report.md

Logic

Assignment 5
CS 276, Artificial Intelligence
Fall 2021, Dartmouth College
Spencer Bertsch

This report contains two sections:

1. Description
2. Evaluation

Description

1. How do your implemented algorithms work?

Understanding Propositional Logic & .cnf Files

In order to understand how to solve this problem, we first need to understand the **.cnf** files provided for us to solve. The cnf (conjunctive normal form) files are filled with clauses representing the rules of sudoku. By taking an initial board and iterating through all the clauses in a **.cnf** file, we can test the board to find how many unsatisfied clauses there are. When the number of unsatisfied clauses reaches zero, then the search is finished, and we can return the current assignment.

In our case, we are working with boolean values that are always separated by an 'or' ($\vee$) symbol. In this way, all we need to do is loop through our current assignment and compare each bool in each clause to the corresponding element of the assignment. If at least one of the assignment values aligns with the clause, then that clause is satisfied, and we can continue. If no values in the assignment align with any values in the clause, then that clause is unsatisfied.

For example, a clause might be:

`~111 ~112` , or `~111 ∨ ~112`

This represents that the value `11` , or row[1], column[1] cannot have the values of both 1 and 2. Assignments such as `{111: True, 112: False}` , or `{111: False, 112: False}` would satisfy the clause. However, the assignment `{111: True, 112: True}` would *not* satisfy the clause because neither value in the assignment matches the truth values listed in the clause.

Additional information on the formulation of the sudoku problem as a satisfiability problem in conjunctive normal form can be found from Ivor Spence here.

GSAT

GSAT is a search algorithm that is designed to solve satisfiability problems given a set of boolean logical expressions. In our case, the boolean logic expressions are in the form of **.cnf** files which we iterate through, altering the assignment of a sudoku board until all clauses are satisfied. The steps of the GSAT algorithm are as follows:

- The below description of the GSAT algorithm was adapted from the description on Gradescope, with some small additions that I thought were helpful important.

1. Generate a random assignment while preserving the initial state of the board, if one was provided.
2. Check to see if the current assignment satisfies all the clauses in the **.cnf** file. If it does, then stop the search and return the solution assignment.

3. Generate a random value between 0 and 1. If the value is greater than a predetermined threshold (*p*), then choose one value in the assignment and flip its truth value from False to True, or True to False. After this, go back to Step 2.
4. If the random value is below (*p*), then iterate through all values in the assignment and find the number of clauses that would be satisfied if the assignment's truth value was flipped.
5. Out of all the variables that had the highest impact on the number of satisfied clauses when flipped, select a random variable and flip its truth value. Go back to Step 2.

The GSAT algorithm takes quite a while to find solutions, even for small problems. I noticed that the GSAT algorithm performed poorly when the *p* value was too low (below 0.5), and performed much better when the *p* value was high (above 0.95). I can attribute this to the fact that the GSAT algorithm explores randomly when the random value is above *p*, which towards the end of the search has a much higher probability of creating additional unsatisfied clauses than satisfying additional clauses by chance.

WalkSAT

WalkSAT is very similar to the GSAT algorithm, except with a few key differences that significantly increase performance and reduce run time.

The main difference that accounts for the improvement over GSAT is that when WalkSAT searches randomly, it only flips variables that we know are causing at lease one clause to be unsatisfied. This means that every time is randomly makes a flip, we know that the resulting assignment should not be getting worse. The WalkSAT flip might cause as many problems as it solves, but it doesn't randomly flip variables in the way that GSAT does.

The runtime boost is due to the fact that WalkSAT only iterates through the 'problem' variables instead of all variables. By 'problem' variables, I mean the variables associated with an unsatisfied clause. Even better, we are selecting one unsatisfied clause uniformly at random, then finding all of the variables associated with that clause, then scoring them and flipping the optimal one. When the search gets close to a solution, then there are only a few unsatisfied clauses and only a handful of problem variables which means that every iteration of the WalkSAT algorithm is extremely fast. By contrast, GSAT iterations occur at the same speed if there are 3,000 unsatisfied clauses or if there are 2 unsatisfied clauses.

What design decisions did you make? How you laid out the problems?

All GSAT and WalkSAT code is in the SAT.py file, and there is a driver function in the run_sat.py file that can be used to easily test either of the search algorithms on any of the puzzle files in the /puzzle directory. There is a README in this directory that shows how to run the run_sat.py file to test each algorithm on different .cnf files. I also consolidated the puzzle files into a new directory called */puzzles*, and I store all of the solutions in a directory called */solutions*.

- A few notes on implementation of GSAT and WalkSAT:

I initially coded the solver using a direct assignment as seen below.

```
assignment: dict = {
    111: False,
    112: True,
    113: False,
    ...
}
```

However, this forces the solver to only work for this specific Sudoku problem. In order to fix this problem, I implemented an *encode/decode* dictionary which is used as a map between the assignment values and the cells in the sudoku board. This *encode/decode* mapper is what allowed me to also implement the map coloring problem using GSAT and WalkSAT.

```
assignment_encode: dict = {
    111: 1,
    112: 2,
    113: 3,
    ...
}
```

```
assignment_decode: dict = {
    1: False,
    2: True,
    3: False,
    ...
}
```

This seemingly pedantic step is what allows the solver to become dynamic and applicable to any problem, not just the sudoku problems in conjunctive normal form. Using the new solver, I was able to also solve the map coloring problem. This solver could solve any constraint satisfaction problem with relatively few variables in the assignment, and relatively few clauses given the .cnf file filled with the logical statements.

Evaluation

1. Do your implemented algorithms actually work? How well?

The implemented SAT solver works exactly as expected on all puzzles, including the puzzle1.cnf and puzzle2.cnf files.

	one_cell	all_cells	rows	rows_and_cols	rules	puzzle1	puzzle2
GSAT Iterations	5	306	378	Unsolved after 2k	Unsolved after 2k	Unsolved after 2k	Unsolved after 2k
WalkSAT Iterations	5	282	346	1,468	3,354	6,784	59,012

Table 1: Results of the GSAT and WalkSAT local search algorithms on different .cnf puzzle files.

I will say that unlike the solution that the professor posted on slack, I had to vary my p value through a large range of values before I was able to get my WalkSAT to find solutions to puzzle1 and puzzle2. I ended up using p=0.85 for puzzle1 and p=0.8 for puzzle2. Using these p values along with a value of p=0.9 for all other puzzles, I was able to achieve the results in *Table 1*.

Lastly, using a combination of the provided code and a small method that writes the resulting assignment after the puzzle is solved, we get to see the solved puzzles! Below we can see a solution to one of the sample problems: puzzle2.cnf

Puzzle 2 Solution:

```
4 3 1 | 6 9 7 | 5 2 8
5 7 2 | 1 3 8 | 9 6 4
6 9 8 | 5 2 4 | 3 1 7
---------------------
8 4 9 | 7 6 2 | 1 5 3
2 5 6 | 8 1 3 | 7 4 9
7 1 3 | 9 4 5 | 2 8 6
---------------------
3 2 5 | 4 7 6 | 8 9 1
1 6 7 | 2 8 9 | 4 3 5
9 8 4 | 3 5 1 | 6 7 2
```

Assignment Extensions

In addition to solving the sudoku problem, I also implemented the Australia map coloring problem in CNF form and used WalkSAT to find a solution.

I encoded the colors ('G', 'R', 'B') inside the CNF file so that each state can have one of the three colors, but not two colors.

For example, in order to show that the state 'WA' can only be red, green, or blue, we add the following clause:

```
-SA__G -SA__R
-SA__G -SA__B
-SA__B -SA__R
```

Additional clauses are needed to ensure that adjacent states are not the same color. An example showing that 'WA' cannot be the same color as 'NT' can be seen below.

```
-WA__G -NT__G
-WA__R -NT__R
-WA__B -NT__B
```

Lastly we need to add the clauses that ensure each state is assigned a color. For this we need to create True clauses that represent each state and each of the three colors: Red, Green, and Blue. An example Clause can be seen below.

```
NSW_B NSW_G NSW_R
```

After making a few slight modifications to the helper functions in the SAT class and running WalkSAT on the map coloring problem, I got a correct solution! Seen below.

```
walksat solution
Number of Tries: 1
Number of Flips: 11
Solved Board: {1: False, 2: False, 3: True, 4: True, 5: False, 6: False, 7: True, 8: False, 9: False, 10: False,
11: True, 12: False, 13: False, 14: False, 15: True, 16: True, 17: False, 18: False}
```

Note that there are 18 resulting binary values which corresponds to three colors multiplied by six states in the map. Six of the values are set to True, showing that one color has been chosen for each of the states in the map. Also note that Tazmania was excluded from this problem since there are no constraints and it can be any color.

The solution corresponds to the following assignment:

```
{'NT__R': True, 'Q___B': True,'V___B': True, 'SA__G': True, 'NSW_R': True, 'WA__B': True}
```

The graph that corresponds to this assignment can be seen below.