PA5, Kunal Jha, CS76 2021 Fall

Description

All of my functions that I designed were in the *SAT* class. I began by creating a method to parse the cnf files and interpret the clauses. It opened the inputted *.cnf* file, iterated through each line and removed all whitespace to create a list of clauses. It then iterated through these clauses and added them to a collection of variables and clauses, making sure to factor in whether the clause had a negative sign before it.

My *gsat* method relied on a recursive call of a *gsatHelper* function. The helper function incremented the number of states visited each time a call to the helper function was made. It then checked to see whether the goal state was reached, and returned the model if the goal was found. Otherwise, it generated a random number between 0 and 1, and flipped a random bit if the number generated was below a threshold (by default this threshold was 0.3). If it flipped a random bit, it then made another call to the helper function and the process was repeated. Otherwise, it determined what the best bit to flip was by iterating through each variable in the model, creating a copy of the model, flipping the variable, then making a call to the *countFulfilled* method, which determined how many of the clauses were fulfilled. It then added this score to a dictionary which mapped the result of countFulfilled to a list of variables that had that score when flipped. After this process was completed, the key with the greatest value in the scores dictionary was determined, and a random variable from this key's value was selected. The variable was then flipped in the actual model, and a recursive call to the *gsatHelper* was made. The actual *gsat* method simply initialized a model (either a pre-selected model or randomly generated one), called the helper function to return the solution, and converted the solution to cnf format using a function described below.

I also had a method to get a random model, which simply iterated through all variables, generated a random number between 0 and 1, flipped a variable if the random number was below 0.5, and added all of the variables (including the flipped one), to a set which was returned. My countFulfilled method simply returned the number of clauses disjoint with the inputted model. The goal test method worked similarly by return False if any of the clauses were disjoint with the model. The solution to cnf method iterated through all variables in the solution and added them to a result list, making sure to include a negative sign if the variable was not equal to its absolute value. Write solution took in this cnf form, wrote the result on the first line, and wrote each bit in the result on a new line.

walk_sat began by generating a random model and iterating for as many flips as a predetermined parameter. For each iteration, it increments the states visited by 1, and returns a solution if found. Otherwise, it picks a random clause from the unsolved clause set, and generates a random number. If the random number is less than some threshold (by default 0.3), a random choice from the selected clause is flipped and the model is updated. Otherwise, scores are determined similar to how they were in the *gsatHelper* method and the bit with the max score is flipped. This process is repeated until a solution is found, and if no solution is found, False is returned.

solve_sudoku.py was made by taking the user's input about which puzzle they'd like to solve and calling either walk_sat or gsat on the cnf file corresponding to the puzzle name. If a solution was found, the solution was written to a .sol file and displayed. Otherwise, the user was informed that no solution was found. In both cases, information about the run time and number of states visited was displayed, with unsolved clauses being displayed if a solution was not found.

Evaluation

Both my *gsat* and *walk_sat* methods worked as expected. Gsat was able to solve some of the smaller test cases, such as only solving rows or a single cell. walk_sat was able to solve all cases except for puzzle_bonus. I noticed that when the random threshold was lower, gsat took longer to find a solution, as did walk_sat. This makes sense because they had to go through more computation steps rather than simply selecting a random bit to flip. Below, I've detailed performance for GSAT and WalkSAT for various test cases:

```
Puzzle: test.cnf
P-value: 0.3
```

```
WalkSAT performance:
Solved after visiting 3 states
Solved in: 0.0035707950592041016 seconds

GSAT performance:
Solved after visiting 12 states
Solved in: 0.0018248558044433594 seconds
```

```
Puzzle: rows.cnf
P-value: 0.35

WalkSAT performance:
Solved after visiting 411 states
Solved in: 0.523097038269043 seconds

GSAT performance:
Solved after visiting 813 states
Solved in: 136.09656500816345 seconds
```

```
Puzzle: rows_and_cols.cnf
P-value: 0.25

WalkSAT performance:
Solved after visiting 1589 states
Solved in: 2.655709981918335 seconds

GSAT performance:
Did not converge in a reasonable amount of time
```

```
Puzzle: puzzle1.cnf
WalkSAT performance:
P-value: 0.2
No solution found after visiting 100000 states
Run time: 143.73558712005615 seconds
Clauses left unsolved: 1

P-value: 0.3
Solved after visiting 3986 states
Solved in: 7.017216682434082 seconds

P-value: 0.4
Solved after visiting 4851 states
Solved in: 8.178408145904541 seconds
```

```
Puzzle: puzzle2.cnf
WalkSAT performance:
P-value: 0.2
Solved after visiting 21880 states
Solved in: 41.056432008743286 seconds

P-value: 0.3
Solved after visiting 7214 states
Solved in: 15.677613019943237 seconds

P-value: 0.4
Solved after visiting 26299 states
Solved in: 51.55216884613037 seconds
```

It seems the optimal p-value is around 0.3, as the algorithm may balance random action with best actions well at this probability.