SAT SOLVER
Report

Implementation of CDCL-based SAT Solver

Student:

Kukreja Kapil Haresh

Under guidance:

Dr. Sukanta Bhattacharjee

Course:

CS508 - Optimization Methods

Introduction

SAT is NP-complete and no known polynomial-time algorithm for it exist till date. Still many improvements over the basic backtracking algorithms can be achieved and have been demonstrated in past few decades including DP, DPLL, CDCL etc. This project implements the CDCL algorithm which is used at the core of most of modern SAT solvers.

The implementation follows the DIMACS input/output requirements and it is assumed that input is in valid CNF format.

> Better than DPLL?

The CDCL algorithm improves on the following extensions to DPLL:

1. Naive decisions: DPLL chooses an arbitrary variable and starts by assuming it to be True. It Fails to consider the problem or structure to make heuristically better decisions.
2. No learning: DPLL fails to gain any information from the current state and conflicts. Hence, may revisit bad partial decisions that had led to the conflict.

Key Features of Implementation:

1. Decision heuristics: VSIDS (Variable State Independent Decaying Sum)
2. 2- Literal Watch: Two literal watch data structure based BCP
3. Random Restarts (with decaying probability)

CDCL algorithm

CDCL_solve()

```
If UnitPropagation() == CONFLICT :
        then return UNSAT
while (not AllVariablesAssigned()) :
        v = VSIDS_decide()  pick a variable by decision heuristics
        while UnitPropagation(v) == CONFLICT:
                dec_level = ConflictAnalysis(v)
                if (dec_level < 0)
                        then return UNSAT
                else Backtrack(dec_level)
return SAT
```

Program:

For better understanding of implementation, please refer to the code and the comments. The code has been highly commented with explanation and working. Here, some key features and their advantages/disadvantages are discussed:

1. **Decision heuristics: VSIDS** (Variable State Independent Decaying Sum)

   VSIDS was proposed more than fifteen years ago and still remains one of the most effective branching heuristics.

   Working:

   - Initially, Count the number of all clauses in which a literal appears. Function: VSIDS_init()
   - Periodically divide all scores by a constant. In our case, similar to MiniSAT decay is by 5% after each conflict. Function: VSIDS_decay()
   - Increment the Variables involved in current conflicts to get higher scores. Function: VSIDS_conflict()
   - Constant decision time when literals kept in a sorted list. By VSIDS_decide()

Two major advantages of VSIDS are:

1. VSIDS is has negligible computational costs;
2. VSIDS gives preference to literals that participate in recent conflicts, i.e. it is Dynamic to the current state of Solver

2. Two Literal Watch Data Structure:

Solvers spend most of their time in BCP (approx. 80%), so this must be efficient. Naive implementation won't work on large problems.

Implementation:

Every clause that remains to be satisfied will have two watch literals associated with it. Instead of keeping track of which clauses are yet to be satisfied, we keep a list of clauses for each literal such that the literal is one of the two watch literals assigned to that clause.

The following cases arrive:

1. A literal in the clause is updated, which is not watched: In this case, we don't need to do anything, not even check whether any update is required.
2. A literal from watch is set false: Here we find another literal to replace this literal and after storing this information in watch, we continue with the search.
3. In case 2 if we can no longer find an alternative: In this case, the remaining literal is the UNIT clause. And hence after assigning as Unit, this clause is satisfied and we continue the search.
4. Backtrack: The watch literals remain unaffected by backtrack, thus saving from large computational costs.
5. In all other cases, the watch literals and clauses remain unaffected.

3. Clause Learning:

Whenever a conflict is found, the Conflict Analyze function makes use of this information to learn a new clause, and avoid the same path in any other subspace in the search in future.

Any cut that separates the conflict from decisions in implication graph can be a learned clause. For simplicity yet effective approach, we consider all decisions involved in the conflict till now, negated, as a clause learnt. This clause is also added to the literal watch data structure.

4. Random Restarts:

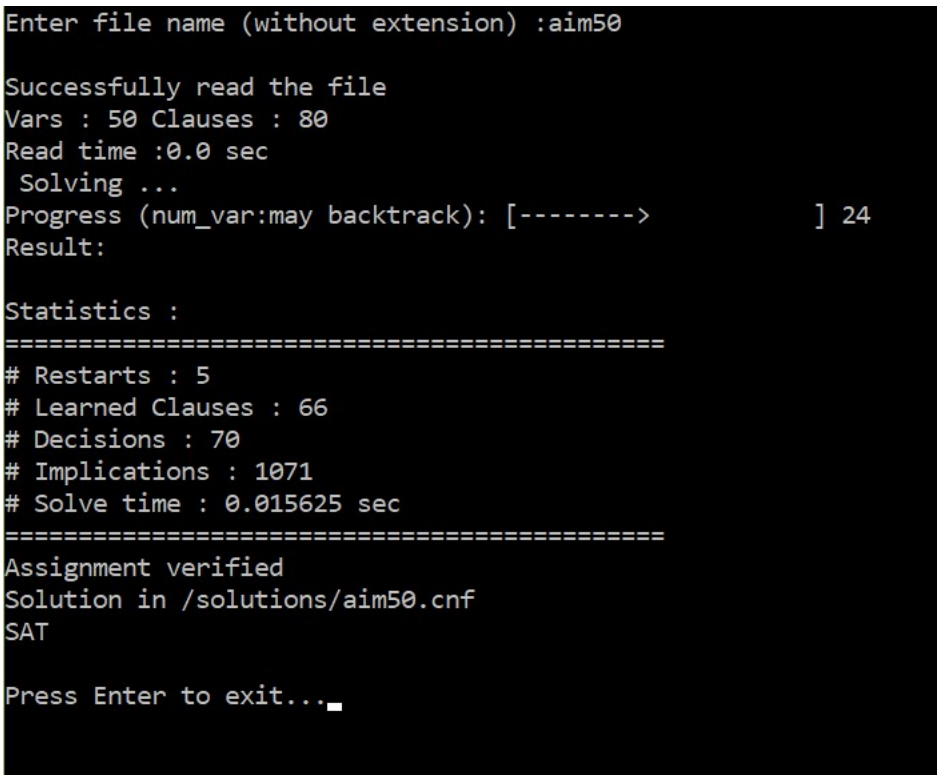
The execution is restarted with all learnt clauses and no decisions with a pre-defined probability. This probability is decayed for next restart by 50% at each restart to ensure the completeness and the program does not get stuck in restarts for large instances.

➔Comparison of run time on Benchmark files:

File / Solver	bmc-2.cnf	bmc-7.cnf	unsat3.cnf	par8.cnf	aim-50	aim100	zebra
Variables:	2810	8710	13	64	50	100	155
Our Solver	15.5	244.1	0.001	0.014	0.015	0.013	0.016
Edusat	0.26	0.324	0.15	0.014	0.031	0.013	0.4

*time in seconds

➔ **Output Screenshot:**



➔CONCLUSION:

The correctness of the SAT Solver was verified through some of the Benchmarks from various sources.

The Solver performs excellently for variables ≈ till 2000.

The performance starts degrading for very large instances which can be optimized further in future work by learned clause deletion and Unique Implication Point based clause learning.