Project: Fast Implementations of WalkSAT and Resolution Proving

Environment:

OS: Windows 10, 64-bit, 8GB RAM

Processor: intel core i5 , CPU at 1.80 GHz

Used MinGW to simulate terminal like environment on Windows, as I do not have access to Linux machine.

Program Structure

- Generate_random_KCNF.py
  - Generates a random CNF clause and outputs in miniSAT input format.
- walksat folder:
  - contains program for faster implementation of WALKSAT in C++
    - use “$make all” command to compile the code when inside the folder.
    - Use “$./walkSAT (filename)” to run, filename can be generated using generate_random_KCNF.py file.
- Resolution folder:
  - Resolution Proving implemented in C++
    - use “$make all” command to compile the code when inside the folder.
    - Use “$./resolution (filename)” to run, filename can be generated using generate_random_KCNF.py file.

Note: Results can be reproduced typing above commands in respective folder.

Data structures used

- **Map** in C++, to represent all clauses.
  - map<string, set<int>> clauses_
- **Set** was used to present unsatisfied clauses.
  - set<string> unsat_clause_

Implementation technique

Initially, every time ‘check_assigned_model()’ function is called after an assignment change, the function checks all clauses. It then generates a vector of unsatisfied clauses.

Previously, the python WalkSAT algorithm was inefficient as it was repeating the same thing and expecting different results in clauses that does not contain the variable that was flipped.

Here, the optimized version of WalkSAT checks if the clause contains the changed variable first, rather repeating the same thing and expecting a different output. If the changed variable does exist in the clause then either we insert the clause or erase it from the unsatisfied clause set based on the model.

It only takes O(Log N) time to check an ordered set, which is a significant improvement from previous running time of O(N). Moreover, by using a vector of string, the erasing time was reduced to O(Log N) from O(N).

Furthermore, the data structure to hold the clauses was changed from vector<vector<int>> to map<string, set<int>>. This gave us a faster look up time of O(Log N).

The variable to flip is decided by the highest number of satisfied clauses the flip results in. This overhead time of determining a variable to flip was reduced by keeping a counter on the increase/decrease of unsat_clause_ count.

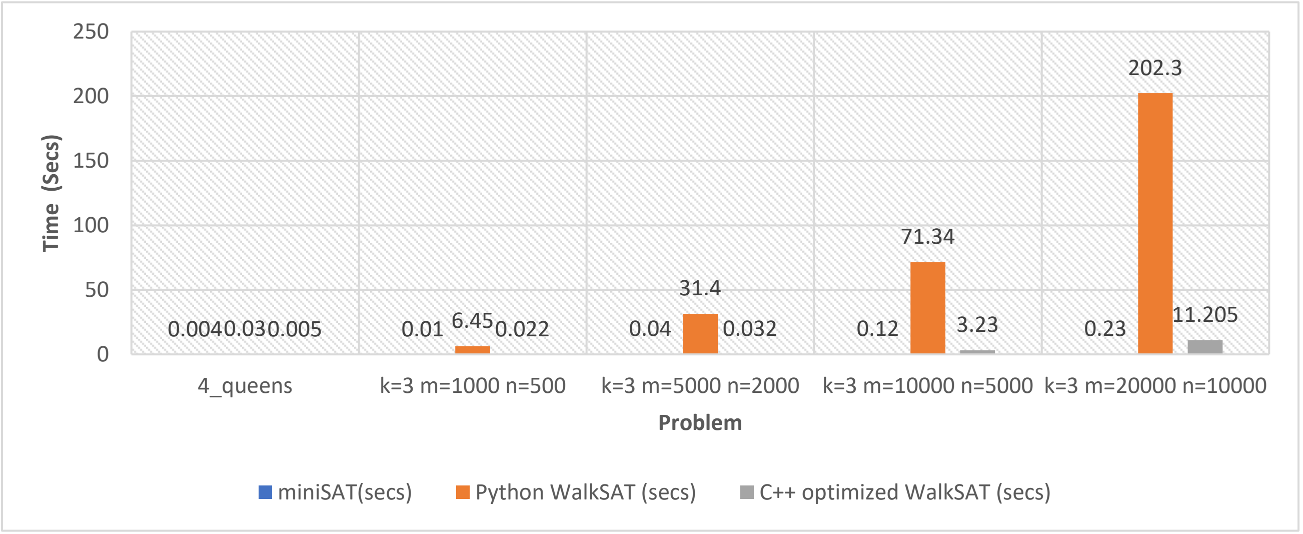
Comparison

One possible reason of why python implementation of WalkSAT is slower than C++, because of the overhead time of calculating the best variable to flip. Python is unable to do this calculation very efficiently. While C++ being a statically typed language, C++ creates more compact and faster runtime code. Another reason being the use of abstract data structures in C++ compared to python.

WalkSAT

k = literals per clause, m = number of clauses, n= number of variables

Problem	miniSAT(secs)	Python WalkSAT (secs)	C++ optimized WalkSAT (secs)
4_queens	0.0040	0.030	0.005
k=3 m=1000 n=500	0.01	6.45	0.022
k=3 m=5000 n=2000	0.04	31.40	0.032
k=3 m=10000 n=5000	0.12	71.34	3.23
k=3 m=20000 n=10000	0.23	202.30	11.205

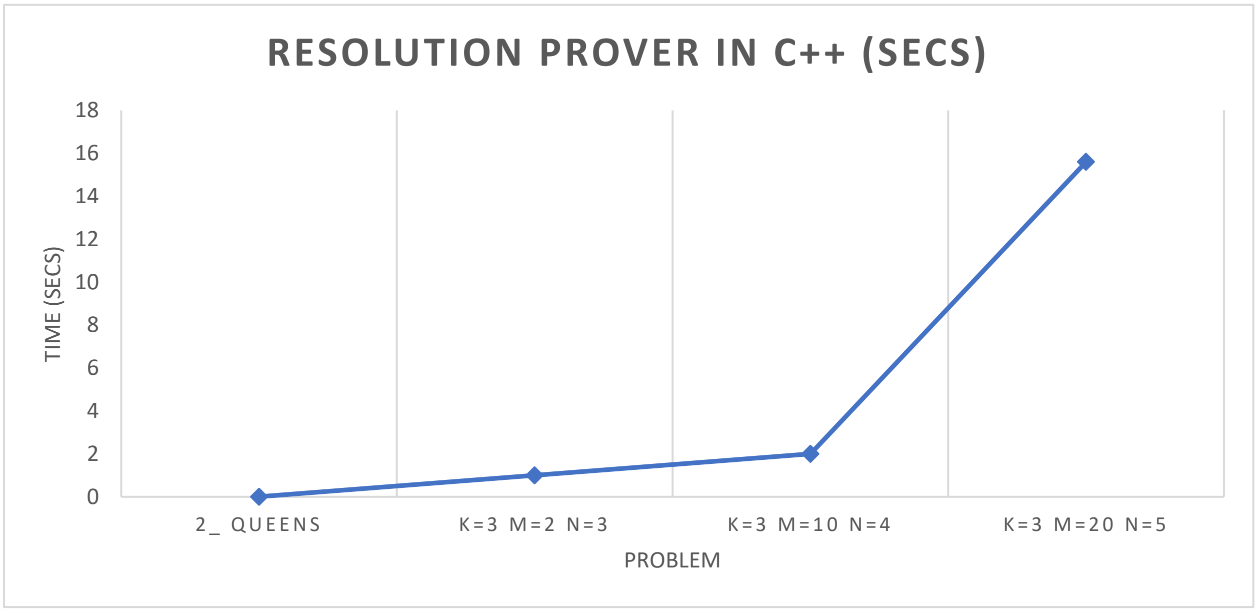


Resolution prover running time:

k = literals per clause, m = number of clauses, n= number of variables

note: a smaller value of value of **m** and **n** was is used to run Resolution prover because for a large value such as used above for WalkSAT, the program took few hours to determine satisfiability. So, only for the purpose of this project a smaller value was used. For example, the time to determine satisfiability for 3_queens problem is shown below.

Problem	Resolution Prover in C++ (secs)
2_ queens	0.004
k=3 m=2 n=3	1
k=3 m=10 n=4	2
k=3 m=20 n=5	15.6
3_queens	5280


RESOLUTION PROVER IN C++ (SECS)
TIME (SECS)
2_ QUEENS
K=3 M=2 N=3
K=3 M=10 N=4
K=3 M=20 N=5
PROBLEM