Fast Implementations of WalkSAT and Resolution Proving

CMPT 310, Spring, 2020; Tongzhou Shen; 2020.04.04

Abstract:

This is a faster implementation of WalkSAT and theorem proving by resolution. Compared to original AIMA WalkSAT code, a Python to C++ translation resulted in a 46% average run time reduction. Further optimizations of the C++ WalkSAT in data structure and algorithm resulted in an addition 10 fold increase in speed.

Program Structure:

- python (folder):
  - rand_cnf_generator.py: Generates a random CNF clause and outputs in miniSAT input format.
  - resolution_prover.py: re-implementation of AIMA pl_resolution() with simplifications.
  - walk_sat.py: re-implementation of AIMA walk_sat() with simplifications.
- cpp (folder)
  - walk_sat (folder): WalkSAT implemented in C++.
  - new_walk_sat (folder): An optimized implementation of WalkSAT in C++.
  - resolution_proving (folder): Resolution Proving implemented in C++.

Results can be reproduced using usages in the README.txt of the respective folder.

Data Structure and Algorithm Optimizations:

Data structure: All Clauses: map<string, set<int>>; Unsatisfied Clauses: set<string>

1. Computation of unsatisfied clauses optimized.

Originally, every time [func] check_assignment() is called after an assignment change, the function checks all clauses again and generates a new vector of unsatisfied clauses. “The definition of insanity is doing the same thing over and over and expecting different results.” This is inefficient as we are expecting different results in clauses that doesn’t contain the variable that we have flipped.

The optimized procedure checks if the clause contains the changed variable first. If so, insert the clause into or erase it from the unsatisfied clause set based on the updated model. Checking if an order_set contains an element is a O(logN) operation, an improvement for checking the clause, a O(n) operation.

This erase procedure requires a more efficient data structure than vector, to reduce O(n) deletion to O(logN)[1] deletion. Since vector<int>, our current representation of a clause is unhashable, and incomparable for the liking of a set. vector<int> is encoded into a string, which gives us both appealing properties.

2. Improvement in data structure holding all clauses.

In the original implementation, conjuncted clause were stored in a vector. This is not flexible. Since unsatisfied clauses is represented as strings, fast lookup O(logN) from string to set<int> representation is required. Clause’s data structure is changed from vector<vector<int>> to map<string, set<int>>.

3. Reducing overhead time in determining the variable to flip

Which variable to flip is decided by the highest number of satisfied clauses the flip results in. Since the situation is hypothetical, instead of modifying unsatisfied clauses set and reverting the change, we can receive a marginal performance gain by keeping a counter on the increase/decrease of unsat clause count.

4. **Pure Symbol Heuristic [Unused]:**

k = # of literals per clause; m = # of clauses; n = # of symbols

The probability of one symbol being pure symbol is 2 * (1/2)^(k*m/n). In practice, a k=3, m=5000, n=2000 randomly generated CNF only has 82 pure symbols. It has a negligible 5% reduction in the size of symbol set. For problems with k*m/n < 2, pure symbol heuristic would not be useful.

5. Hyperparameter optimization

Tuning p in walkSAT is a example of exploration vs exploitation. Higher the p, more likely it is for the algorithm to flip a random variable, instead of calculating flip to minimize the number of clauses unsatisfied. In a problem with Python, highly randomized flipping is solved faster. However, the same pattern is not observed in C++, where minimum time is achieved with a p of 0.5. Since there are more overhead with calculation of best variable to flip, one possible explanation of Python’s abysmal performance with low p is that it is unable to compute the best variable efficently.

Result:

p/time	Python Time(s)	C++ Optimized Time (s)		Problem/ Runtime	miniSAT(s)	Python WalkSAT (s)	C++ WalkSAT (s)	C++ WalkSAT Optimized (s)
0.25	51.4157	0.944		4-queens	0.0044	0.0309	0.019	0.007
0.5	23.9643	0.871		k-3 m=7500 n=5000	0.0616	23.96	10.82	0.900
0.75	12.3012	0.886		k-3 m=10000 n=5000	0.0806	67.81	31.38	2.531
0.9	8.99349	1.054		k-3 m=12500 n=5000	0.0700	129.10	94.89	11.562
0.95	7.97184	0.983						

Conclusion: Pseudocode can sometimes be optimized to run orders of magnitude faster through using an efficient language and applying various techniques in algorithm and data structure to reduce redundancy.

1. Although unordered_set/map (hashtable) allows O(1) deletions and insertions, they are not chosen due to the time complexity of picking a random element from the set, a common operation in walkSAT.