Fast Implementations of WalkSAT and Resolution Proving

By : Shresth Kapila (301355384)
& Harnoor Singh (301355738)

CMPT 310 : Artificial Intelligence Survey

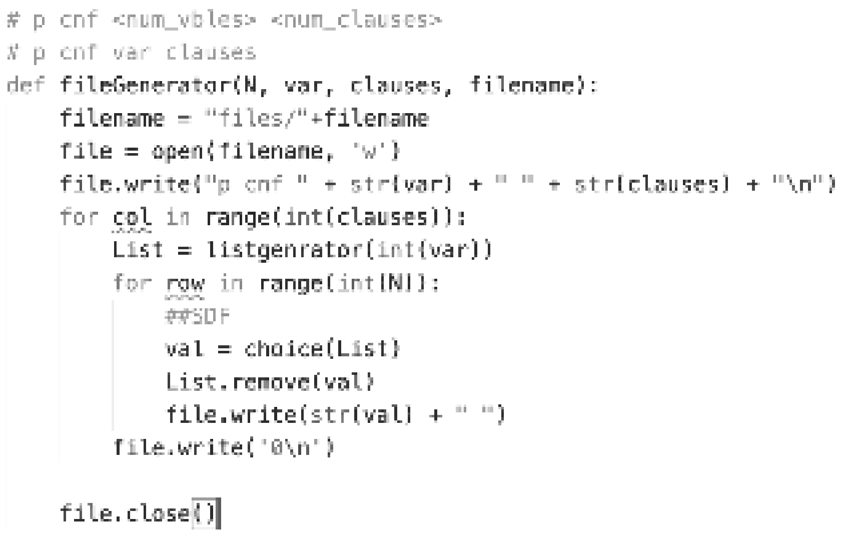
Professor Toby Donaldson

SCHOOL OF COMPUTING SCIENCE

Summer 2020

Generating K-CNF

K-cnf expressions are generated using python. The implementation is in filegenerator.py, it takes the name of the file, number of literals, symbols and clauses. The function fileGenerator in filegenerator.py makes a new file with the given name in the files folder.

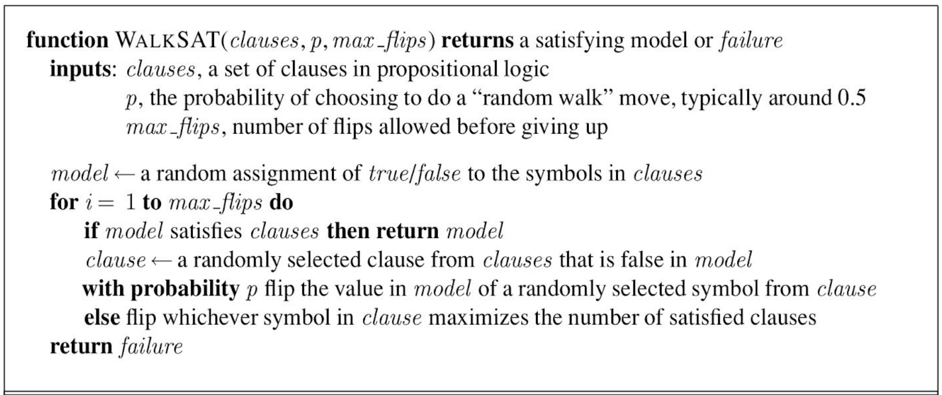


```python
# p cnf <num_vbles> <num_clauses>
# p cnf var clauses
def fileGenerator(N, var, clauses, filename):
    filename = "files/"+filename
    file = open(filename, 'w')
    file.write("p cnf " + str(var) + " " + str(clauses) + "\n")
    for col in range(int(clauses)):
        List = listgenrator(int(var))
        for row in range(int(N)):
            ##SDF
            val = choice(List)
            List.remove(val)
            file.write(str(val) + " ")
        file.write('0\n')

    file.close()
```

WalkSAT

Data Structures and Algorithms

WalkSat is a type of local search algorithm used to solve satisfiability problems. It takes an unsatisfied statement, then picks a symbol from that clause and flips it. There are two ways to pick the symbol to be flipped. First is to choose it randomly, which really is inefficient because the algorithm might end up flipping the same variable multiple times. Second way is the min_conflict way which actually minimizes the total number of unsatisfied clauses in each run. The algorithm in the figure 7.18 from the book Artificial Intelligence: A Modern Approach (3rd Edition) by Stuart Russell, Peter Norvig just repeatedly checks all the clauses and then generates new unsatisfied clauses each time, resulting in no track of flipped symbol.



```
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.

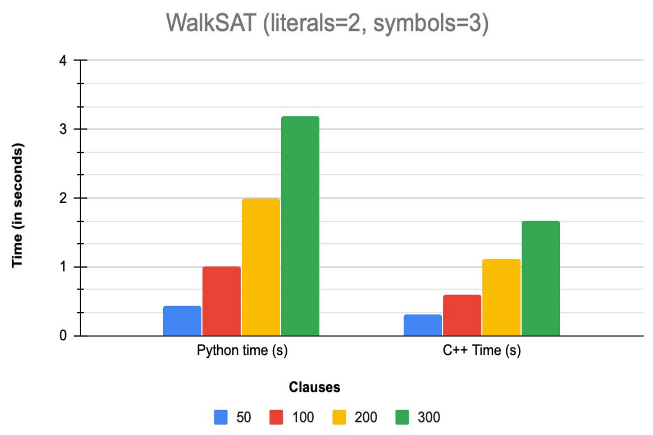
A solution to that was if we could somehow track and check the flipped symbol in the list of unsatisfied clauses. The optimised version of walksat first checks if the chosen clause contains the flipped symbol, if it does then just insert the clause into the list of unsatisfied, else make a new model after removing it from unsatisfied clauses.
In total two implementations of walkSAT are there, first one is almost similar to the AIMA
We used C++ to implement the algorithm. Major complex operations are inserting , removing, and choosing the symbols.All the clauses are saved in a map of string and vector of integers, unsatisfied clauses are saved in a set of strings. Generally, map.erase() is a `O(log n)` complexity operation, which is better than using a vector or array which are `O(n)` operations.

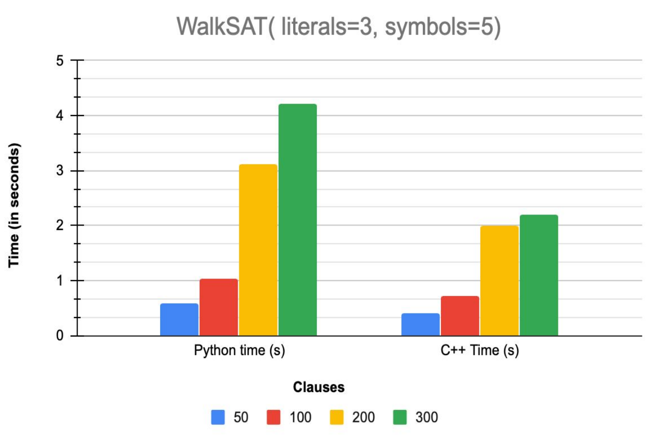
Runtime

Initially for the small number of clauses the C++ implementation was not doing alot better then the python implementation but as the number of clauses increased the running time for C++ implementation got a lot better. Approximately, there is 44% reduction in running time for clauses equal to 200 and 300.

WalkSAT - Python VS C++ Runtime					
Literals = 2 Symbols = 4			Literals = 3 Symbols = 5		
Clauses	Python time (s)	C++ Time (s)	Clauses	Python time (s)	C++ Time (s)
50	0.4264	0.3054	50	0.5825	0.3972
100	1.0126	0.5881	100	1.0297	0.7195
200	1.9986	1.1121	200	3.1147	1.9874
300	3.1926	1.6767	300	4.2045	2.2042

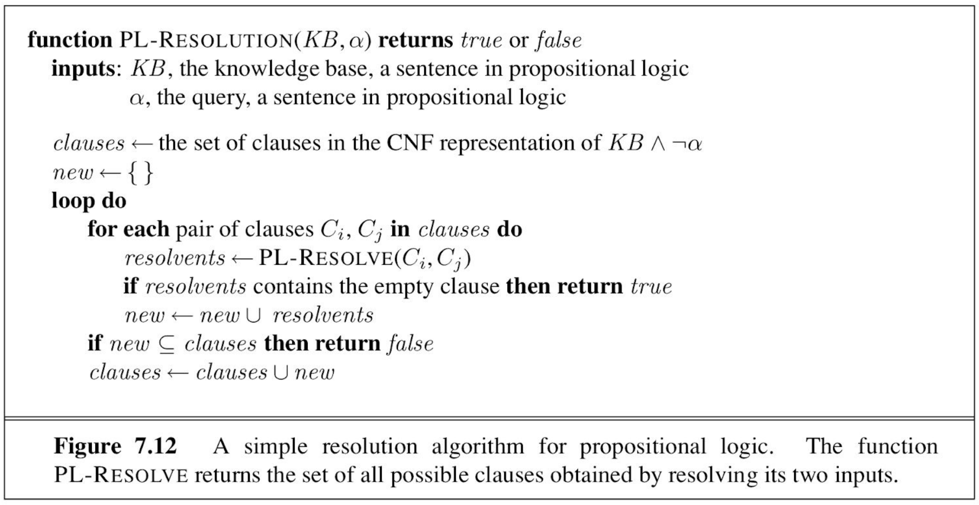
Considering all of the tested files, C++ implementation showed about 38% reduction in runtime as compared to the python implementation of walkSat. Which is quite better as there was not a big difference in algorithm used, it is quite interesting how C++ is able to make the almost similar algorithm run that fast.





Resolution Proving

Taking a look at the algorithm from the algorithm in the figure 7.12 from the book Artificial Intelligence: A Modern Approach (3rd Edition) by Stuart Russell, Peter Norvig which shows the proof by contradiction that is (KB ^ ¬***α***) is unsatisfiable. This returns all the possible clauses that are solved by adding to the set that are not present. This converts the (KB ^ ¬***α***) to CNF.

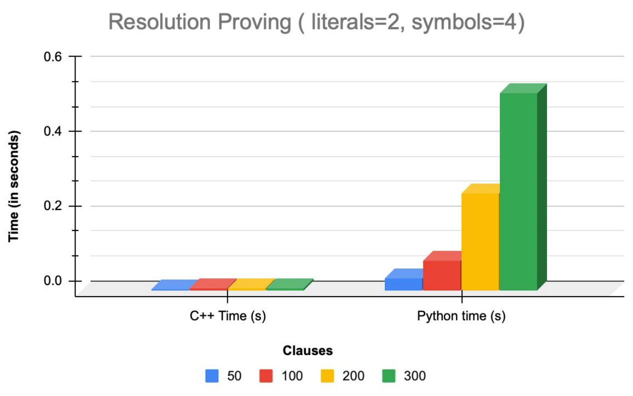


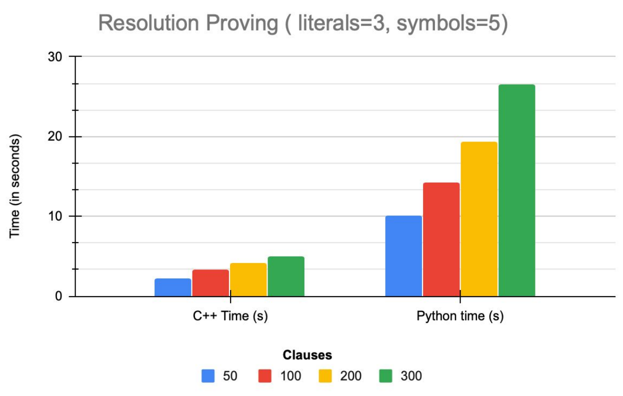
```
function PL-RESOLUTION($KB, \alpha$) returns $true$ or $false$
  inputs: $KB$, the knowledge base, a sentence in propositional logic
          $\alpha$, the query, a sentence in propositional logic

  $clauses \leftarrow$ the set of clauses in the CNF representation of $KB \wedge \neg\alpha$
  $new \leftarrow \{\}$
  loop do
     for each pair of clauses $C_i, C_j$ in $clauses$ do
        $resolvents \leftarrow$ PL-RESOLVE($C_i, C_j$)
        if $resolvents$ contains the empty clause then return $true$
        $new \leftarrow new \cup resolvents$
     if $new \subseteq clauses$ then return $false$
     $clauses \leftarrow clauses \cup new$
```

Figure 7.12 A simple resolution algorithm for propositional logic. The function PL-RESOLVE returns the set of all possible clauses obtained by resolving its two inputs.

Compared to the pl_resolution function in logic.py of the aima-python which contains the function pl_resolve that uses the method of disjuncts consuming more time to resolve clauses. Therefore, instead, in our fast implemented algorithm we tried using the map<string, set<int>> to reduce the time for consuming. Also, implementing the same algorithm in c++ instead of python made a huge difference in the running time (as below).

Resolution Proving - Python VS C++ Runtime					
Literals = 2 Symbols = 4			Literals = 3 Symbols = 5		
Clauses	C++ Time (s)	Python Time (s)	Clauses	C++ Time (s)	Python Time (s)
50	0.0043	0.0342	50	2.2201	10.1344
100	0.0049	0.0797	100	3.3394	14.2336
200	0.0052	0.2597	200	4.1038	19.3998
300	0.0061	0.5296	300	4.9946	26.551


Resolution Proving (literals=2, symbols=4)
Time (in seconds)
C++ Time (s)
Python time (s)
Clauses
🟦 50
🟥 100
🟨 200
🟩 300


Resolution Proving (literals=3, symbols=5)
Time (in seconds)
C++ Time (s)
Python time (s)
Clauses
🟦 50
🟥 100
🟨 200
🟩 300