CS76 Assignment 5: GSAT and WalkSAT applied to CNF problems - Sudoku

Himadri Narasimhamurthy | 2/14/19

Introduction

The cnf problem is a useful way of modelling and solving problems dealing with assigning values to variables along some particular set of rules or constraints which are put together as boolean expressions. I went about designing my SATs in the manner described in design notes - first I created a SAT class which can be applied to any cnf problem, then I applied it to the Sudoku problems and the Australia Map constraint satisfaction problem from the CSP assignment.

The CNF Problem

The SAT class (**SAT.py**) was where the breakdown of the CNF problem happened. Since I needed to populate my constructor for SAT with the variables, the constraints, and the solution - I had to parse the .cnf file and loop through the lines writing in constraints and variables.

As I wrote my SAT **init**, I allowed it to take in a file name which can either be a .sud file with a .cnf file written or a .cnf file directly. I also gave the user the ability to limit the SAT if they wanted to - I limit automatically at 100,000 loops.

```
def __init__(self, file_name, limit = 100000):
```

The data structures that I used for variables, clauses, and the solution are below. I originally had variables as an array which caused SAT to run incredibly slowly due to the number of repeated variables so I switch to set. The clauses are in an array and the solution is initialized as a dictionary. The satisfied and unsatisfied lists are empty arrays as well.

```
    self.variables = set()  #no repeats!
    self.clauses = []
    self.solution = {}

    #for use in SATs
    self.satisfied = []
    self.unsatisfied = []
```

I played around a lot with the data structures that I could use to store our solution starting with an array of tuples, an array of Variable objects which contained string variables and their boolean values, and then finally a dictionary. Due to the necessity to be able to access the boolean of each variable determined by the SAT algorithms I decided on a dictionary (key is strings of the variables and value is boolean) for quick lookup and change which was implemented as such and populated with random booleans before SAT:

```
# generates a random boolean for each variable - will change later
 for v in self.variables:
     # got hint to generate rand boolean from here (https://stackoverflow.com/questio
ns/6824681/get-a-random-boolean-in-python)
     self.solution[v] = bool(random.getrandbits(1))
```

Now, in order to populate our self variables, we need to read our .cnf file - which I read line by line and append each variable to a list that will become one clause in self.clauses. Then, we add the variable as a string to self.variables (with no negative, all we want are the values). Finally, at the end of looping through a clause and adding all of its clause variables to the temporary array, we sort the variables

```
    #read cnf lines from .cnf file
    file = file_name[:-4] + ".cnf"
    f = open(file, "r")
    cnfs = f.readlines()   #all lines in file as a list

    #populate variables and clauses needed for SAT
    for c in cnfs:
        vars = []
        for v in c.split():
            vars.append(v)

            #update variable list
            if v not in self.variables:
                stripped_v = v.strip().strip("-")
                self.variables.add(stripped_v)

        #populate clause list - each line with or and each clause with and
        self.clauses.append(clause)

    f.close()
```

GSAT

The first SAT algorithm that I implemented was GSAT - which works by checking the length of . The code runs until the limit which you set in the constructor. Then we get a random fraction between 0 and 1 - if it is above the threshold, we chose a random variable to flip. Else, if it is less than the threshold, we get the heighest valued variable. Flip the boolean of whichever variable we get. See code at **GSAT**

```
            # if random frac is above threshold
            if random.uniform(0, 1) > self.threshold:

                # choose a random variable
                r = random.randint(0, len(self.variables) - 1)
                var = str(list(self.variables)[r])

            else:
                # if below threshold, then get the var with highest val
                var = self.highest_var(self.variables)

            # flip the variable's boolean
            self.solution[var] = not self.solution[var]
```

There are two helper functions - first is **end_sat**. This is the check of whether we exit the loop and have satisfied all conditions and the length of our unsatisfied list is 0. This function is also helped by **update*clause*list** - update clause list iterates through the variables in the clause (positives and negatives) and ORs them together with the sat boolean - then the entire clause expression is ANDed with the final end boolean.

We then have our second helper function, **highest_var** which takes in our variable list and returns the highest valued variable to flip if we are below the threshold.

After looking into ways to value the variables, I decided on a simple heuristic which finds the variable that has the longest satisfied list after the boolean is flipped.

```
    #flip the boolean to make sat and unsat for when the boolean would be flipped
        self.solution[v] = not self.solution[v]

        #updates the clause lists - satisfied and unsatisfied
        self.end_sat()

        #flip back so that we can still flip at the end of sat
        self.solution[v] = not self.solution[v]
        curr_score = len(self.satisfied)
```

GSAT Performance

All tests are running with random seed of 41 and threshold of 0.07 as seen on piazza

The GSAT, as expected, ran extremely slowly on any puzzle with more cnfs than the rows puzzle. When printing on each iteration, it was computationally expensive to

1. One Cell

    ```
    solution found in 5 iterations of SAT
    ```

2. Rows

```
    solution found in 1033 iterations of SAT
    #took 26 minutes
    6 4 8 | 3 2 5 | 7 9 1
    9 8 2 | 6 5 3 | 7 1 4
    4 9 8 | 2 5 1 | 7 3 6
    ---------------------
    9 4 7 | 6 1 8 | 5 2 3
    8 6 4 | 9 2 7 | 5 1 3
    7 5 3 | 6 2 1 | 9 4 8
    ---------------------
    6 4 1 | 9 7 5 | 2 8 3
    2 4 8 | 3 7 1 | 6 5 9
    4 2 7 | 3 8 6 | 5 1 9
```

WalkSAT

WalkSAT has a lot of similar infrastructure as GSAT and uses the same two helper functions to check whether we have satisfied conditions and to get highest valued variable.

The change comes before we choose our random threshold - instead of choosing a random variable from the entire list of variables, we choose a random unsatisfied clause and choose a random variable from that clause if we're above the threshold - what this does is ensure that we attempt to satisfy an unsatisfied clause instead of picking a random variable.

```
    # pick random unsatisfied clause
            r = random.randint(0, len(self.unsatisfied) - 1)
            clause = self.unsatisfied[r]
```

Though the program was working faster than GSAT, as I ran the program while printing out the length of the unsatisfied clause list, it would often be stuck on a small number of unsatisfied clauses (i.e. 2-4). This is something that I would like to explore fixing for my extra credit implementation of a random walk.

WalkSAT Performance

All tests are running with random seed of 41 and threshold of 0.07 as seen on piazza

The results are not identically reproducable in terms of which solution was chosen or the time/number of iterations due to the random nature of the walk - this means that even when the same code was run on the same puzzle twice, there would be small variations in number of iterations/speed.

1. One Cell

```
    solution found in 3 iterations of SAT
```

2. Rows

```
    solution found in 367 iterations of SAT
```

3. Rows and Cols

```
    solution found in 3261 iterations of SAT
    #took 72 seconds
```

4. All Cells

```
    solution found in 283 iterations of SAT
    #took 4 seconds
```

5. Rules

```
    solution found in 1239 iterations of SAT
    #took 19 seconds
```

6. Sudoku1

```
    solution found in 29413 iterations of SAT
    #took 9 mins
    5 8 6 | 4 2 3 | 1 7 9
    3 7 4 | 6 9 1 | 8 5 2
    1 9 2 | 5 7 8 | 4 6 3
    ---------------------
    8 1 3 | 9 6 5 | 7 2 4
    2 4 9 | 8 3 7 | 5 1 6
    7 6 5 | 1 4 2 | 9 3 8
    ---------------------
    4 5 7 | 3 8 6 | 2 9 1
    6 2 8 | 7 1 9 | 3 4 5
    9 3 1 | 2 5 4 | 6 8 7
```

7. Sudoku2

```
    solution found in 15827 iterations of SAT
    #took 5 minutes
    5 3 1 | 9 2 6 | 7 4 8
    2 7 8 | 1 3 4 | 6 5 9
    4 9 6 | 7 5 8 | 1 3 2
    ---------------------
    8 1 9 | 4 6 7 | 5 2 3
    6 5 2 | 8 9 3 | 4 7 1
    7 4 3 | 5 1 2 | 9 8 6
    ---------------------
    1 6 4 | 3 8 5 | 2 9 7
    3 2 7 | 6 4 9 | 8 1 5
    9 8 5 | 2 7 1 | 3 6 4
```

8. Bonus Puzzle

```
    solution found in 34081 iterations of SAT
    #took 11 minutes
    5 3 4 | 6 7 8 | 9 1 2
    6 7 2 | 1 9 5 | 3 4 8
    1 9 8 | 3 4 2 | 5 6 7
    ---------------------
    8 5 9 | 7 6 1 | 4 2 3
    4 2 6 | 8 5 3 | 7 9 1
    7 1 3 | 9 2 4 | 8 5 6
    ---------------------
    9 6 1 | 5 3 7 | 2 8 4
    2 8 7 | 4 1 9 | 6 3 5
    3 4 5 | 2 8 6 | 1 7 9
```

Extension: Map Coloring Problem

I was able to model the australia map coloring problem as a cnf problem which can be found in full in **australia.cnf**. The construction of the problem as a CNF problem is simple - we have our OR clauses as positive (i.e. WAr WAg WAb) and our AND clauses as negative (i.e. -WAr -NTr) and each clause is comprised of the territory abbreviation (7) and a color option (3). I could also test that my SAT did not try to find solutions to impossible problems so I modified the .cnf file to create a new one **aus.cnf** which has all territories and only two colors and made sure that my SAT found no solutions within the imposed limit.

The problem is solved in **solve_australia.py** by feeding the cnf into SAT in the same way it is done in solve_sudoku.py and then parsing the result .sol file in order to display the territory and the color. The output on a run of GSAT is as follows:

```
solution found in 9 iterations of SAT
WA maps to red

NT maps to green

NSW maps to green

T maps to green

Q maps to red

V maps to red

SA maps to blue
```

Extension: Running WalkSAT with Spence's Constraints

My hypothesis prior to running the code with the added constraints was that it would slow the speed of walkSAT considerably in the step where we are checking if we have any unsatisfied clauses and updating sat and unsat lists since we exponentially have increased the number of clauses. This did show up as true (the first puzzle is without Spence's constraints, the second is with) and the second took about 20 seconds longer on puzzle1. However, since the clauses are more selective, we can see that it iterated significantly fewer times - which shows that the majority of the time went to computing the scores on each of the variables in the larger number of clauses. You can run and observe similar results from **solve_sudoku.py**.

```
solution found in 23843 iterations of SAT
the code took 436.38047099113464 seconds to run.
5 7 6 | 4 3 8 | 1 2 9
1 8 4 | 2 9 6 | 5 3 7
9 2 3 | 1 7 5 | 8 6 4
---------------------
8 3 5 | 7 6 1 | 4 9 2
4 9 1 | 8 2 3 | 7 5 6
7 6 2 | 5 4 9 | 3 8 1
---------------------
3 1 9 | 6 5 7 | 2 4 8
6 4 8 | 3 1 2 | 9 7 5
2 5 7 | 9 8 4 | 6 1 3

solution found in 9691 iterations of SAT
the code took 458.26591420173645 seconds to run.
5 6 7 | 4 3 1 | 8 2 9
4 2 8 | 7 9 6 | 5 1 3
3 1 9 | 5 8 2 | 4 6 7
---------------------
8 3 4 | 9 6 5 | 1 7 2
1 9 6 | 8 2 7 | 3 5 4
7 5 2 | 1 4 3 | 6 9 8
---------------------
9 7 1 | 3 5 8 | 2 4 6
6 4 3 | 2 1 9 | 7 8 5
2 8 5 | 6 7 4 | 9 3 1
```

In order to add the extra constraints, I simply added new row*clause2 and col*clause2 functions to **Sudoku.py** that were based entirely off the given code for cell clause. This allowed me to add this base code (changed variables slightly for col_clause2) to the existing for loops to get each negative AND constraint for rows and columns.

```
def row_clause2(self, r):
    .
    .
    .
     #added to go through each col in row and add each neg constraint
        for ci in range(1, 10):
            for cj in range(ci + 1, 10):
                s += self.sudoku_literal(r, ci, value, neg=True) + " "
                s += self.sudoku_literal(r, cj, value, neg=True) + " "
                s += "\n"

    return s
```