Simulator for Logic

Summer Project 2017

Under the Guidance of Prof. G. Sivakumar

Department of Computer Science and Engineering, IIT Bombay

NAME	COLLEGE
AKASH PRASANNA BASABHAT	PES University

May-July 2017

Contents

1. Introduction to SAT Solver.....3

2. THE DPLL Procedure.....4

3. Key Heuristics of Modern DPLL based SAT Solvers.....5

4. Binary Decision Diagrams and Reduced Binary Decision Diagrams.....8

5. Variable Ordering and BDD vs SAT.....9

6. Libraries used - PYEDA and DD 10

7. Deliverables of the Project.....11

8. References 14

9. Future Work.....15

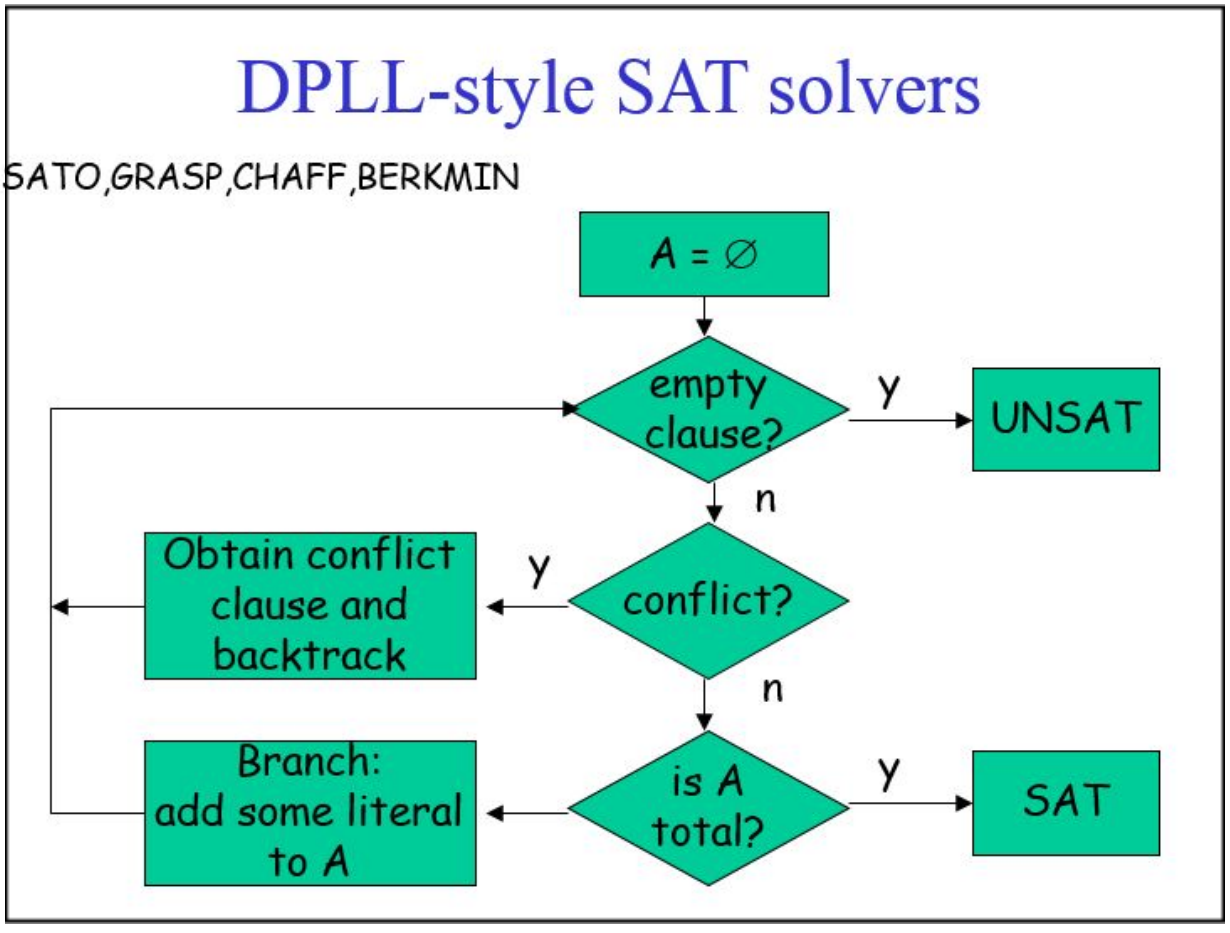
10. Acknowledgement.....16

Introduction to SAT Solvers

SAT is short for "satisfiability". SAT happens to fall under what are called *decision problems* in computer science. What that means is that the answer to a particular instance of the problem is either "yes" or "no". Decision problems are often simply identified with the set of inputs for which the answer is "yes", and that set is given a capitalized name.

In essence, SAT solvers provide a generic combinatorial reasoning and search platform. The underlying representational formalism is propositional logic. However, the full potential of SAT solvers only becomes apparent when one considers their use in applications that are not normally viewed as propositional reasoning tasks.

A *complete* solution method for the SAT problem is one that, given the input formula *F*, either produces a satisfying assignment for *F* or proves that *F* is unsatisfiable.



The DPLL Procedure

The Davis-Putnam-Logemann-Loveland or DPLL procedure is a complete, systematic search process for _nding a satisfying assignment for a given Boolean formula for proving that it is unsatisfiable. Davis and Putnam came up with the basic idea behind this procedure. However, it was only a couple of years later that Davis, Logemann, and Loveland presented it in the efficient top-down form in which it is widely used today. It is essentially a branching procedure that prunes the search space based on falsified clauses.

The algorithm, DPLL-recursive (*F*), sketches the basic DPLL procedure on CNF formulas. The idea is to repeatedly select an unassigned literal ` in the input formula *F* and recursively search for a satisfying assignment for *F*. The step where such an ` is chosen is commonly referred to as the *branching* step. Setting ` to TRUE or FALSE when making a recursive call is called a *decision*, and is associated with a *decision level* which equals the recursion depth at that stage. The end of each recursive call, which takes *F* back to fewer assigned variables, is called the *backtracking* step.

Algorithm 2.1: `DPLL-recursive`(F, ρ)

```
Input  : A CNF formula F and an initially empty partial assignment ρ
Output : UNSAT, or an assignment satisfying F
begin
    (F, ρ) ← UnitPropagate(F, ρ)
    if F contains the empty clause then return UNSAT
    if F has no clauses left then
        Output ρ
        return SAT
    ℓ ← a literal not assigned by ρ                    // the branching step
    if DPLL-recursive(F|ℓ, ρ ∪ {ℓ}) = SAT then return SAT
    return DPLL-recursive(F|¬ℓ, ρ ∪ {¬ℓ})
end

sub UnitPropagate(F, ρ)
begin
    while F contains no empty clause but has a unit clause x do
        F ← F|x
        ρ ← ρ ∪ {x}
    return (F, ρ)
end
```

Key Heuristics of Modern DPLL-Based SAT Solvers

The efficiency of state-of-the-art SAT solvers relies heavily on various features that have been developed, analyzed, and tested over the last decade. These include fast unit propagation using watched literals, learning mechanisms, deterministic and randomized restart strategies, effective constraint database management (clause deletion mechanisms), and smart static and dynamic branching heuristics.

Variable (and value) selection heuristic is one of the features that vary the most from one SAT solver to another. Also referred to as the *decision strategy*, it can have a significant impact on the efficiency of the solver. The commonly employed strategies vary from randomly fixing literals to maximizing a moderately complex function of the current variable- and clause-state, such as the MOMS (Maximum Occurrence in clauses of Minimum Size) heuristic or the BOHM heuristic. One could select and _x the literal occurring most frequently in the yet unsatisfied clauses (the DLIS (Dynamic Largest Individual Sum) heuristic) or choose a

literal based on its weight which periodically decays but is boosted if a clause in which it appears is used in deriving a conflict.

Clause learning has played a critical role in the success of modern complete SAT solvers. The idea here is to cache causes of conflict in a succinct manner (as learned clauses) and utilize this information to prune the search in a different part of the search space encountered later.

The watched literals scheme of Moskewicz, introduced in their solver zChaff, is now a standard method used by most SAT solvers for efficient constraint propagation. This technique falls in the category of lazy data structures introduced earlier by Zhang in the solver Sato. The key idea behind the watched literals scheme, as the name suggests, is to maintain and watch two special literals for each active (i.e., not yet satisfied) clause that are not FALSE under the current partial assignment; these literals could either be set to TRUE or be as yet unassigned.

Hence, one can always find such watched literals in all active clauses. Further, as long as a clause has two such literals, it cannot be involved in unit propagation. These literals are maintained as follows. Suppose a literal ` is set to FALSE. We perform two maintenance operations. First, for every clause *C* that had ` as a watched literal, we examine *C* and find, if possible, another literal to watch (one which is TRUE or still unassigned).

With this setup, one can test a clause for satisfiability by simply checking whether at least one of its two watched literals is TRUE. Moreover, the relatively small amount of extra book-keeping involved in maintaining watched literals is well paid off when one unassigns a literal ` by backtracking.in fact, one needs to do absolutely nothing!
The invariant about watched literals is maintained as such, saving a substantial amount of computation that would have been done otherwise. This technique has played a critical role in the success of SAT solvers, in particular those involving clause learning.

Even when large numbers of very long learned clauses are constantly added to the clause database, this technique allows propagation to be very efficient .The long added clauses are not even looked at unless one assigns a value to one of the literals being watched and potentially causes unit propagation.

Conflict-directed backjumping, introduced by Stallman and Sussman, allows a solver to backtrack directly to a decision level *d* if variables at levels

d or lower are the only ones involved in the conflicts in both branches at a point other than the branch variable itself. In this case, it is safe to assume that there is no solution extending the current branch at decision level *d*, and one may flip the corresponding variable at level *d* or backtrack further as appropriate. This process maintains the completeness of the procedure while significantly enhancing the efficiency in practice.

Fast backjumping is a slightly different technique, relevant mostly to the now popular *FirstUIP* learning scheme used in SAT solvers Grasp and zChaff.

It lets a solver jump directly to a lower decision level *d* when even one branch leads to a conflict involving variables at levels *d* or lower only (in addition to the variable at the current branch). Of course, for completeness, the current branch at level *d* is *not* marked as unsatisfiable; one simply selects a new variable and value for level *d* and continues with a new conflict clause added to the database and potentially a new implied variable. This is experimentally observed to increase efficiency in many benchmark problems. Note, however, that while conflict-directed backjumping is always beneficial, fast backjumping may not be so. It discards intermediate decisions which may actually be relevant and in the worst case will be made again unchanged after fast backjumping.

Assignment stack shrinking based on conflict clauses is a relatively new technique introduced by Nadel in the solver Jerusat, and is now used in other solvers as well. When a conflict occurs because a clause C_0 is violated and the resulting conflict clause C to be learned exceeds a certain threshold length, the solver backtracks to almost the highest decision level of the literals in C. It then starts assigning to FALSE the unassigned literals of the violated clause C_0 until a new conflict is encountered, which is expected to result in a smaller and more pertinent conflict clause to be learned.

Conflict clause minimization was introduced by E´en and S¨orensson in their solver MiniSat. The idea is to try to reduce the size of a learned conflict clause C by repeatedly identifying and removing any literals of C that are implied to be FALSE when the rest of the literals in C are set to FALSE. This is achieved using the subsumption resolution rule, which lets one derive a clause A from $(x \vee A)$ and $(\neg x \vee B)$ where $B \subseteq A$ (the derived

clause *A* subsumes the antecedent (*x_A*)). This rule can be generalized, at the expense of extra computational cost that usually pays off, to a sequence of subsumption resolution derivations such that the final derived clause subsumes the first antecedent clause
.
Randomized restarts, introduced by Gomes, allow clause learning algorithms to arbitrarily stop the search and restart their branching process from decision level zero. All clauses learned so far are retained and now treated as additional initial clauses. Most of the current SAT solvers, starting with zChaff, employ aggressive restart strategies, sometimes restarting after as few as 20 to 50 backtracks.

This has been shown to help immensely in reducing the solution time. Theoretically, unlimited restarts, performed at the correct step, can probably make clause learning very powerful. We will discuss randomized restarts in more detail later in the chapter.

Binary Decision Diagrams and Reduced Binary Decision Diagrams

A **binary decision diagram** (**BDD**) or **branching program** is a data structure that is used to represent a Boolean function. On a more abstract level, BDDs can be considered as a compressed representation of sets or relations.

A Binary Decision Diagram (BDD) is a rooted, directed acyclic graph. A BDD is used to compactly represent the truth table, and therefore complete functional description, of a Boolean function. Vertices of a BDD are called *terminal* if they have no outgoing edges and are called *non-terminal* otherwise. There is one non-terminal vertex, called

the *root*, which has no incoming edge. There is at least one and there are at most two terminal vertices, one labelled *0* and one labelled *1*. Non-terminal vertices are labelled to represent the variables of the corresponding Boolean function.

Reduced ordered decision diagrams (ROBDDs) are based on a fixed ordering of the variables and have the additional property of being reduced. This means:

Irredundancy: The low and high successors of every node are distinct.

Uniqueness: There are no two distinct nodes testing the same variable with the same successors.

ROBBDs recover the important canonicity property: for a fixed variable ordering, each boolean function has a canonical (unique) representation as an ROBDD. This means we can compare boolean functions by constructing their ROBDDs and checking if they are equal.

ROBDDs have at most two leaf nodes, labelled 0 and 1. We sometimes draw them multiple times to avoid complex edges leading to them.

Variable Ordering of Binary Decision Diagram

The size of the BDD for a given function depends on the order chosen for the variables. There are functions—sums of products where each product has support disjoint from the others—for which some orders lead to BDDs that are linear in the number of variables, whereas other orders give numbers of nodes that are exponential in the number of variables. The same occurs for the BDDs of adders. At the other end of the spectrum there are functions for which all possible BDDs are provably exponential in size and functions whose BDDs are linear for all orderings.

Though clearly one would not want to choose a worst case ordering for an adder, the importance of the ordering problem cannot be argued from the existence of functions exhibiting such an extreme behavior; equally well, the ordering problem cannot be dismissed simply because there are functions that are insensitive to the ordering. Indeed, even simple heuristics easily avoid the worst case behavior for adders.

The practical relevance of the variable ordering problem rests on the existence of functions lacking a well understood structure, for which different orders—all derived by plausible methods—give widely different results.

BDD VS SAT

- Many models that cannot be solved by BDD symbolic model checkers, can be solved with an optimized SAT Bounded Model Checker.
- The reverse is true as well.
- BMC with SAT is faster at finding shallow errors and giving short counterexamples.
- BDD based procedures are better at proving absence of errors.

Libraries Employed in the Project

PYEDA

PyEDA is a Python library for electronic design automation.

Features:

- Symbolic Boolean algebra with a selection of function representations:
  - Logic expressions
  - Truth tables, with three output states (0, 1, “don’t care”)
  - Reduced, ordered binary decision diagrams (ROBDDs)
- SAT solvers:
  - Backtracking
  - PicoSAT
- Espresso logic minimization
- Formal equivalence
- Multi-dimensional bit vectors
- DIMACS CNF/SAT parsers
- Logic expression parser

DD

A pure-Python (3 and 2) package for manipulating:

- Binary decision diagrams (BDDs).
- Multi-valued decision diagrams (MDDs).

The intended workflow is:

- develop your algorithm in pure Python (easy to debug and introspect),
- use the bindings to benchmark and deploy

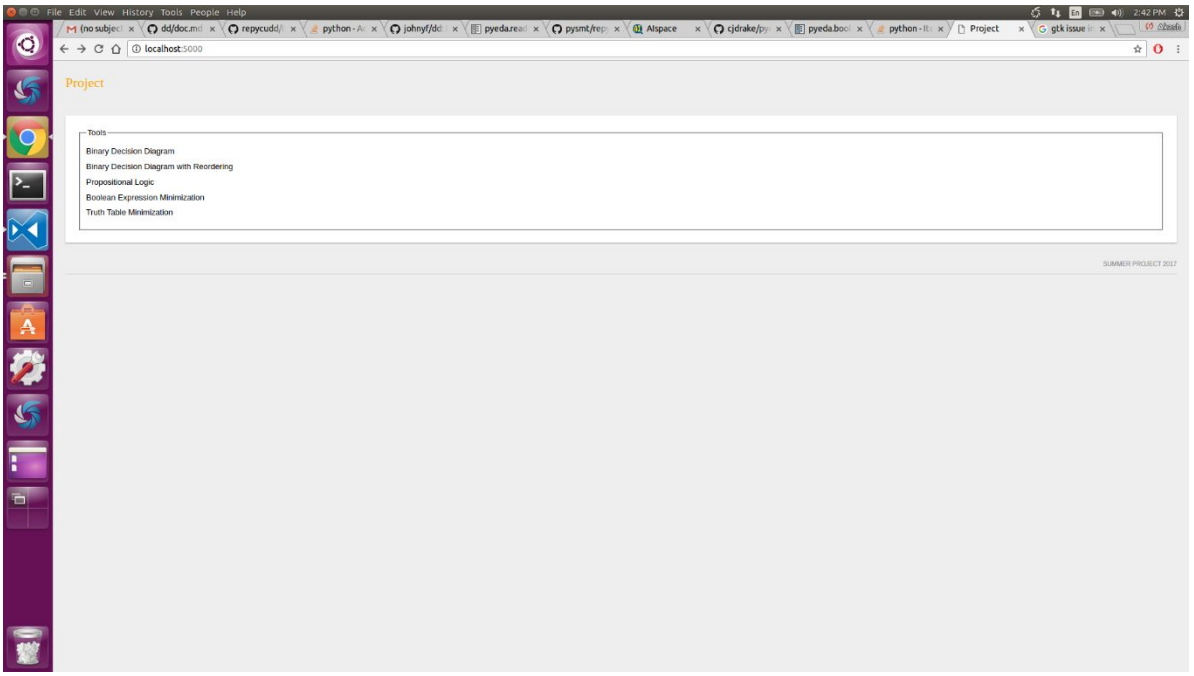
Contains:

- Dynamic variable reordering using Rudell's sifting algorithm.
- Reordering to obtain a given order.
- Conversion from BDDs to MDDs.
- Conversion functions to networkx and pydot graphs.

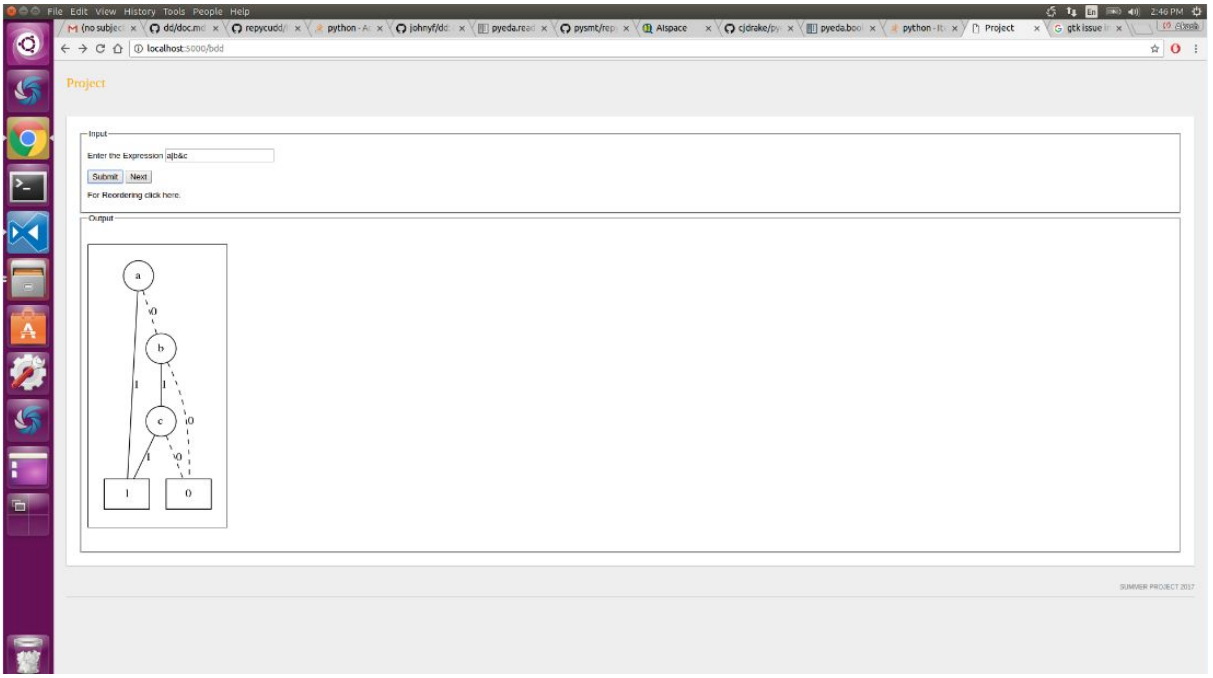
Deliverable of the project

The aim of our project was to build a teaching tool for binary decision diagrams on a java independent web browser. Our tool would showcase to the learner the step by step production of binary decision diagrams. It would also

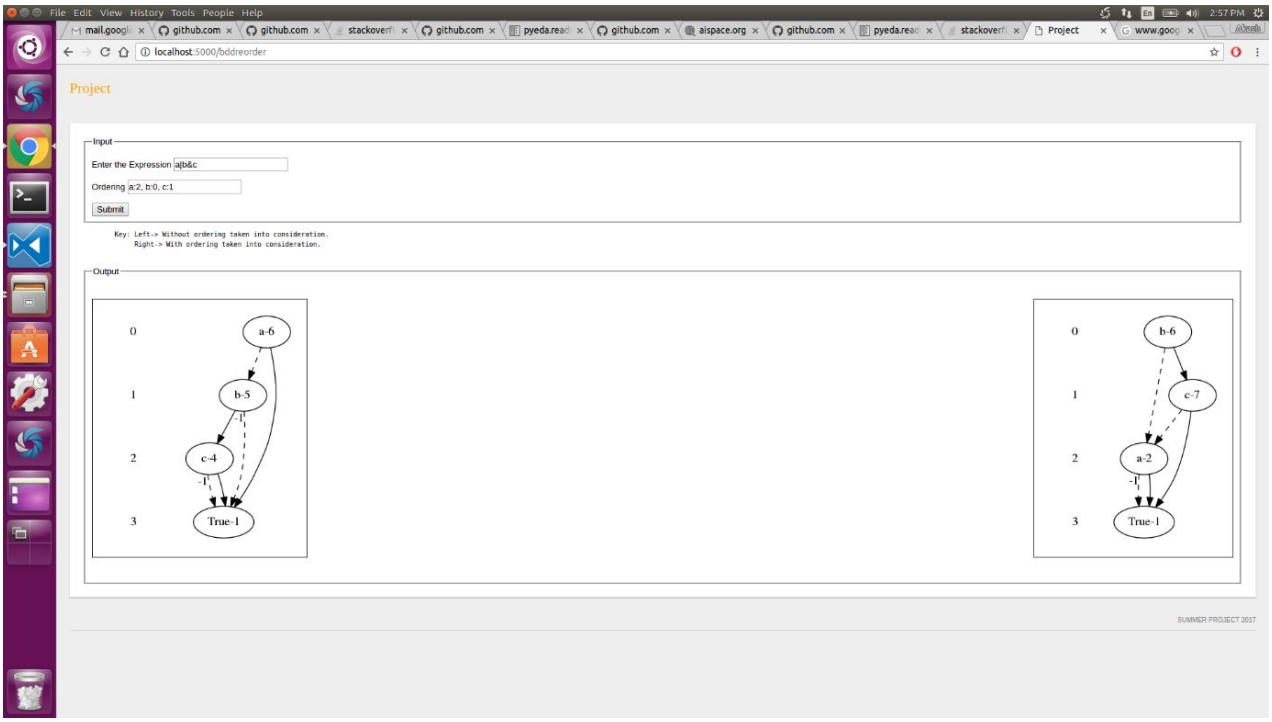
have the option for production of variables based on a certain order as given by the learner.



Main page of our web app project



Page illustrating the generation of Binary Decision Diagrams



Page illustrating reordering in Binary Decision Diagrams

Our main aim was to produce a tool similar to Visbdd (https://bdd.hpi.uni-potsdam.de/visBDD/) on a java independent browser. We understood the working of the tool and the generation of the binary decision diagrams in this tool first and then tried to understand the various approaches to the problem statement.

We had libraries such as pyeda and dd, but we needed to implement the step through production of the binary decision diagrams along with the ordering. We made changes to the code of bdd.py in pyeda and reorder.py in dd to implement the same. This needed us to brush up on our reading and understanding of python. We also made significant changes to the code and added our own functions.

For the back end, we used Flask. Flask is a micro-framework for Python based on Werkzeug toolkit and Jinja 2 template engine. It was a convenient way to implement client-server based framework.

References

- Lecture notes on Binary Decision Diagrams - https://www.cs.cmu.edu/~fp/courses/15122-f10/lectures/19-bdds.pdf
- Binary Decision Diagrams – Umasswww.ecs.umass.edu/ece/labs/vlsicad/ece667/reading/somenzi99bdd.
- Binary Decision Diagrams – Dept of CSE, IIT Madras www.cse.iitm.ac.in/~manas/docs/BDDs.pdf
- Lecture 11 – BDD's – IIT KGP www.facweb.iitkgp.ernet.in/~isg/SWITCHING/SLIDES/05-BDD.pdF
- BDD's @EECS - UC Berkeley https://people.eecs.berkeley.edu/~sseshia/219c/lectures/BinaryDecisionDiagrams.pdf
- OBDD and ROBDD – NPTEL https://nptel.ac.in/courses/106103016/12
- An Introduction to Binary Decision Diagrams – NTU www2.ee.ntu.edu.tw/~yen/courses/cav02/restricted/bdd-survey.ps
- visBDD - Visualization of the OBDD - ITE - Algorithm bdd.hpi.uni-potsdam.de/visBDD/
- Stack Overflow Wikipedia
- Wolfram Alpha
- PyPI
- AIspace.org

FUTURE WORK

- To implement the step by step generation of binary decision diagrams while keeping order in mind.

- To use some heuristics which produce bdds on a criteria as mentioned by the learner explicitly.

- To illustrate and compare two or more bdds produced by different heuristics for the learner to understand the differences between the two.

- To build new tools apart from those which we have created on our web app and add further tools such as those seen in AIspace to our web app

ACKNOWLEDGEMENT

Through this project, I would like to express my sincere gratitude to Prof. G. Sivakumar for providing me with invaluable guidance, direction and suggestions throughout the course of my internship.

This was my first project which had elements of AI and logic, and it was only through the guidance of Prof. G. Sivakumar, that I found my web app interesting and enjoyable to build.

This project was a great kickstarter project for future projects in the AI domain and I shall continue to build and implement such projects to improve and understand the basics of AI better.

Once again, I would like to thank Prof. G. Sivakumar for providing me with such an invaluable learning experience under his guidance which shall definitely help me in my future when I explore future projects in logic.