Anna Matusewicz
CS 76 Prof. Balkcom
10/26/17

Design Notes:

My solvers for this assignment are all contained in `SAT.py`. When initialized, this class takes a .cnf file which it uses to make a tuple of the variables, where the index of each is the translation, and a list of all the clauses, which are kept are sets. The GSAT and WalkSAT algorithms can then be called to solve the .cnf file. Finally, the `write_solution()` method writes a solution (.sol) file that can be read by `Display.py`. Additional functionalities I added are a timer, iteration counter, and unsolved clause counter so that the runtime, iterations, and number of unsolved clauses (in the case of a failure) for GSAT and WalkSAT can be printed with the results.

sort_clauses:

In order to keep the clauses in a usable format, I read in a .cnf file line by line and split the clause into a list of variables using `.split(" ")`. Then for each variable in the clause, I remove the negation symbol if present then, if the variable has not already been documented, I add it to the list:

```
edit = c
if c[0] == "-":
    edit = c[1:]
    negative = True
if edit not in var:
    var.append(edit)
```

Then I make a set out of the clause using the index of the variable +1 (to eliminate 0 as a possibility) and multiplying it by -1 if necessary, finally adding the set to the list of clauses:

```
if negative:
    new_clause.add(-1 * (var.index(edit) + 1))
else:
    new_clause.add(var.index(edit) + 1)
```

GSAT:

My GSAT has the option of accepting a threshhold for frequency of random flips and a premade model. If there is no premade starting model, one is made using `get_rand_model()` which essentially makes an initial guess at the solution. The frequency and model are then passed to a helper method `gsat_helper()`.

If the model is correct it is returned, otherwise GSAT will flip bits. If the random number `rando = random.random()`, between 0 and 1, is less than the threshhold then a random bit in the model is flipped. If it is higher, every variable in the model is given a score and the bit with the highest score is flipped. In the case of a tie a random bit from the highest scoring bits list is chosen. After every flip the helper method recurses. This continues until an answer is found or a RecursionError is caught, printing for example:

```
Recursion limit reached. Solution not reachable by GSAT.
No solution found after visiting 996 states
Run time: 303.2829008102417 seconds
...
```

WalkSAT:

My WalkSAT has the options for accepting a threshhold for frequency of random flips and a limit for the number of flips. The default numbers are .3 and 100,000 respectively. A model is made using `get_rand_model()` which essentially makes an initial guess at the solution. Instead of recursing however, a for-loop is utilized so that at most a certain number of flips can be made.

If the model is correct it is returned, otherwise WalkSAT will flip bits. First however, it finds a set of all the unsatisfied clauses using `get_union_set()` and chooses a random one from it. If the random number between 0 and 1 is less than the threshhold then a random bit in the chosen clause is flipped. If it is higher, the variables in the clause are given a score and the bit with the highest score is flipped. In the case of a tie a random bit from the highest scoring bits list is chosen:

```
for var in clause:
    model_temp = set(model)
    model_temp.remove(-1 * var)
    model_temp.add(var)

    score = self.count_corr_clauses(model_temp)
    # score is key for a list of bits
    if score in scores:
        scores[score].append(var)
    else:
        scores[score] = [var]

    # Get the highest scoring bit
    sort = sorted(scores.keys(), reverse=True)
    highest_bit = random.choice(scores[sort[0]])

    model.remove(-1 * highest_bit)
    model.add(highest_bit)
```

After every flip the for loop restarts. This continues until an answer is found or the flip limit is reached in which case WalkSAT returns `False`.

Scoring:

Scoring occurs in the `count_corr_clauses()` method by utilizing the `.is_disjoint()` function of sets where disjoint sets indicate the clause is unfulfilled:

```
count = 0
for clause in self.clauses:
    if not clause.isdisjoint(model):
        count += 1

return count
```

Getting Unsatisfied Clauses:

The method `get_union_set()` is passed the working model and utilizes the `.isdisjoint()` function of sets to determine whether or not a clause is unsatisfied:

```
union = []
for clause in self.clauses:
    if clause.isdisjoint(model):
        union.append(clause)
```

Testing:

My first debugging issue occured when GSAT was too slow to handle all_squares.cnf. I realized when inspecting my code that I was keeping my model and my individual clauses in lists but using them as sets, switching back and forth between the two each time. After changing my code to only keep them as sets, the code was exponentially faster and all_squares.cnf and rows.cnf could both be solved with GSAT.

Current GSAT results:

one_cell.cnf

```
Solved after visiting 6 states
    Solved in: 0.0005362033843994141 seconds

8 0 0 | 0 0 0 | 0 0 0
0 0 0 | 0 0 0 | 0 0 0
0 0 0 | 0 0 0 | 0 0 0
---------------------
0 0 0 | 0 0 0 | 0 0 0
0 0 0 | 0 0 0 | 0 0 0
0 0 0 | 0 0 0 | 0 0 0
---------------------
0 0 0 | 0 0 0 | 0 0 0
0 0 0 | 0 0 0 | 0 0 0
0 0 0 | 0 0 0 | 0 0 0
```

all_cells.cnf

```
Solved after visiting 510 states
Solved in: 140.72616124153137 seconds

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

rows.cnf

```
Solved after visiting 700 states
Solved in: 196.24468398094177 seconds

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

My largest issue, however, was that my scoring method was preventing my WalkSAT from working beyond rows.cnf. It would plateau around 20-30 clauses left unsolved for rows_and_cols.cnf until it reached the maximum amount of flips allowed. After unit testing in which every other method of my solver worked, I figured out that the `count_corr_clauses()` was causing the issue. Before I fixed it, the counter would skip clauses that the changed bit were not a part of by using:

```
if var not in clause:
    continue
```

I found however that this caused there to be inaccurate scoring as it did not reflect the amount of clauses changed from unsolved to solved. By removing this if statement, it may take slightly longer in the counting method, but the counts became a more accurate reflection of what the best bit to flip was and ultimately sped up my code and reduced the amount of flips needed to solve each board.

Current WalkSAT results with random seed of 1:

one_cell.cnf

```
Solved after visiting 4 states
Solved in: 0.0002582073211669922 seconds

3 0 0 | 0 0 0 | 0 0 0
0 0 0 | 0 0 0 | 0 0 0
0 0 0 | 0 0 0 | 0 0 0
---------------------
0 0 0 | 0 0 0 | 0 0 0
0 0 0 | 0 0 0 | 0 0 0
0 0 0 | 0 0 0 | 0 0 0
---------------------
0 0 0 | 0 0 0 | 0 0 0
0 0 0 | 0 0 0 | 0 0 0
0 0 0 | 0 0 0 | 0 0 0
```

all_cells.cnf

```
Solved after visiting 298 states
Solved in: 0.4709172248840332 seconds

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

rows.cnf

```
Solved after visiting 418 states
Solved in: 0.817852258682251 seconds

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

rows_and_cols.cnf

```
Solved after visiting 1684 states
Solved in: 4.6110100746154785 seconds

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

rules.cnf

```
Solved after visiting 4080 states
Solved in: 11.939801931381226 seconds

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

puzzle1.cnf

```
Solved after visiting 12516 states
Solved in: 35.15302586555481 seconds

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

puzzle2.cnf

```
Solved after visiting 9604 states
Solved in: 27.181575775146484 seconds

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

puzzle_bonus.cnf

My WalkSAT does have the ability to solve puzzle_bonus.cnf in about 2 minutes, however it only works about 50% of the time when given the parameters of 100,000 flips max and a threshhold of .3. Additionally, it does not work for a random seed of 1.

Results of 10 WalkSAT attempts:

1.

```
No solution found after visiting 100000 states
Run time: 282.10074186325073 seconds
Clauses left unsolved: 7
```

2.

```
Solved after visiting 36517 states
Solved in: 127.12821412086487 seconds

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
```

3.

```
No solution found after visiting 100000 states
Run time: 634.3941159248352 seconds
Clauses left unsolved: 5
```

4.

```
Solved after visiting 34368 states
Solved in: 102.37410306930542 seconds

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
```

5.

```
Solved after visiting 85337 states
Solved in: 246.23885798454285 seconds

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

6.

```
Solved after visiting 70451 states
Solved in: 182.59933304786682 seconds

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

7.

```
No solution found after visiting 100000 states
Run time: 273.6039640903473 seconds
Clauses left unsolved: 1
```

8.

```
Solved after visiting 25909 states
Solved in: 72.25825572013855 seconds

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
```

9.

```
No solution found after visiting 100000 states
Run time: 248.60510110855103 seconds
Clauses left unsolved: 2
```

10. When given a seed of 1

```
No solution found after visiting 100000 states
Run time: 255.99513697624207 seconds
Clauses left unsolved: 1
```