Karan Sachdeva
301368050

Fast Implementations of WalkSat and Resolution Proving

Analysis Report

What is WalkSat?

In computer science, WalkSat is a local search algorithm that is used to solve Boolean satisfiability problems. WalkSat first picks a clause which is unsatisfied by the current assignment, then flips a variable within that clause. The clause is then picked at random among unsatisfied clauses. The variable is picked that will result in the fewest previously satisfied clauses becoming unsatisfied, with some probability of picking one of the variables at random. When picking at random, WalkSat ensures a chance of one out of the number of variables in the clause of fixing a currently incorrect assignment.

WalkSat is a randomized algorithm and so cannot prove that a clause is unsatisfiable. But it is perhaps surprising that it can work so well as it has undergone extensive testing, and it can efficiently satisfy many large clause sentences.

Pseudo Code for WalkSat

function WALKSAT(*clauses*, *p*, *max_flips*) **returns** a satisfying model or *failure*
  inputs: *clauses*, a set of clauses in propositional logic
    p, the probability of choosing to do a "random walk" move, typically around 0.5
    max_flips, number of flips allowed before giving up

  model ← a random assignment of *true*/*false* to the symbols in *clauses*
  for *i* = 1 **to** *max_flips* **do**
    if *model* satisfies *clauses* **then return** *model*
    clause ← a randomly selected clause from *clauses* that is false in *model*
    with probability *p* flip the value in *model* of a randomly selected symbol from *clause*
    else flip whichever symbol in *clause* maximizes the number of satisfied clauses
  return *failure*

Figure 7.18 The WALKSAT algorithm for checking satisfiability by randomly flipping the values of variables. Many versions of the algorithm exist.

Generating Random Clauses with Python

In our WalkSat, we made sure that its input is in conjunctive normal form (CNF). CNF is composed of variables(y), literals(k) and clauses(x).

An example of CNF is:

(x1 | -x5 | x4) &
(-x1 | x5 | x3 | x4) &
(-x3 | x4).

MiniSat, like most SAT solvers, accepts its input in a simplified "DIMACS CNF" format, which is a simple text format. Every line beginning "c" is a comment. The first non-comment line must be of the form:

So, the CNF expression above would be written as:

c a comment.

p cnf 5 3
1 -5 4 0
-1 5 3 4 0
-3 -4 0

Note that the python file cnf.py randomly generates k-CNF expressions in the “DIMACS CNF” format which is the same as the MiniSat input format as asked in the assignment. We only used the random library for our program.

Implementation of WalkSat in Java

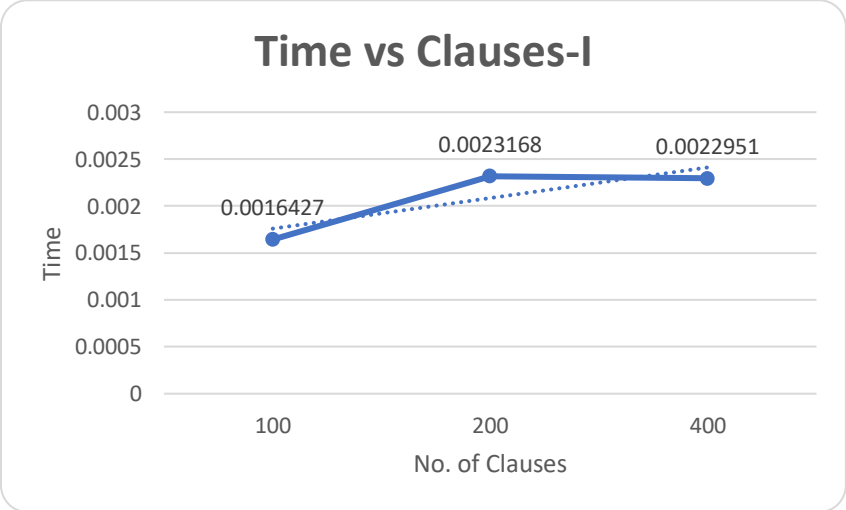
To implement our WalkSat, we use an efficient language like java as Python will not be efficient enough. We used the following java libraries:
import java.util.*

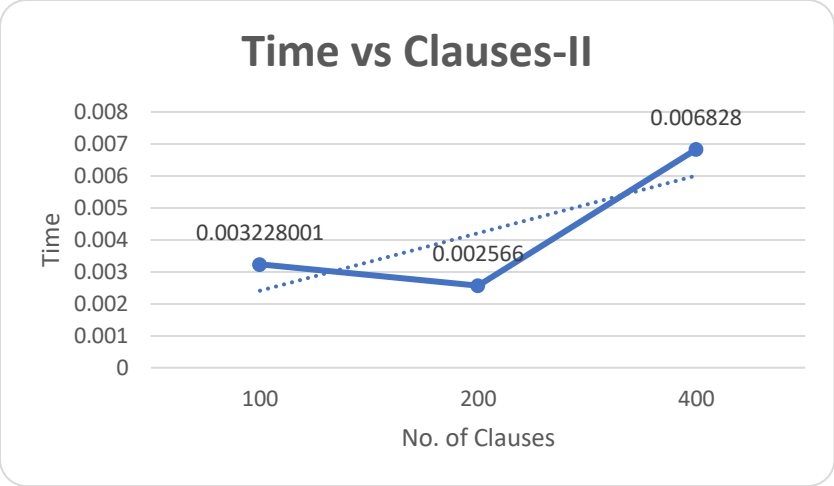
```
import java.util.stream.*
import java.io.*
import java.nio.file.Paths
import static java.nio.file.Files.lines
import static java.lang.System.exit
```

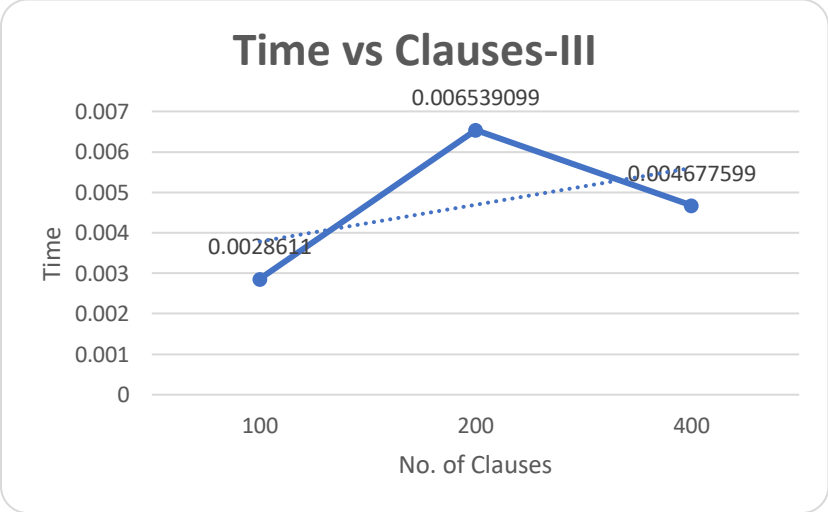
Analysis of WalkSat

The following is the data obtained by our WalkSat program.

k (literals)	x (clauses)	y (variables)	time (seconds)
8	100	10	0.0016427
8	200	10	0.0023168
8	400	10	0.0022951
16	100	20	0.003228001
16	200	20	0.002566
16	400	20	0.006828
32	100	40	0.0028611
32	200	40	0.006539099
32	400	40	0.004677599







Conclusion

From the data above, we can conclude that increasing the number of clauses increases in time. This linear trend is also proved by the dotted line in the above graphs. It can also be concluded that time is increased by increasing the number of literals(k) and the number of variables(y). While running the program, one might also find no solution for a specific clause but that does not prove anything about the clause.

References

- https://en.wikipedia.org/wiki/WalkSAT
- https://www2.cs.sfu.ca/CourseCentral/310/tjd/chp7_sat.html
- Artificial Intelligence: A Modern Approach (3rd Edition), Stuart Russell, Peter Norvig