COSC 76 Project 5: Sudoku and Propostional Logic

Seungjae Jason Lee

Introduction

Objective

The objective of the project is to write two solvers for propositional logic problems: GSAT, and WALKSAT. Then, we aim to test these solvers by solving sudoku problems. We have been provided with python files that would read sudoku puzzles and convert them into cnf. Thus, our goal is to write SAT class that solves propositional logic problems using two solvers.

Propositional Logic Problem Rules

1. Sudoku: Given fixed slots of the board, find a solution to the game. Every column, row, and square areas (blocks) should not share same numbers. (same rule as regular sudoku).
2. Map Coloring(Extension): The goal of the problem is to assign user-provided colors to different regions. The only rule is that neighboring regions should not have same colors.
3. N Queen Problems(Extension): The goal of the problem is to find a way to assign N chess queens on N by N board so that no queens attack each other.

Code Design

Since the sudoku class was already implemented for us, we only have to implement SAT class, which solves propositional logic problems using Gsat and Walksat.

Building SAT Model

SAT class

`__init__` takes the filename of cnf file. The method builds the problem through `build_problem` method and initializes assignment for each variable (True or False) through `random_assignment` method. These are important variables of the class:

1. `self.assignment` : a list of True and False. Each slot is equivalent to a variable of a problem.

2. `self.clauses` : a list of sets where each set represents clauses. Each atomic sentence in a set is represented as +(index + 1) or -(index+1) where negative number indicates the negation. For example, if variable '119' is stored at index 0 of self.assignment, then +1 and -1 are possible atomic sentences of that variable.

3. `self.index_to_value` , `self.value_to_index` : dictionaries that convert from index + 1 of self.assignment to the actual variable name/ number of a problem and also the other way as well.

4. `self.iteration` : The maximum iteration for walksat solver. If a solution is not found in the given interations, walksat stops looking for an answer.

As for the methods, there are five important helper methods:

- `build_problem(self, filename)` : This method takes a filename of a cnf file and reads the file to build the problem. It does three things to build a problem: build clauses, build dictionaries mentioned above, and initializes self.assignment with the right size.

- `random_assignment(self)` : This method assigns random value(True or False) for each slot in the self.assignment. For each random assignment, I used random.random to generate a random floating number less than 1 and assigned True if the number is less than 0.5. Else, I assigned False.

- `check_clause(self)` : This method checks if the current assignment in self.assignment satisifies all of the clauses of a problem. It iterates over each clause and checks each atomic sentence in a clause.

- `count_sat_clauses(self, index)` : This method counts how many clauses are satisfied if we change the value at given index. So, for given input index, we change the boolean value at self.assignment[index] and iterates over clauses to count. This method is helper method for Walksat solver.

- `write_solution(self, filename)` : This method takes a filename as input. It writes a solution to a problem to a file with given name. Each line of a file has a single variable either as positive or negative. A variable is True if positive and False if negative.

Implementing GSAT

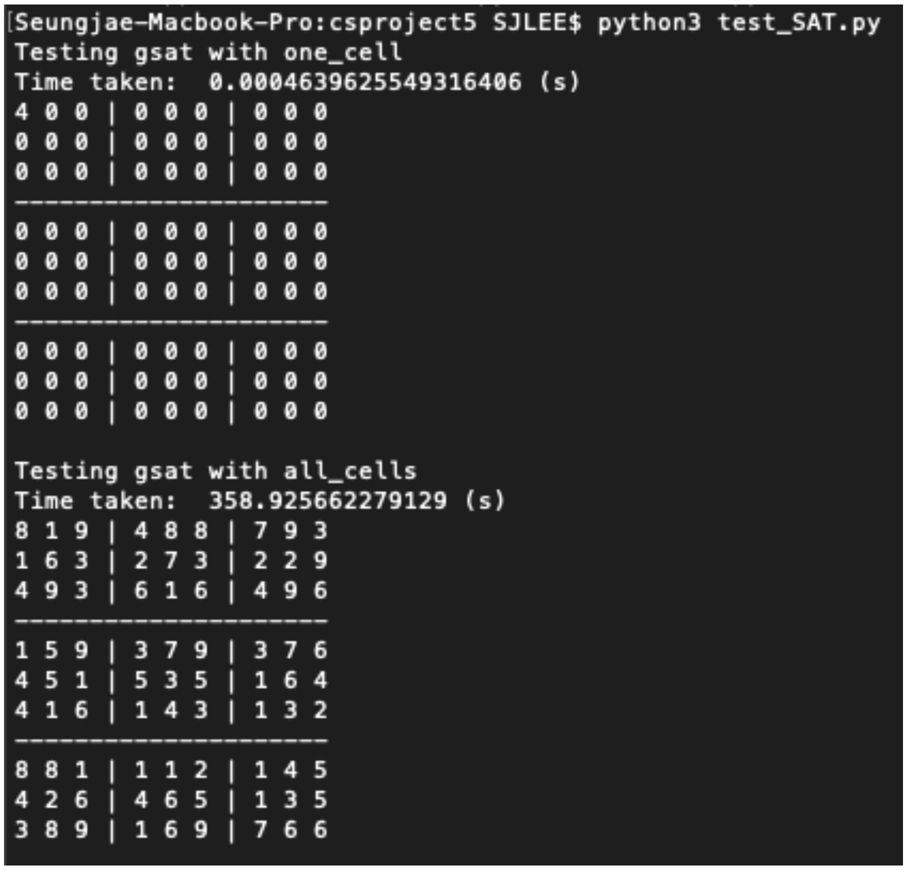
For the GSAT, there are two options for each iteration. If a random number is bigger than threshold number, GSAT randomly flips one variable and checks if all of the clauses are satisfied. If the random number is less than threshold number, GSAT finds a variable with most number of satisified clauses if flipped, flips that variable, and checks if all of the clauses are satisfied.

Implementing Walksat

For Walksat, it uses a similar approach with a bit of modification. Similar to Gsat, for each iteration, if a random number is bigger than the threshold number, the algorithm randomly flips one variable and checks if all of the clauses are satisfied. Else, it finds all the clauses that are unsatisfied. Then, it randomly chooses one clause and scores each variable in the clause. Then, the algorithm finds a variable with highest score (most number of satisfied clauses if flipped), flips that variable, and checks if the current assignment satisfies all of the clauses.

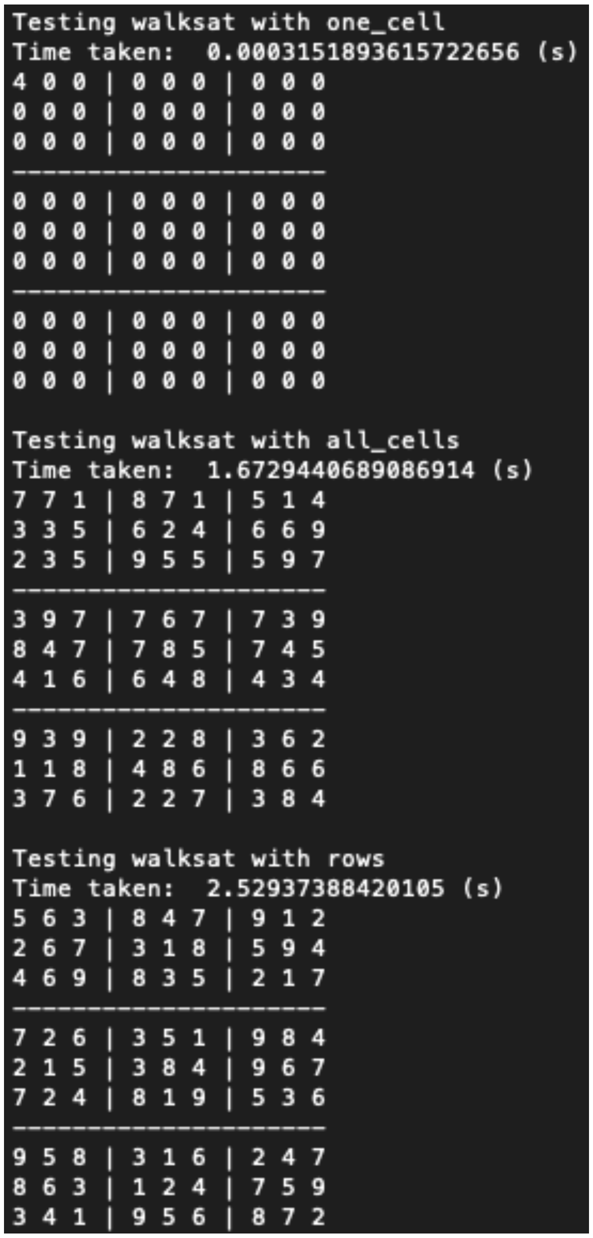
Evaluation

Testing GSAT: For GSAT, I tried first two cnfs: one_cell and all_cells.cnf. Since all_cells.cnf worked and took a long time relative to one_cell.cnf, I decided not to continue and test other cnfs as they are more complicated. Demonstration of all_cells.cnf is good enough to show that GSAT is very time consuming when the number of variable and number of clauses increase. The picture of result is shown below. I used threshold of 0.7.

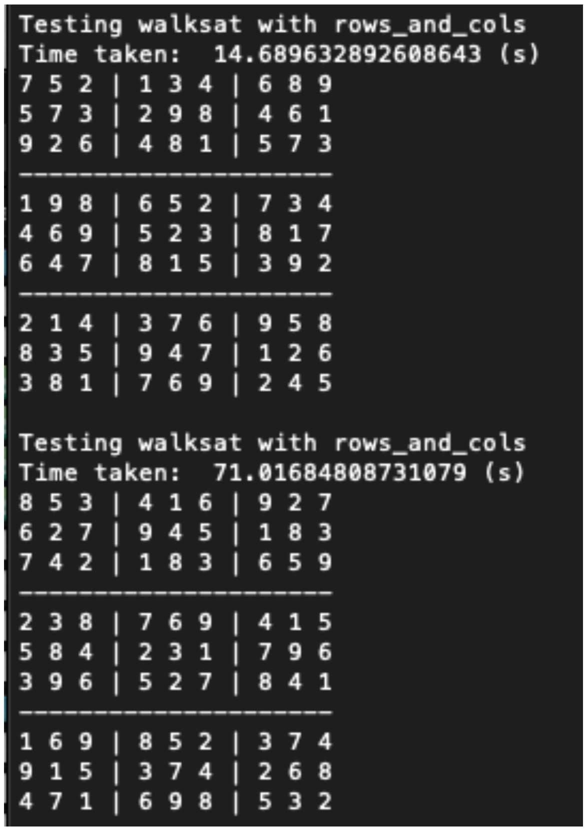


```
[Seungjae-Macbook-Pro:csproject5 SJLEE$ python3 test_SAT.py
Testing gsat with one_cell
Time taken:  0.0004639625549316406 (s)
4 0 0 | 0 0 0 | 0 0 0
0 0 0 | 0 0 0 | 0 0 0
0 0 0 | 0 0 0 | 0 0 0
---------------------
0 0 0 | 0 0 0 | 0 0 0
0 0 0 | 0 0 0 | 0 0 0
0 0 0 | 0 0 0 | 0 0 0
---------------------
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

Testing gsat with all_cells
Time taken:  358.925662279129 (s)
8 1 9 | 4 8 8 | 7 9 3
1 6 3 | 2 7 3 | 2 2 9
4 9 3 | 6 1 6 | 4 9 6
---------------------
1 5 9 | 3 7 9 | 3 7 6
4 5 1 | 5 3 5 | 1 6 4
4 1 6 | 1 4 3 | 1 3 2
---------------------
8 8 1 | 1 1 2 | 1 4 5
4 2 6 | 4 6 5 | 1 3 5
3 8 9 | 1 6 9 | 7 6 6
```

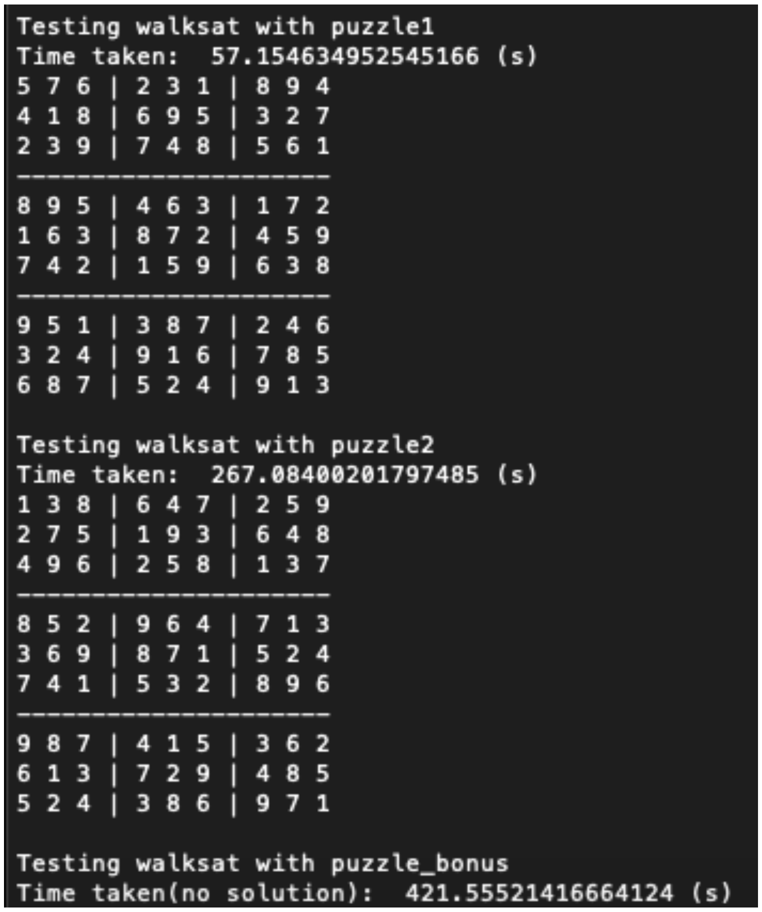
Testing Walksat: I tried all_cnfs with walksat. Although one_cell.cnf took more time than GSAT as the problem is simple, every other cnfs were faster. In the first picture, we can see that all_cells.cnf took only 1.67 seconds while GSAT took 6 minutes. The second picture shows that the runtime varies due to randomness of the algorithm; I ran same problem two times but got different results and different computation time (14 seconds vs 71 seconds). Regardless of the randomness and variation of computation time, we can still conclude that walksat is faster and more efficient than GSAT. The third picture shows the output of running some of the harder puzzle cnfs; however, I failed to find a solution for puzzle_bonus.



```
Testing walksat with one_cell
Time taken:  0.0003151893615722656 (s)
4 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

Testing walksat with all_cells
Time taken:  1.6729440689086914 (s)
7 7 1 | 8 7 1 | 5 1 4
3 3 5 | 6 2 4 | 6 6 9
2 3 5 | 9 5 5 | 5 9 7
---------------------
3 9 7 | 7 6 7 | 7 3 9
8 4 7 | 7 8 5 | 7 4 5
4 1 6 | 6 4 8 | 4 3 4
---------------------
9 3 9 | 2 2 8 | 3 6 2
1 1 8 | 4 8 6 | 8 6 6
3 7 6 | 2 2 7 | 3 8 4

Testing walksat with rows
Time taken:  2.52937388420105 (s)
5 6 3 | 8 4 7 | 9 1 2
2 6 7 | 3 1 8 | 5 9 4
4 6 9 | 8 3 5 | 2 1 7
---------------------
7 2 6 | 3 5 1 | 9 8 4
2 1 5 | 3 8 4 | 9 6 7
7 2 4 | 8 1 9 | 5 3 6
---------------------
9 5 8 | 3 1 6 | 2 4 7
8 6 3 | 1 2 4 | 7 5 9
3 4 1 | 9 5 6 | 8 7 2
```



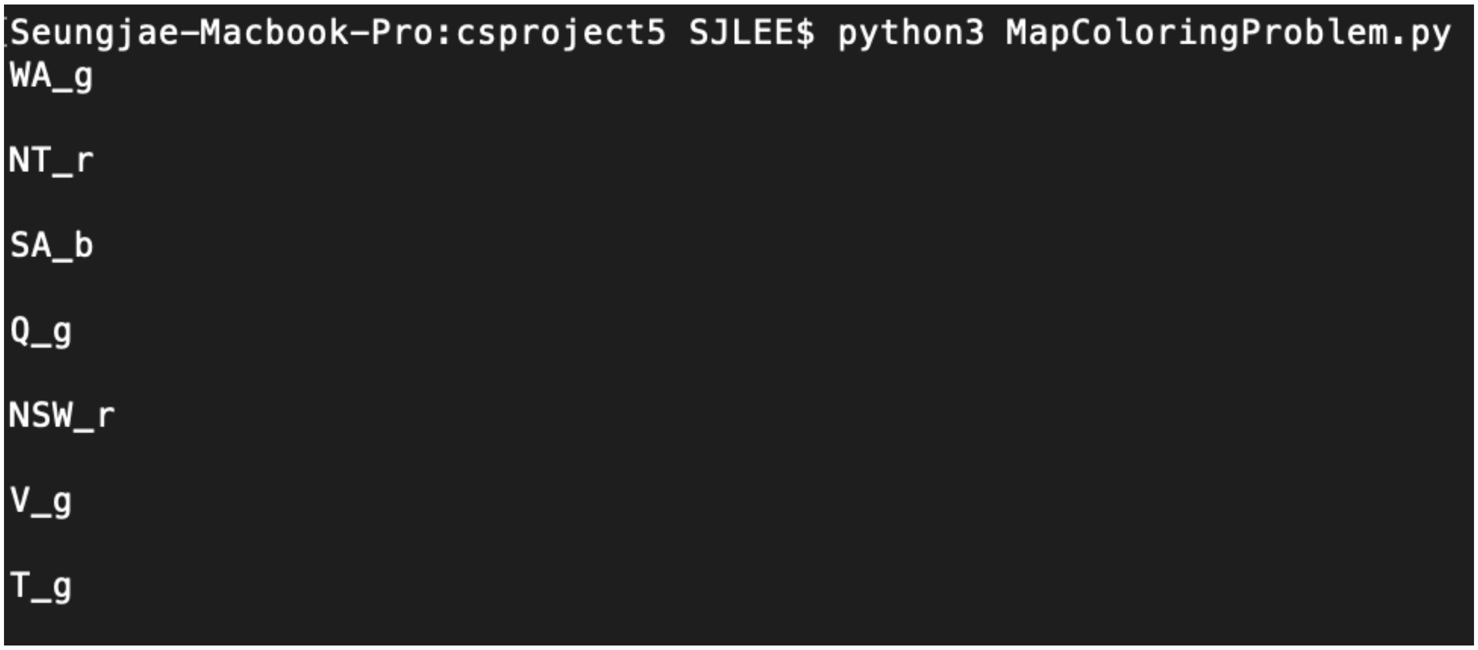
```
Testing walksat with rows_and_cols
Time taken:  14.689632892608643 (s)
7 5 2 | 1 3 4 | 6 8 9
5 7 3 | 2 9 8 | 4 6 1
9 2 6 | 4 8 1 | 5 7 3
---------------------
1 9 8 | 6 5 2 | 7 3 4
4 6 9 | 5 2 3 | 8 1 7
6 4 7 | 8 1 5 | 3 9 2
---------------------
2 1 4 | 3 7 6 | 9 5 8
8 3 5 | 9 4 7 | 1 2 6
3 8 1 | 7 6 9 | 2 4 5

Testing walksat with rows_and_cols
Time taken:  71.01684808731079 (s)
8 5 3 | 4 1 6 | 9 2 7
6 2 7 | 9 4 5 | 1 8 3
7 4 2 | 1 8 3 | 6 5 9
---------------------
2 3 8 | 7 6 9 | 4 1 5
5 8 4 | 2 3 1 | 7 9 6
3 9 6 | 5 2 7 | 8 4 1
---------------------
1 6 9 | 8 5 2 | 3 7 4
9 1 5 | 3 7 4 | 2 6 8
4 7 1 | 6 9 8 | 5 3 2
```



```
Testing walksat with puzzle1
Time taken:  57.154634952545166 (s)
5 7 6 | 2 3 1 | 8 9 4
4 1 8 | 6 9 5 | 3 2 7
2 3 9 | 7 4 8 | 5 6 1
---------------------
8 9 5 | 4 6 3 | 1 7 2
1 6 3 | 8 7 2 | 4 5 9
7 4 2 | 1 5 9 | 6 3 8
---------------------
9 5 1 | 3 8 7 | 2 4 6
3 2 4 | 9 1 6 | 7 8 5
6 8 7 | 5 2 4 | 9 1 3

Testing walksat with puzzle2
Time taken:  267.08400201797485 (s)
1 3 8 | 6 4 7 | 2 5 9
2 7 5 | 1 9 3 | 6 4 8
4 9 6 | 2 5 8 | 1 3 7
---------------------
8 5 2 | 9 6 4 | 7 1 3
3 6 9 | 8 7 1 | 5 2 4
7 4 1 | 5 3 2 | 8 9 6
---------------------
9 8 7 | 4 1 5 | 3 6 2
6 1 3 | 7 2 9 | 4 8 5
5 2 4 | 3 8 6 | 9 7 1

Testing walksat with puzzle_bonus
Time taken(no solution):  421.55521416664124 (s)
```

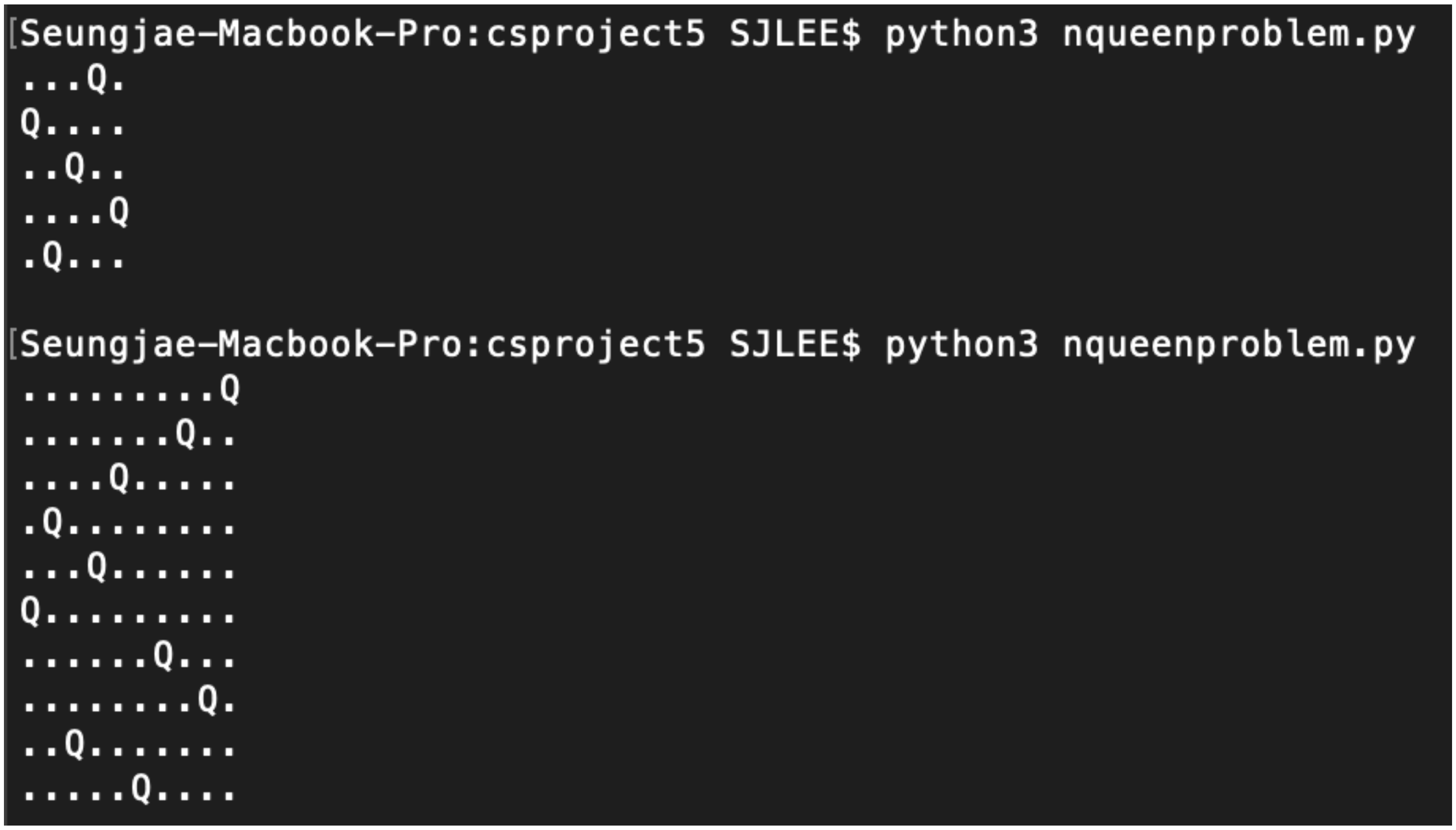
Extension

Extension 1: Map Coloring Problem I made a Map Coloring Problem class, which converts Map coloring problem into cnf. The class takes a list of locations, a list of tuples of neighboring regions, and a list of colors. `generate_cnf` method generates a cnf file with given filename. The method first creates cnf for "Every region needs to have at least one color assigned". Then, it writes cnf for "every region needs to have only color assigned." Lastly, it writes cnf for neighbors by not having overlapping colors. `display_solution` method takes a solution filename (.sol file), and reads the solution and displays the result. An example is shown below. I used the same example as in the book. (Australia map)


```
Seungjae-Macbook-Pro:csproject5 SJLEE$ python3 MapColoringProblem.py
WA_g

NT_r

SA_b

Q_g

NSW_r

V_g

T_g
```

Extension 2: N Queen Problem I made a n-queen problem class, which takes a dimension(N) as an input. Then, `generate_cnf` method finds and writes all possible cnfs for N queens so that they will not attack each other. First, the program writes cnf for columns: only one queen for column. Then, the program writes cnf for rows. Lastly, it writes cnf for diagonal, which is not as straight forward as first two. Diagonal move really depends on the position of the queen and thus I had to find mathematical pattern/formula that calculates valid diagonal move. The code below shows "diagonal cnf" part. `display_solution` reads .sol file and prints the output on terminal. An example is shown below.

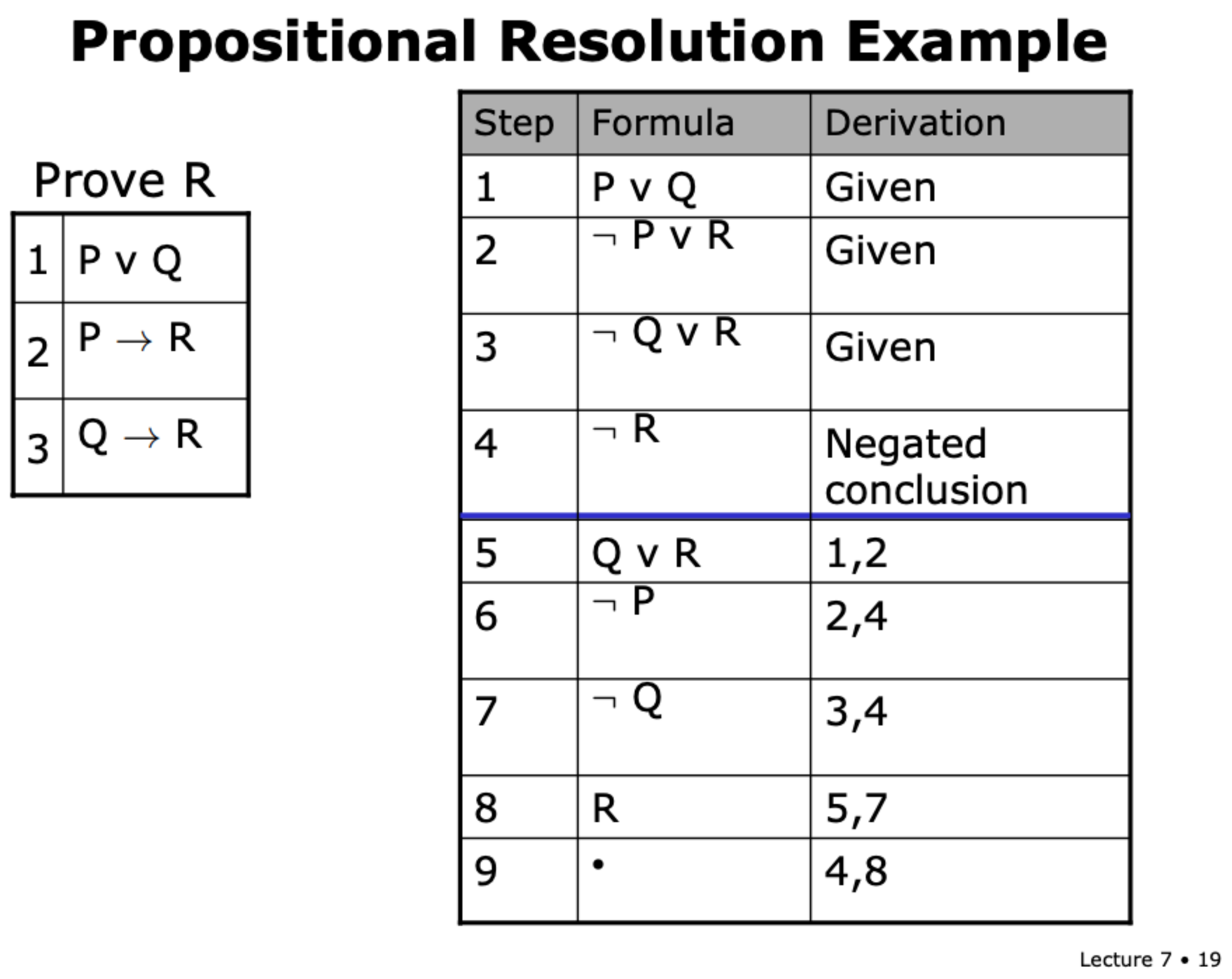
```
for i in range(self.dimension): # x coordinate
    for j in range(self.dimension): # y coordinate
         for k in range(self.dimension-1): # k+1: # of diagonal move
              if i < self.dimension-1-k and j > k: #checks if moving to right bottom
is valid
                 string +=  "-"+str(i)+str(j)+" -"+str(i+k+1)+str(j-k-1)+"\n"
              if j < self.dimension-1-k and i < self.dimension-1-k: #checks if moving
 to right top is valid
                 string +=  "-"+str(i)+str(j)+" -"+str(i+k+1)+str(j+k+1)+"\n"
```


```
[Seungjae-Macbook-Pro:csproject5 SJLEE$ python3 nqueenproblem.py
...Q.
Q....
..Q..
....Q
.Q...

[Seungjae-Macbook-Pro:csproject5 SJLEE$ python3 nqueenproblem.py
.........Q
.......Q..
....Q.....
.Q........
...Q......
Q.........
......Q...
........Q.
..Q.......
.....Q....
```

Extension 3: Implementing Resolution: In SAT class, I made a method called `resolution`. This method takes a clause (set with string variables) and returns True if the clause is True. It uses resolution rule to make new clauses out of the current clauses and negated input clause. `resolve` method takes two clauses, unifies them, and gets rid of two complementary literals if there are any. We run the code until we find a clause such that resolving the clause with negated input clause leads to empty clause. Then, it returns True. The method stops checking once there aren't any new clauses made through resolution and returns False.

Demonstration is shown below.

Extension 4: Demonstration of Resolution To demonstrate my implementation of resolution, I have two different examples. First one is very simple example I found from online. The picture below shows the problem. We have three sentences in our knowledge base, which are saved as cnf in .cnf file. Then, I tested resolution method with set(["R"]) as an input, and got True as output. The second picture shows the result.



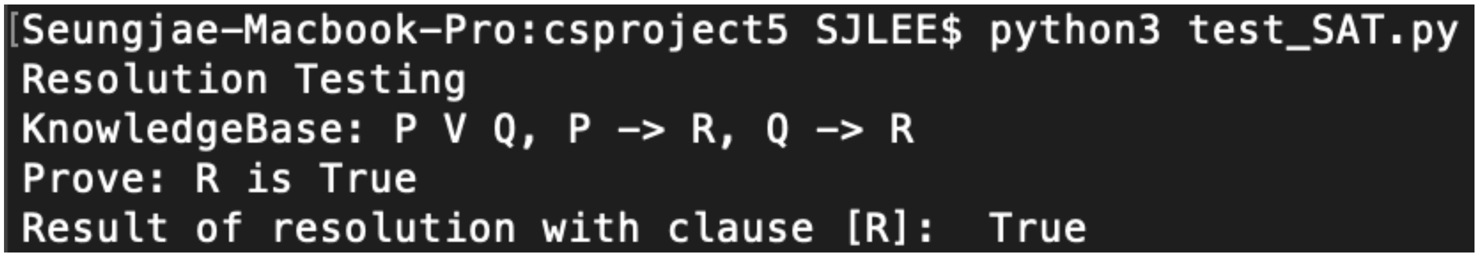
Propositional Resolution Example

Prove R

1	P v Q
2	P → R
3	Q → R

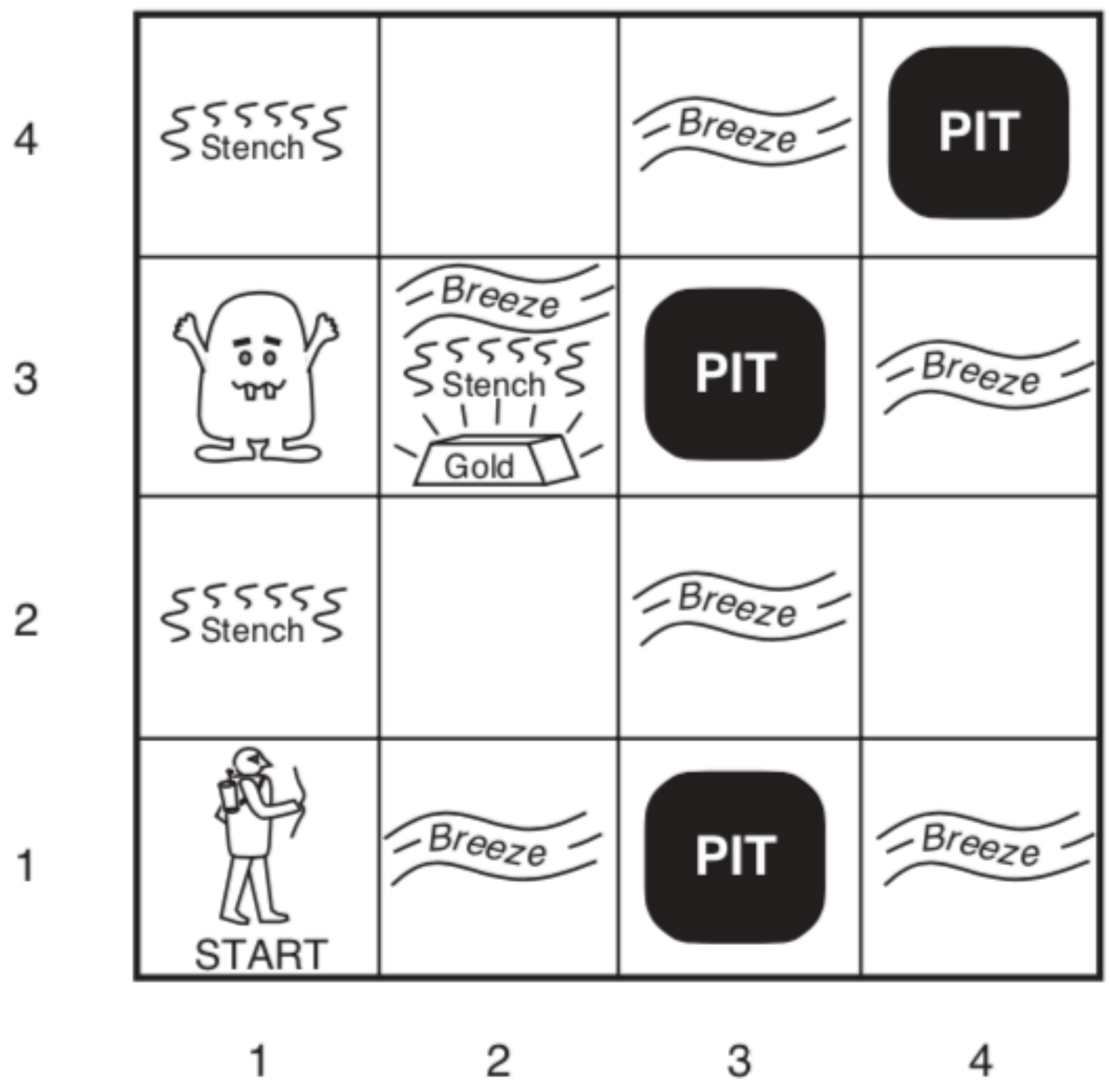
Step	Formula	Derivation
1	P v Q	Given
2	¬ P v R	Given
3	¬ Q v R	Given
4	¬ R	Negated conclusion
5	Q v R	1,2
6	¬ P	2,4
7	¬ Q	3,4
8	R	5,7
9	•	4,8

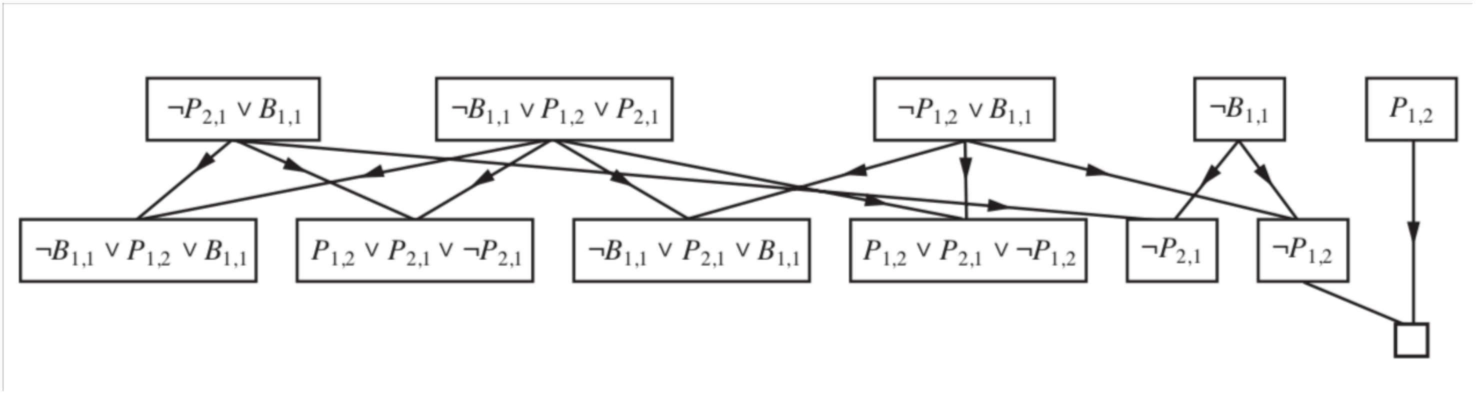
Lecture 7 • 19

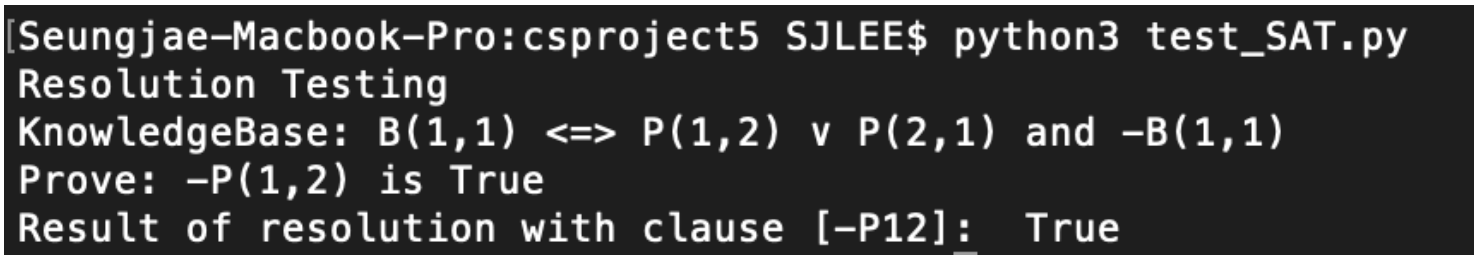


```
[Seungjae-Macbook-Pro:csproject5 SJLEE$ python3 test_SAT.py
 Resolution Testing
 KnowledgeBase: P V Q, P -> R, Q -> R
 Prove: R is True
 Result of resolution with clause [R]:  True
```

For the second example, I used a simple case of Wumpus maze demonstrated in the book. The picture of the whole map is shown below. So, for the demonstration, our main protagonist is at 0,0. We want to evaluate whether if there is not a pit at (1,2). Then, our current knowledgebase is that B(1,1) ⇔ P(1,2) ∨ P(2,1) and -B(1,1); in other words, if there is a breeze at (1,1), there should be a pit at either (1,2) and (2,1), and there is no breeze at (1,1). Using these knowledgebase converted into cnf, we test if there is no pit at (1,2). The second picture shows how resolution actually works and the last picture shows the result of running the code. (cnf saved in resolve_2.cnf)




$\neg P_{2,1} \vee B_{1,1}$
$\neg B_{1,1} \vee P_{1,2} \vee P_{2,1}$
$\neg P_{1,2} \vee B_{1,1}$
$\neg B_{1,1}$
$P_{1,2}$
$\neg B_{1,1} \vee P_{1,2} \vee B_{1,1}$
$P_{1,2} \vee P_{2,1} \vee \neg P_{2,1}$
$\neg B_{1,1} \vee P_{2,1} \vee B_{1,1}$
$P_{1,2} \vee P_{2,1} \vee \neg P_{1,2}$
$\neg P_{2,1}$
$\neg P_{1,2}$



```
Seungjae-Macbook-Pro:csproject5 SJLEE$ python3 test_SAT.py
Resolution Testing
KnowledgeBase: B(1,1) <=> P(1,2) v P(2,1) and -B(1,1)
Prove: -P(1,2) is True
Result of resolution with clause [-P12]:  True
```