Report

COSC276, Fall2021, PA5, Xuedan Zou

A. Description

In this problem we convert the sudoku problem into a CNF satisfication problem and try to solve it by both the WalkSAT and GSAT algorithm.

The first challenge here is to convert the sudoku problem into CNF. In order to do so, consider each cell in the sudoku, it has in total nine possible values: 1~9. We use a three digit number to indicate the possible value of each cell: *abc*, where a, b and c all range from 1 to 9. By this means, we use a and b to indicate the x and y coordinate value seperately of a cell, with the left upper coner to be 11, and right bottom coner to be 99. The coorinate for each cell in a standard sudoku can be indicated as follow:

```
11 12 13 14 15 16 17 18 19
21 22 ...           ... 29
...                    ...
91 92 ...           ... 99
```

We then use c to indicate the input value of that cell. So variable 111 means that the cell located at the coordinate 11 with value 1. We now know that for each variable, there can only be two values, true and false, and that can be indicated as the symbol. So *111* means the value of the cell located at the coordinate 11 is with value 1 and *-111* means is not with value 1. For a standard sudoku problem, we have 9*9 cells in total, and for each cell it has 9 possible values, thus the total number of variables we have for a standard sudoku problem is 729 (9^3).

Now we are able to convert a given sudoku problem to its CNF expression.(it has been provieded in the given original codes) For each clause of in CNF, at least one variable should be true. (to make the clause true and thus the whole CNF could be true) We use an **assignment** list in our SAT model to store the true and false value for each variable. For each variable, it has a mapped index value and the absolute value of that index value refers to the index of the assignment. Since before our variables range from 111 to 119, then from 121 to 129 etc. , we should convert these three digit numbers to continuous indexes begun from 1. That means 111 maps to 1, 119 maps to 9, and then 121 maps to 10 ... The reason we want those indexes begun from 1 rather than 0 is because we want to use the positive and negative symbol to distinguish the true and false for each variable. If we used 0 to indicate 111, then the variable 111 and -111 are converted to the same index 0 which causes problems. So now variable 111 has its index value 1 and store as *assignment[1] = 1*, while variable -111 has its index value -1 and store as *assignment[1] = -1*. We use the following code to convert the given CNF file to the corresponded index values:

```
  # convert the str value like 111 to the variable value like 1
    def converter_stoi(self, in_str):
        # convert this str to int value
        int_str = int(in_str)
        value = abs(int_str)
        # use abc to express a three digit number, then here index = 81*(a-1) + 9*(b-1) +
c
        c = value % 10
        b = int((value % 100) / 10)
        a = int(value / 100)
        index = 81 * (a - 1) + 9 * (b - 1) + c

        self.itos[index] = int_str

        if int_str < 0:
            index = -index
        return index
```

We then have to implement the GSAT algorithm. The step of GSAT algorithm is given on Gradescope:

```
1) Choose a random assignment (a model).
2) If the assignment satisfies all the clauses, stop.
3) Pick a number between 0 and 1. If the number is greater than some threshold h, choose a
variable uniformly at random and flip it; go back to step 2.
3) Otherwise, for each variable, score how many clauses would be satisfied if the variable
value were flipped.
4) Uniformly at random choose one of the variables with the highest score. (There may be
many tied variables.) Flip that variable. Go back to step 2.
```

GSAT combines the random algorithm together with a scoring based choosing method. The idea for GSAT in general is keeping trying until finding a solution. Here in step 3 we may get a list of the variableswith the same (highest) score, so we should maintain a list to store all of these satisfied variables and randomly pick one to flip. We use a **dictionary** here to store the score of each value as follow:

```
                flip_count = {}
                for i in range(1, self.variables_num + 1):
                   # score how many clauses can be true if one of the variable
                   # is fliped
                   self.assignment[i] *= -1
                   flip_count[i] = self.clause_satisfy()
                   self.assignment[i] *= -1
```

We implement a clause_satisfy function to help us count how many clauses are now satisfied with the current list of assignment. We also maintain an unsatisfied list at the same time(this is used to implement the WalkSAT algorithm disscussed later):

```
    # count how many clause are true given the assignment
    def clause_satisfy(self):

        clause_true = 0
        self.unsatisfied = []
        for clause in self.cnf:
            # for each clause at least one term is true
            is_true = False
            for i in clause:
                index = abs(i)
                if i > 0:
                    bool = 1
                else:
                    bool = -1
                if self.assignment[index] == bool:
                    clause_true += 1
                    is_true = True
                    break
            # if this clause is false, then add it to the unsatisfied list
            if is_true == False:
                self.unsatisfied.append(clause)
```

Finally, we implement WalkSAT algorithm here to solve the large-scale CNF problem like sudoku problem here much more quickly. The key trick of WalkSAT is choosing a smaller number of variables that might be flipped. WalkSAT only choose the variable from a random unsatisfied clues to flip rather than from all the variables. By this way, the speed in scoring process can be greatly improved. We have shown our clause_satisfy function before and we maintained a **unsatisfied list** each time we call that function, now when we randomly pick variables to flip, we can just pick variable belonged to the clause in that list:

```
        # pick a clause randomly from the unsatisfied clauses
        pick_clause = random.choice(self.unsatisfied)
```

B. Evaluation

It is not hard to understand that our GSAT algorithm works not well on a standard sized sudoku problem. It is likely to take million times of loop to get a solution. We just test our GSAT algorithm on *one_cell*,*all_cells* and *rows* mini problem and it solves all of them perfectly.

We then use our WalkSAT algorithm to solve the given sudoku problem, we set the h to be 0.7 and the max iteration time to be 100,000 and we get the solutions of the given *sudoku1* and *sudoku2* problems shown bellow:

```
Puzzle1:
5 0 0 0 0 0 0 0 0
0 0 0 0 9 0 0 0 0
0 0 0 0 0 0 0 6 0
8 0 0 0 6 0 0 0 0
0 0 0 8 0 0 0 0 0
7 0 0 0 0 0 0 0 0
0 0 0 0 0 0 2 0 0
0 0 0 0 1 0 0 0 5
0 0 0 0 0 0 0 0 0

Solution:
5 8 7 | 1 2 6 | 9 4 3
3 1 6 | 4 9 8 | 5 2 7
2 9 4 | 3 7 5 | 8 6 1
---------------------
8 4 9 | 5 6 7 | 1 3 2
1 5 3 | 8 4 2 | 7 9 6
7 6 2 | 9 3 1 | 4 5 8
---------------------
9 7 1 | 6 5 3 | 2 8 4
4 3 8 | 2 1 9 | 6 7 5
6 2 5 | 7 8 4 | 3 1 9

Puzzle2:
0 3 0 0 0 0 0 0 0
0 0 0 1 0 0 0 0 0
0 9 0 0 0 0 0 0 0
8 0 0 0 6 0 0 0 3
0 0 0 8 0 0 0 0 0
7 0 0 0 0 0 0 0 6
0 0 0 0 0 0 0 0 0
0 0 0 0 0 9 0 0 5
0 0 0 0 0 0 0 0 0

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

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

Obviously, both of the solutions are correct. However, when we try our WalkSAT with the *puzzle_bonus*, it becomes extremely time confusing, without giving a solution within 100,000 iteration times. We expand our maximum looping time into 1,000,000 and try again, after about 600,000 times of iteration, we finally get the solution bellow:

```
Bonus:
5 3 0 0 7 0 0 0 0
6 0 0 1 9 5 0 0 0
0 9 8 0 0 0 0 6 0
8 0 0 0 6 0 0 0 3
4 0 0 8 0 3 0 0 1
7 0 0 0 2 0 0 0 6
0 6 0 0 0 0 2 8 0
0 0 0 4 1 9 0 0 5
0 0 0 0 0 0 0 0 0

Solution:
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
```

We do count the satisfied clauses for each time of iteration when running WalkSAT and notice that with more times of iteration, the number of satisfied clauses in general is imporving (but with a relatively slow pace).

C. Bonus

In the original WalkSAT algorithm we consider all of the variables in sudoku as real variables and flip any of the varibles that are satisified with specific condition. However, for the sudoku problem we have already known the values of some variables in the very beginning. We may improve the speed of our WalkSAT by treating those first given constant cells as real constants rather than variables.

As a result, we maintain a list of **constant** with the initial given constant cells' indexs with their symbols. For our sudoku problem, the symbols are always positive. Then before we implement the steps of WalkSAT, we first put the values(1 or -1) of constant in the assignment:

```
    for i in range(1, self.variables_num + 1):
                if i in self.constant:
                    self.assignment[i] = 1
                elif -1 * i in self.constant:
                    self.assignment[i] = -1
```

Then, we shouldn't change the assignment of those constant values anymore. Each time we pick a variable and want to flip it, we should check if that variable is really a variable or a constant:

```
                # choose that flip value from the random picked clause
                random_flip_value = random.choice(pick_clause)

                # if we would consider the constant values
                if has_constant == True:
                   # we cannot flip the constants, so pick again
                   while random_flip_value in self.constant or -1 * random_flip_value in
self.constant:
                       random_flip_value = random.choice(pick_clause)

                self.assignment[abs(random_flip_value)] *= -1
```

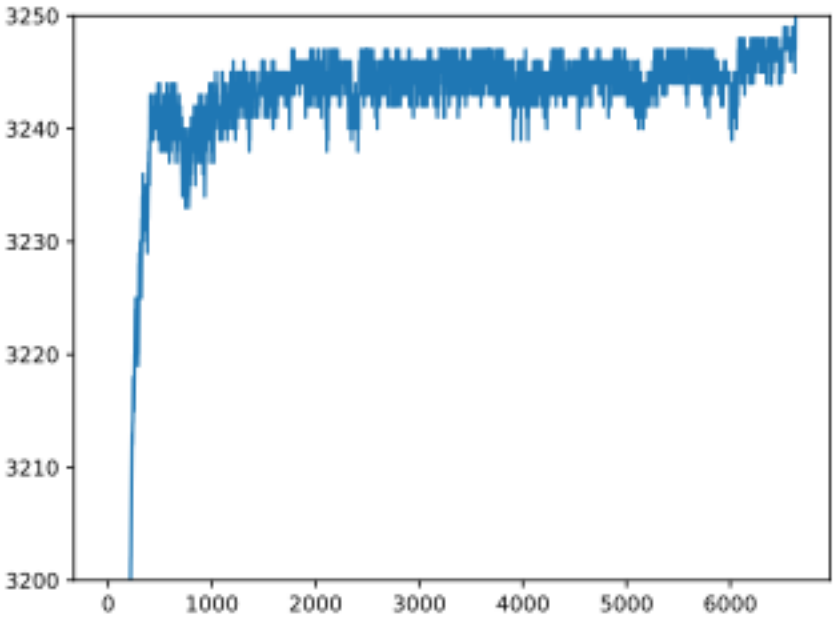
Finally, in the scoring process, we should ignore those constants by scoring them with -1:

```
                flip_count = {}
                for value in pick_clause:

                    # if we would consider the constant values
                    if has_constant == True:
                        # we cannot flip the constants, so ignore them
                        if value in self.constant or -1 * value in self.constant:
                            flip_count[value] = -1
                            continue

                    i = abs(value)
```

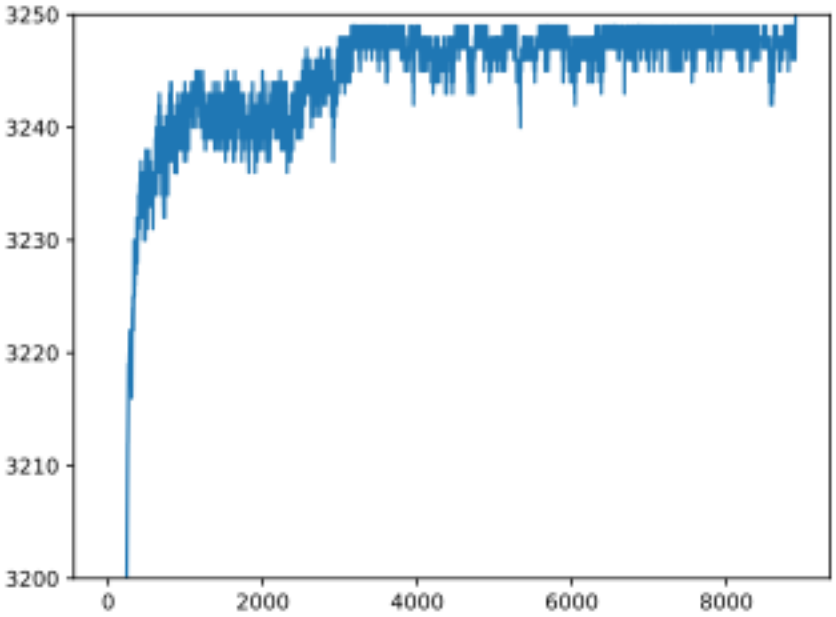
```
                    self.assignment[i] *= -1
                    flip_count[i] = self.clause_satisfy()
                    self.assignment[i] *= -1
```

Now our WalkSAT consider given constants as real constants and we test our WalkSAT with and without considering constants. To better observe the relationship between the most satisfied clauses and the iteration times, we store a list to memorize the number of most satisfied clauses (in the scoring step) during the iteration. We then use **matplotlib** to draw the result out.

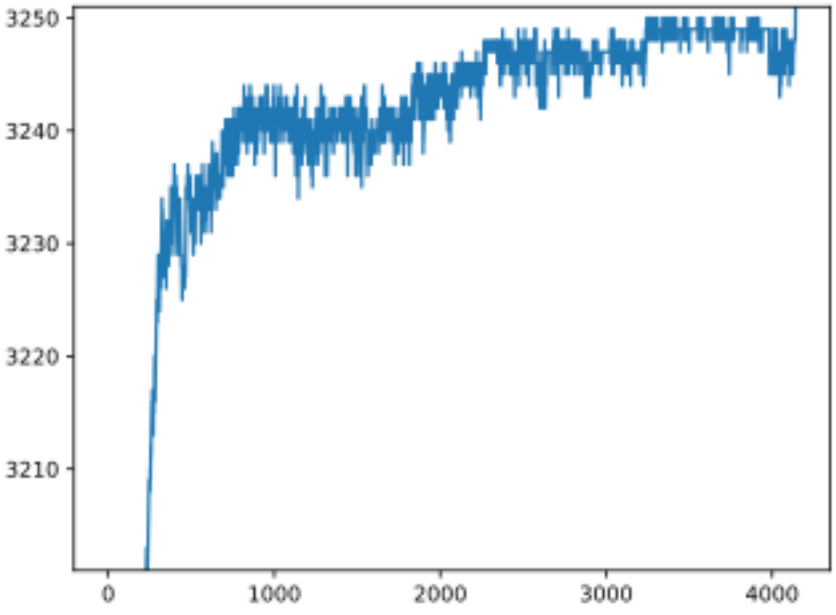
For puzzle1, if we considered the constants, it will take about 9513 times of iteration to get the solution by a given random seed.



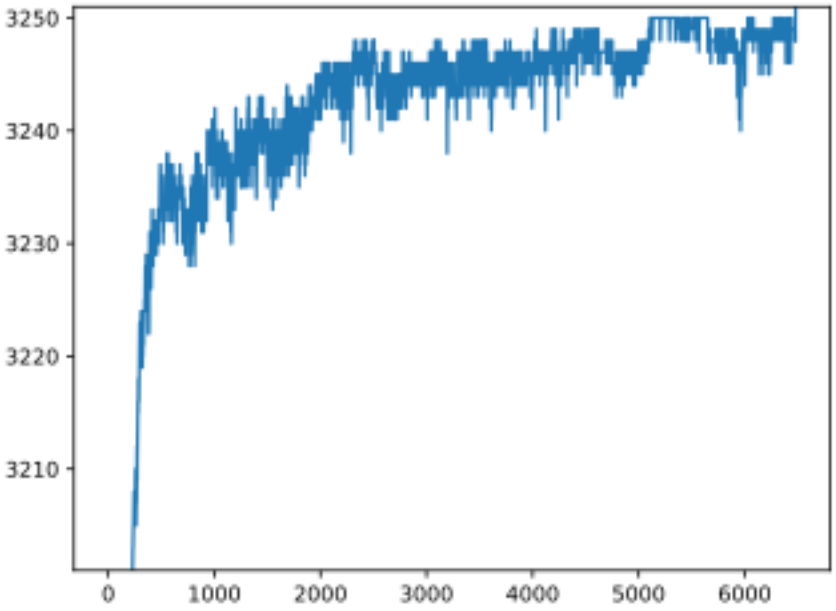
Tested with the same given random seed, it will take about 12733 times of iteration without considering the constants (with the same h and max_iterate values).



For puzzle1, if we considered the constants, it will take about 5993 times of iteration to get the solution by a given random seed.



Similarly, without considering the constants and anything else the same, it will take about 9199 times of iteration, much longer time.



As a result, we can primarily confirm that considering the constants can effectively improve the speed of our WalkSAT algorithm (by reducing the needed iteration times).