.67
Unsat_100_430	100|430	1198.57	346.07	621.26
Unsat_50_80	50|80	1215.68	250.03	212.61

Table 2: timing for benchmarks with decision heuristic disabled

Analysis of the Results and scope of improvement:

As observed in the table of results we see that for most of the test cases when the decision heuristic is enabled then the decision making becomes faster with few exception test cases, marked in red in two tables above. This result aligns with our understanding that when the variable which occurs in most of the clauses is correctly assigned then the decision making becomes faster. The reason for the exception cases in our benchmark is that as discussed above unlike minisat we are assigning a free literal to 1 at first and propagate that to all the clauses where the free literal is there. May be in the exception cases most of the clauses where we are setting the free literal to 1 is not satisfying them in the first attempt and henceforth the algorithm needs to be backtracked and set the free literal to 0 and that's what is taking more time.

This problem can be overcome by the simple hack. As of now we are simply keeping the count of the variable irrespective of the polarity. For example if variable 1 is appearing in 10 clauses irrespective of polarity, the count of variable one is considered as 10 and that is what should be taken into account while sorting the occurrence array in order to make a list of free literal using the decision heuristic of selecting variable based on the descending order of occurrence. If we change the decision heuristic as follows, instead of not considering the polarity of the free literal while calculating the number of occurrence, if we consider, for example there will be 2 occurrence array. Let's say variable 1 occurs in 30 variables in un-complemented form and in 20 variables as a complemented form. Then while making the decision when we see that variable 1 is the free literal then based on the polarity the variable will be assigned. For the above example it will be assigned to 1 as 30 > 20. Plus we can make this decision heuristic a little more dynamic by decreasing the count number for each variable after a certain number of clauses has been satisfied by its specific assignment. Counts of a variable of certain polarity can also be increased when be backtrack after hitting a conflict case.

If we borrow the concept of VSID from Chaff then we can scale down the count of each variable by a constant amount randomly. This technique has proven to be helpful in many sat solvers so we suppose this implementation would have increased the speedup when the decision heuristic is used.

We tried implementing the non-chronological backtracking algorithm and conflict driven learning paradigm but we were getting segmentation fault in our code, so that's a coding error and if we would have spent more time on that then we could have debugged that issue and benchmarked the implementation

The key points that can be inferred from the results are as follows

1. The computation time increases with increase in the number of variables and number of clauses.
2. The time to get unsatisfied result is more generally
3. Using the decision heuristic based on descending order of occurrence most of the benchmarks gave better performance than the implementation without the decision heuristic.

Conclusion:

From this project we understood the complexity of the DPLL algorithm and successfully implemented a decision heuristic to efficiently select the free-literal based on the number of occurrences. Implementation of the decision heuristic makes the algorithm faster for most of the benchmarks. Our heuristic lacks certain flexibility and intricacies as discussed in the earlier section which if implemented can improve the efficiency of our code.