CS463 Program 3: SAT Solvers

Implementor's Notes
Peer X, Peer Y
CS463G
10/16/2024

GENERAL APPROACH

The very first step of this project was to figure out what the heck the .CNF files were, how they were formatted, and how to use them. This was done using a class that would be used to represent the formulas `(class CNFFormula)`, and the only purpose for this class was to be able to parse through each line of the .CNF files and get important information such as the number of variables, number of clauses, a list to hold all clauses, and a set to hold variables. Then it is run through one of the three solvers:

DPLL algorithm's basic premise is to start with an empty assignment of truth values to the CNF formula and identify pure literals and unit clauses to eliminate them from the clauses that contain them. This is the heuristic for DPLL. Then it chooses an unassigned variable and assigns a TRUE value, this is recursively called to assign TRUE to the next unassigned variable until the very last unassigned variable. If a satisfying assignment is not found, assign the current variable (latest assigned variable) to FALSE. If it is still not satisfied, backtrack to the previous variable and do DPLL again. The program contains a lot of checks such as functions `_is_consistent()` and `_is_clause_falsified()` to check each clause and verify that there are no conflicts with each step of the process. I've included statistics for DPLL including 'decisions', 'backtracks', and 'unit_propagations.' This effectively shows the amount of work that is being done with each formula. If the algorithm satisfies all clauses, it will return True, otherwise return False, signaling that there is no satisfying assignment. A sample output looks like:

```
Formula satisfied
Statistics:      {'decisions':
41024,     'unit_propagations':
1182491, 'backtracks': 41012}
Time  elapsed  for  uf75-02.cnf:
141.0605 seconds
```

WalkSAT algorithm is a local search but it factors in probability. It combines the greedy search nature of GSAT (that will be explained more later) and randomization through a declared variable probability, p. It starts with a random truth value assignment for all variables with an initialized value for maximum number of flips, max_flips. At each flip until max_flips, select an unsatisfied clause and choose with a probability p = 0.5, whether to perform a flip that results in the fewest unsatisfied clause (greedy) or randomly flip a selected variable (random). This is done until either a satisfying assignment is found, where all clauses are satisfied, OR when max_flips is reached and no solution is found. The greedy flip temporarily flips the selected variable and checks how many clauses would become satisfied and flips the variable that would result in the most number of satisfied clauses. The sample output for WalkSAT looks like:

```
Formula satisfied
Statistics:     {'flips':   20,
'satisfied_clauses': 91}
Time elapsed for uf20-0158.cnf:
0.0047 seconds
```

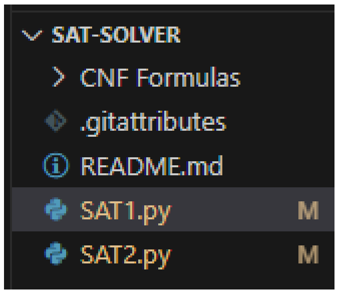
When flips reach a maximum number of flips (set to 1000), 99.9% of the time, it means that the formula is not satisfied.

The GSAT algorithm, where the "G" is short for "greedy," does not care about being extensive or thorough in its search. Instead, it acts exactly how its name suggests and tries to find the quick and easy path to success. Our implementation of GSAT works by generating a random true/false assignment to every variable in the assignment, and flipping variables individually to see if an optima can be found. More often than not the GSAT finds a local optima and gets stuck, that is why we have allowed it 10 resets and 100 flips per reset to allow it to explore while staying reasonable to computational and time restraints.

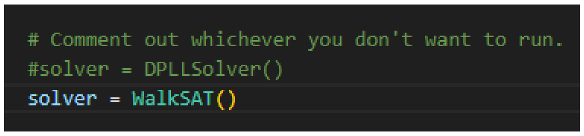
At the lower level, one will find that our GSAT algorithm begins by randomizing the truth table of the variables. That is, $x_1 : x_n$ are assigned true or false. From there, the algorithm will iterate through each instance x, in the process flipping it and calculating the resulting fitness. Note that the fitness is calculated by how many clauses of the overall assignment return true. If the fitness with the new flipped bit is higher than the fitness before the change, then we store that variable as the new threshold and note the new minimum fitness to achieve. This process happens for 100 flips, and at the end of 100 flips the program checks to see if the optima it found on this runthrough of GSAT beats the previous optimas. Once the program reaches 100 flips it restarts, with 10 restarts being determined by the project parameters.

RUNNING THE CODE

We were given a folder of formulas called `CNF Formulas` that contain a bunch of .CNF files. So we directly reference this folder in the code, meaning the folder needs to be in the same workspace. `SAT1.py` contains the solvers for DPLL and WalkSAT, while `SAT2.py` has GSAT. It should look something like this for file directory:



In `SAT1.py`, it is more convenient to run one solver at a time.



```
# Comment out whichever you don't want to run.
#solver = DPLLSolver()
solver = WalkSAT()
```

So just choose which one you want to run. SAT1.py runs all the files in sequential order.

In SAT2.py, just run the file like normal. You can specify what file you want to run in main.

If you are given the error "Error: File xyz is not found," then make sure that your directory is within the folder SAT-Solver, as our code relies on the user to have that as their directory to successfully extract the files to run the algorithm on.

ISSUES

One of the main concerns that first appeared was what .CNF files were and how they were used to check for clause satisfiability. Because of this, the data structure that was decided on majorly depended on our understanding. For example, we decided to store clauses into lists as ints and variables as sets. Then using a class to turn those clauses and variables into a usable formula.

Next was the implementation of recursion for DPLL. Recursive things have always been difficult to understand and so for DPLL, finding the correct way to implement the algorithm was challenging due to all the constant checks. The code stores clauses as lists of literals. While this works, searching through lists repeatedly can be inefficient for large CNF formulas like we see in the graphs. Additionally, the recursion can lead to depth limit on larger files going up to like 200K backtracks which takes up a LOT of memory and heats up my computer like crazy.

FINDINGS

WalkSAT, overall, took less than a minute to run all formulas in CNF Formulas, as opposed to DPLL. However, the more difficult formulas simply run out of flips without finding a satisfying assignment.

Increasing max_flips = 100,000 resulted in some of the easier ones to always solve, taking more than 10,000 flips sometimes, while the more complicated ones still do not solve.

GSAT was very quick, but had terrible odds of finding the correct solution for files that had large numbers of clauses. While it was outside of the scope of the data collection for this project, we tried increasing the number of flips to 1000 and the number of resets to 300 and found that those new, albeit extremely expensive, parameters drastically increased the odds that a solution was found for even the most complex files. Therefore, our conclusion for GSAT is that it is moderately trustworthy for simple SAT problems given enough flips and resets.

Finally, DPLL took about 9 hours and 30 minutes to fully compile and solve all the formulas. Longer, more complex formulas like uf75-06 took an hour and a half to complete producing over 2 million backtracks and 2 million decisions made while some took less than 10 seconds.

LEARNING OUTCOME

Throughout this project, the two collaborators gained first time experience working with boolean satisfiability problems, with which our SAT algorithms solved them. Once we had a decent understanding of them, only then were we able to break into coding the project. Furthermore this project was a great opportunity to get better at algorithms. We both feel that this class is a spiritual successor to CS315, Algorithm Design and Analysis, and that this project (along with the Pyraminx project) have bettered our skill at implementing and debugging algorithms.

As far as technical skills learned, we both had to get a basic understanding of how to parse files, which neither of us have done since the shell project of CS270.

RESULTS AND GRAPHS

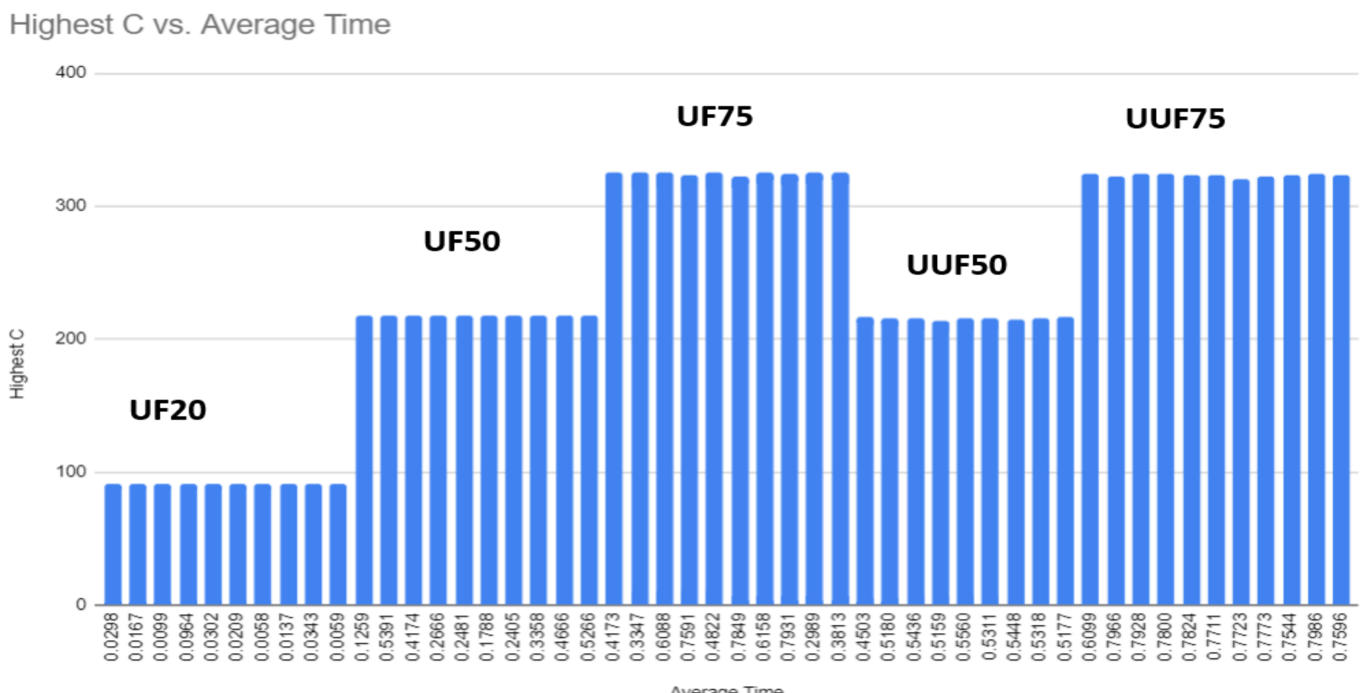
Below are the compiled results for our WalkSAT and GSAT algorithm trials. Note that we do not have the fitness for the DPLL algorithm because it either solves or fails.

WalkSAT Results

File	Average Time	Fitness	Solved (total clauses)
uf20-0156	0.0298	91	Yes
uf20-0157	0.0167	91	Yes
uf20-0158	0.0099	91	Yes
uf20-0159	0.0964	91	Yes
uf20-0160	0.0302	91	Yes
uf20-0161	0.0209	91	Yes
uf20-0162	0.0058	91	Yes
uf20-0163	0.0137	91	Yes
uf20-0164	0.0343	91	Yes
uf20-0165	0.0059	91	Yes
uf50-01	0.1259	218	Yes
uf50-02	0.5391	218	Yes
uf50-03	0.4174	218	Yes
uf50-04	0.2666	218	Yes
uf50-05	0.2481	218	Yes
uf50-06	0.1788	218	Yes
uf50-07	0.2405	218	Yes
uf50-08	0.3358	218	Yes
uf50-09	0.4666	218	Yes
uf50-10	0.5266	218	Yes
uf75-01	0.4173	325	Yes
uf75-02	0.3347	325	Yes
uf75-03	0.6088	325	Yes
uf75-04	0.7591	323	No
uf75-05	0.4822	325	Yes
uf75-06	0.7849	322	No
uf75-07	0.6158	325	Yes
uf75-08	0.7931	324	No

uf75-09	0.2989	325	Yes
uf75-10	0.3813	325	yes
uuf50-01	0.4503	217	No
uuf50-02	0.5180	216	No
uuf50-03	0.5436	216	No
uuf50-04	0.5159	214	No
uuf50-05	0.5560	216	No
uuf50-06	0.5311	216	No
uuf50-07	0.5448	215	No
uuf50-08	0.5318	216	No
uuf50-09	0.5177	217	No
uuf50-10	0.6099	324	No
uuf75-01	0.7966	322	No
uuf75-02	0.7928	324	No
uuf75-03	0.7800	324	No
uuf75-04	0.7824	323	No
uuf75-05	0.7711	323	No
uuf75-06	0.7723	320	No
uuf75-07	0.7773	322	No
uuf75-08	0.7544	323	No
uuf75-09	0.7986	324	No
uuf75-10	0.7596	323	No

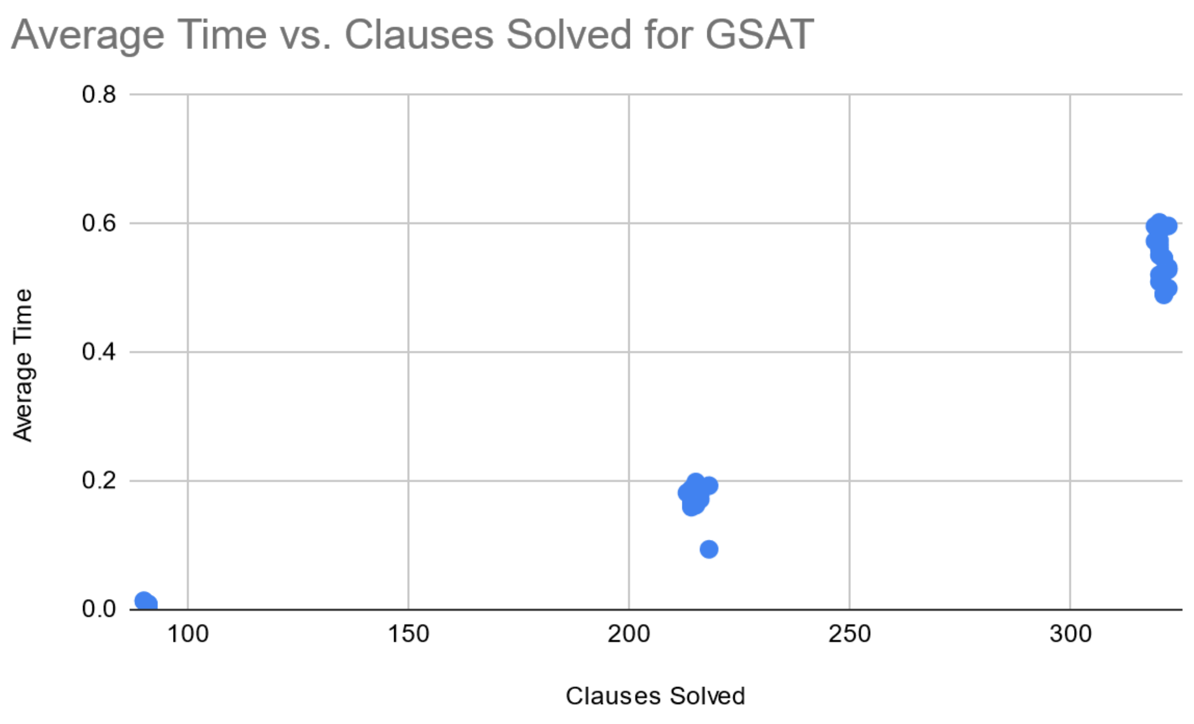
This is a graph for the highest C value versus Average Time for WalkSAT, where max_flips = 1000:



GSAT Results

File	Average Time	Fitness	Solved?
uf20-0156	0.005514860153	91	Yes
uf20-0157	0.008158922195	91	Yes
uf20-0158	0.00600028038	91	Yes
uf20-0159	0.01401472092	90	No
uf20-0160	0.01299762726	90	No
uf20-0161	0.009016752243	91	Yes
uf20-0162	0.001026153564	91	Yes
uf20-0163	0.002026557922	91	Yes
uf20-0164	0.004018545151	91	Yes
uf20-0165	0.004243850708	91	Yes
uf50-01	0.1752252579	215	No
uf50-02	0.1725642681	214	No
uf50-03	0.1665530205	214	No
uf50-04	0.165324688	215	No
uf50-05	0.192961216	218	Yes
uf50-06	0.1776211262	215	No
uf50-07	0.1670403481	214	No
uf50-08	0.09396839142	218	Yes
uf50-09	0.1788415909	216	No
uf50-010	0.1625723839	215	No
uf75-01	0.4896273613	321	No
uf75-02	0.6023201942	320	No
uf75-03	0.5747721195	320	No
uf75-04	0.573236227	319	No
uf75-05	0.5963902473	319	No
uf75-06	0.5511209965	320	No
uf75-07	0.5099170208	320	No
uf75-08	0.5951418877	321	No
uf75-09	0.5968527794	322	No
uf75-010	0.5470912457	321	No
uuf50-01	0.1818163395	213	No
uuf50-02	0.1724188328	214	No
uuf50-03	0.1715238094	216	No
uuf50-04	0.1594865322	214	No

uuf50-05	0.1653048992	215	No
uuf50-06	0.1789243221	215	No
uuf50-07	0.193567276	215	No
uuf50-08	0.187867371	214	No
uuf50-09	0.1985992828	215	No
uuf50-10	0.1878598832	216	No
uuf75-01	0.5286400318	322	No
uuf75-02	0.5602674484	320	No
uuf75-03	0.4919271469	321	No
uuf75-04	0.59223032	320	No
uuf75-05	0.5672245026	320	No
uuf75-06	0.4998829986	322	No
uuf75-07	0.5450993902	321	No
uuf75-08	0.532005992	322	No
uuf75-09	0.520912949	320	No
uuf75-10	0.523124321	321	No



```
Formula satisfied
Assignment:  {1: True, 2: True, 3: False, 4: False, 5:
False, 6: True, 7: True, 8: True, 9: False, 10: False, 11:
False, 12: False, 13: True, 14: True, 15: True, 50: True,
33: True, 60: True, 39: True, 57: False, 70: False, 51:
True, 44: False, 16: True, 17: False, 67: True, 18: False,
19: False, 20: True, 71: False, 29: False, 73: True, 53:
False, 25: True, 34: True, 61: True, 66: False, 69: False,
68: False, 56: True, 47: False, 24: False, 49: False, 59:
True, 46: True, 64: True, 37: True, 35: False, 72: True,
65: False, 27: True, 31: False, 21: True, 45: False, 36:
True, 23: False, 41: True, 54: False, 48: False, 40: False,
43: True, 42: True, 22: True, 62: True, 38: True, 52:
False, 58: False, 74: True, 26: True, 28: True, 30: True,
32: True, 55: True, 63: False, 75: False}
1: True 2: True 3: False 4: False 5: False 6: True 7: True
8: True 9: False 10: False 11: False 12: False 13: True 14:
True 15: True 16: True 17: False 18: False 19: False 20:
True 21: True 22: True 23: False 24: False 25: True 26:
True 27: True 28: True 29: False 30: True 31: False 32:
True 33: True 34: True 35: False 36: True 37: True 38: True
39: True 40: False 41: True 42: True 43: True 44: False 45:
False 46: True 47: False 48: False 49: False 50: True 51:
True 52: False 53: False 54: False 55: True 56: True 57:
False 58: False 59: True 60: True 61: True 62: True 63:
False 64: True 65: False 66: False 67: True 68: False 69:
False 70: False 71: False 72: True 73: True 74: True 75:
False
Statistics:  {'decisions': 2630111, 'unit_propagations':
64647456, 'backtracks': 2630097}
Time elapsed for uf75-06.cnf: 5663.8009 seconds
```