**Assignment 5 Report**

**WalkSAT**

**Optimizations in Point Form:**

1. C++ code
2. Optimized loops and conditionals (ran speed tests for different structures and used fastest)
3. Passing by reference (creating new copies of vectors takes time)
4. Using hashmaps and hashsets (saves time for look-up and update by element)
5. Testing for pure variables (assign proper value and remove clauses that contain them)
6. Using variable to clause mapping (hashmap to clauses that contain the variable)
7. Using dynamic programing to store the state of different clauses (minimizes times program calculates whether a clause is satisfied)
8. When a clause is unsatisfied, if any of the variables it contains flips the clause will be satisfied. (do not need to recalculate if the clause is satisfied)

Optimizations 6-8 are used in “walk\_sat\_dynamic\_programming.cpp”, making the program an order of magnitude faster than “walk\_sat.cpp” which only uses optimizations 1-6.

**My Implementations:** **“walk\_sat\_dynamic\_programming.cpp”** (the optimized one) and “walk\_sat.cpp”

I tested the correctness of my programs with 3-queens (unsatisfiable) and 4-queens (has 2 possible correct variable assignments). My WalkSAT is still drastically outperformed by MiniSAT the fastest freely available SAT solver.

**Times:**

Generated k-CNF sentence:

*((literals per clause) k = 4, (# clauses generated) m = 20000, (# symbols) n = 4400)*

“walk\_sat\_dynamic\_programming.cpp”: 1.047s (averaged over 10 runs)

“walk\_sat.cpp”: 22.391s (averaged over 5 runs)

“logic.py”: 29.979s (averaged over 5 runs)

“minisat”: 0.0156s (averaged over 10 runs)

4Queens:

“walk\_sat\_dynamic\_programming.cpp”: 0.018s (averaged over 10 runs)

“walk\_sat.cpp”: 0.017s (averaged over 10 runs)

“logic.py”: 0.073s (averaged over 10 runs)

“minisat”: 0.000s (averaged over 10 runs)

3Queens (unsatisfiable):

“walk\_sat\_dynamic\_programming.cpp”: 0.093s (averaged over 10 runs)

“walk\_sat.cpp”: 0.136s (averaged over 10 runs)

“logic.py”: 0.093s (averaged over 10 runs)

“minisat”: 0.000s (averaged over 10 runs)

\*timed using Linux command line “time” function

\*max\_flips for “logic.py” is 1,000 where for my implementations it is 10,000

My implementation of WalkSAT with dynamic programming is exponential regarding problem size, but the rate of increase is slower than regular WalkSAT and the textbook implementation. Therefore, the more complex the problem, the more WalkSAT\_Dynamic outperforms the other two programs percentagewise. MiniSAT is still by far the fastest SAT solver due to year of refinement and optimization by the open source community.

**Potential Improvements**

WalkSAT dynamic could make use of heuristics to determine the best clauses to pick rather than random selection. The use of more sophisticated memory management techniques could also greatly improve my programs performance since I am using dynamic programming.

**Resolution Prover**

**Optimizations in Point Form:**

1. If two clauses share more than one pair of complementary literals, then any resolvents will be a tautology. Therefore, my resolution algorithm automatically returns none in these cases.
2. Pass by Reference
3. Use unordered sets for faster clause comparisons

Resolution Prover takes in a KB in minisat input format and a query from the command line and determines if the query is entailed by the KB. When the query is not entailed by the KB, the resolution prover takes much longer as it must exhaust all possible resolvent clauses.

The Resolution Prover program is much slower than the SAT solvers. The Resolution Prover program cannot solve the problems sets used for the SAT solvers in a reasonable amount of time, thus I am using a different one.

**Times:**

Generated k-CNF sentence:

*((literals per clause) k = 2, (# clauses generated) m = 2000, (# symbols) n = 440)*

*(query is entailed)*

“propositional\_theorem\_prover.cpp”: 15.774s (averaged over 5 runs)

\*timed using Linux command line “time” function

\*increasing the literals per clause increases time complexity exponentially due to having no O(1) equality comparisons for clauses

**Potential Improvements**

The main components slowing down my program is that I do not have an efficient way to determine whether the new clauses are a subset of the known clauses, as well as no O(1) equality comparisons for clauses. In this implementation, I am using a double loop to compare all values in both sets.

Ideally, I would create a custom hash function such that clauses containing the same elements have the same hash, and thus I could use an unordered set for the clause sets. This would improve the subset calculation performance from O(n^2) to O(n) where n is the size of the larger clause set. This hash value would also allow for O(1) equality comparisons between clauses.

**k-CNF generator**

My k-CNF generator guarantees that there are no duplicate variables in one clause and that all variables are used at least once in the sentence. The generator creates a text file in the mini-SAT format. It takes in parameters k, m ,n. (k = # literals per clause, m = # clauses, n = # symbols)

**Correctness**

MiniSAT contains statistics about the number of variables and clauses. I intentionally generated edge examples like k = n to verify that there are no issues.