San Jose State University
Department of Computer Engineering
Section 01
CMPE 130 Final Project Report

Final Project Report

Title Sudoku Puzzle Using Depth First Search Algorithm

Semester FALL 2019 **Date** 12/05/2019

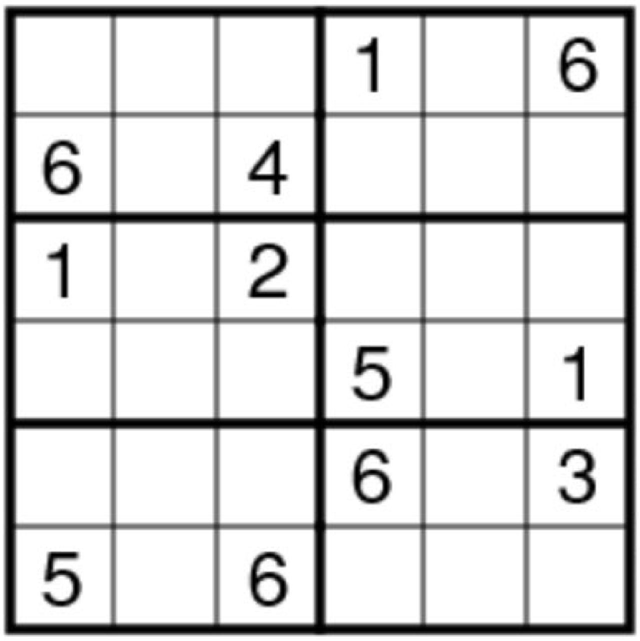
by

Name	Phat Le	**SID**	012067666
	Thien Hoang		012555673
	Manh Tuan Tran		013208468

***Abstract*— The purpose of this project is to design and build the Sudoku solver machine using the algorithm based on Depth First Search.**

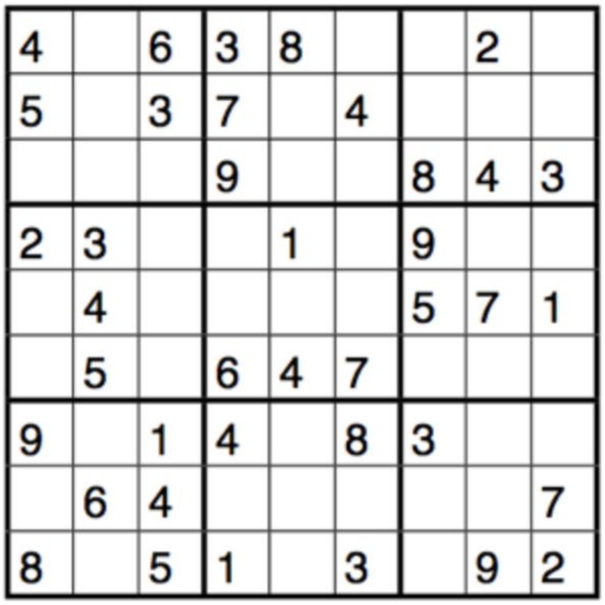
I. INTRODUCTION

Sudoku is one of the most famous puzzle games in the world. This game is a very simple that it must be filled every square in a 9x9 grid with one of the digits 1 to 9. In addition, the number cannot be filled more than twice of the same number on the same row, column, or within the same 3x3 quadrant. To solve the problem, Depth First Search algorithm(DFS) will be applied since it is suitable to find constant results. Pictures below demonstrate the sample board layout of the 6x6 sudoku puzzle and the 9x9 sudoku puzzle.



			1		6
6		4			
1		2			
			5		1
			6		3
5		6			

Figure 1: This is a 6x6 sudoku puzzle example



4		6	3	8			2	
5		3	7		4			
			9			8	4	3
2	3			1		9		
	4					5	7	1
	5		6	4	7			
9		1	4		8	3		
	6	4						7
8		5	1		3		9	2

Figure 2: This is a 9x9 sudoku puzzle example

II. DESIGN METHODOLOGY

DFS algorithm:

DFS is an algorithm of transverse to the tree or graph data structures. The method will start from the root node and will go to all possible branches to find the solution. If there is not any solution then it backtracks and goes to the other unvisited vertice. Specifically, every time a new node is visited but no solution was found, the algorithm will move back to the previous visited node and begin looking at other unvisited branches. Furthermore, the algorithm requires the use of the stack to keep track of the previously visited nodes. In this way, it can easily be backtracking to the previous node by popping the stack.

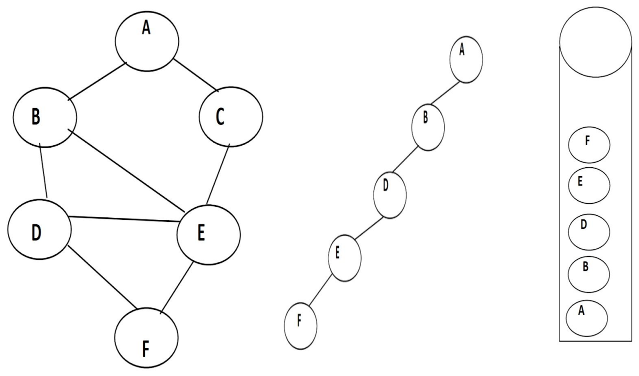


Figure 3: show the undirected graph, depth-first traversal, and push to the stack

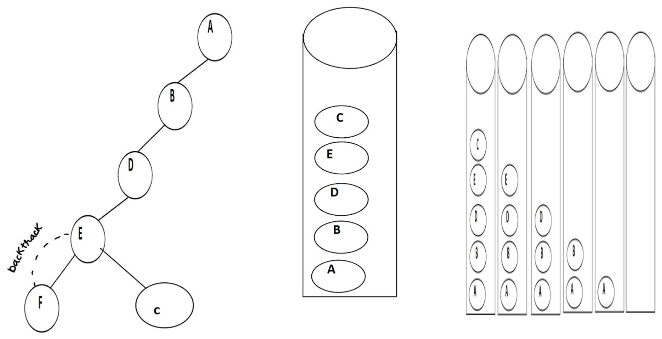


Figure 4: show the depth-first traversal backtrack, push and pop out of the stack.

Sudoku Solver Implementation:

Board object Definition:

In order to solve Sudoku, the algorithm first need to understand the characteristics of the game. The first information it needs is the type of the Sudoku game whether it is 9x9 or 6x6 since the range of distinct numbers are different. Another difference between the two types of the map is the quadrants. While 9x9 have 9 quadrants with a height of 3, 6x6 have only 6 quadrants with a height of 2. Therefore, it needs the height variable as described in *Figure 4* below.

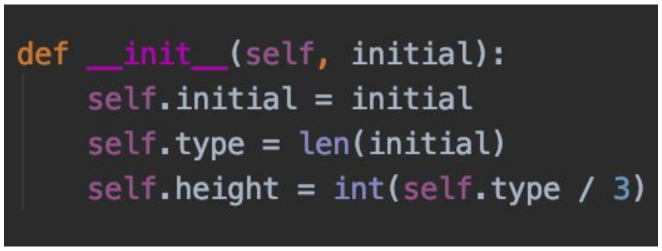

```
def __init__(self, initial):
    self.initial = initial
    self.type = len(initial)
    self.height = int(self.type / 3)
```

Figure 4: Defining characteristics of the Sudoku board.

Sudoku Board Pattern Recognition:

The algorithm would first verify and recognize the Sudoku pattern. In other words, the algorithm would return true if the given board is 9x9 or 6x6 mapped. If the number of rows is different than the number of columns or quadrants, the algorithm will return false and stop the solving process. As a result, the algorithm would become more efficient due to the fact that the process does not have to begin when needed, especially for unnecessary processes on pushing and popping the stack during DFS.

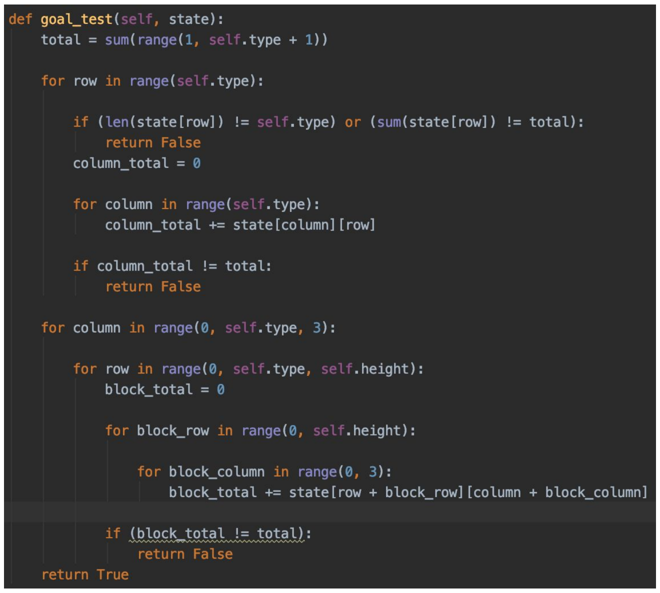
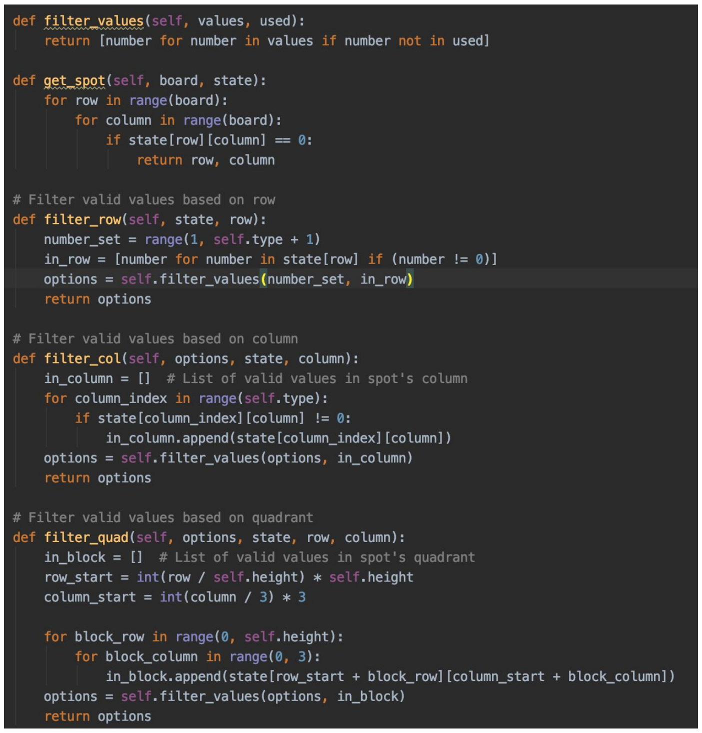

```
def goal_test(self, state):
    total = sum(range(1, self.type + 1))

    for row in range(self.type):

        if (len(state[row]) != self.type) or (sum(state[row]) != total):
            return False
        column_total = 0

        for column in range(self.type):
            column_total += state[column][row]

        if column_total != total:
            return False

    for column in range(0, self.type, 3):

        for row in range(0, self.type, self.height):
            block_total = 0

            for block_row in range(0, self.height):

                for block_column in range(0, 3):
                    block_total += state[row + block_row][column + block_column]

            if (block_total != total):
                return False
    return True
```

Figure 5: Make sure if the given board is 6x6 or 3x3.

As shown in *Figure 5* above, the program compares the expected number of rows and columns based on the given type with the actual counting of row and column, then return false if they do not match or true if the value is as expected.

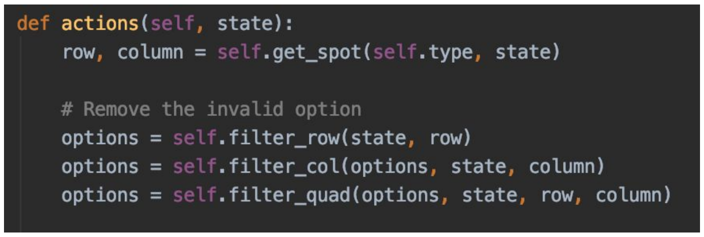
Number Filtering:

The Sudoku's main rule is not to fill the empty cell with the repeated number in the same row, column or quadrant. For this reason, the algorithm also has three different functions to filter numbers that have already appeared in the row, column and quadrants. Specifically, for each of the empty cells on the board, the algorithm would first have to go through three filtering functions to neglect the existing numbers currently on the board.



```python
def filter_values(self, values, used):
    return [number for number in values if number not in used]

def get_spot(self, board, state):
    for row in range(board):
        for column in range(board):
            if state[row][column] == 0:
                return row, column

# Filter valid values based on row
def filter_row(self, state, row):
    number_set = range(1, self.type + 1)
    in_row = [number for number in state[row] if (number != 0)]
    options = self.filter_values(number_set, in_row)
    return options

# Filter valid values based on column
def filter_col(self, options, state, column):
    in_column = []  # List of valid values in spot's column
    for column_index in range(self.type):
        if state[column_index][column] != 0:
            in_column.append(state[column_index][column])
    options = self.filter_values(options, in_column)
    return options

# Filter valid values based on quadrant
def filter_quad(self, options, state, row, column):
    in_block = []  # List of valid values in spot's quadrant
    row_start = int(row / self.height) * self.height
    column_start = int(column / 3) * 3

    for block_row in range(0, self.height):
        for block_column in range(0, 3):
            in_block.append(state[row_start + block_row][column_start + block_column])
    options = self.filter_values(options, in_block)
    return options
```

Figure 6: Three-layer of filter to find the possible number filling in the empty box.



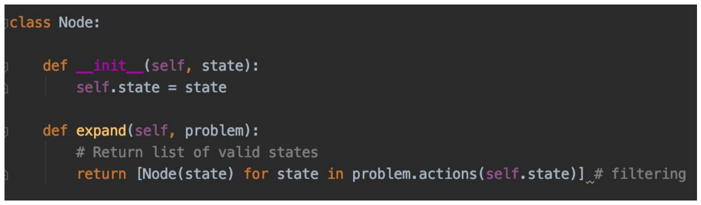
```python
def actions(self, state):
    row, column = self.get_spot(self.type, state)

    # Remove the invalid option
    options = self.filter_row(state, row)
    options = self.filter_col(options, state, column)
    options = self.filter_quad(options, state, row, column)
```

Figure 7: Three-layers of filter to find the possible number filling in the empty box.

As shown in *Figure 7* above, the algorithm finds the first empty spot and immediately filters all the invalid number that can be written to that cell.

Node:

To apply DFS, the algorithm needs to define the node object to store valid number which can be stored to the stack to further backtracking node.



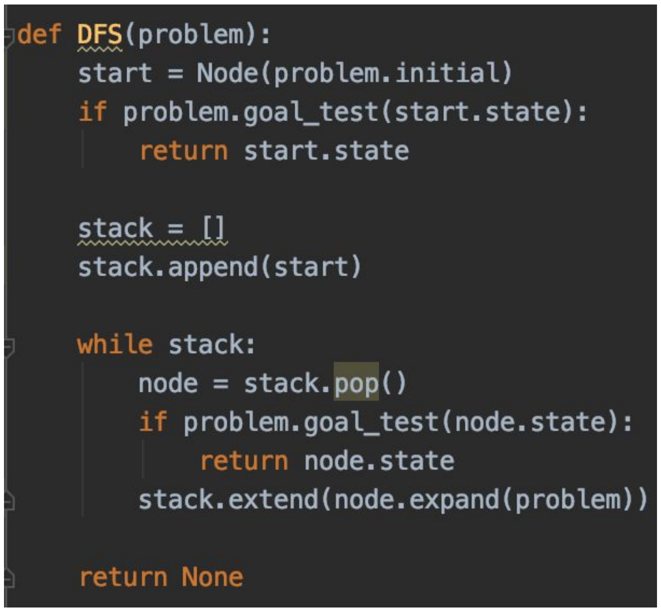
```python
class Node:

    def __init__(self, state):
        self.state = state

    def expand(self, problem):
        # Return list of valid states
        return [Node(state) for state in problem.actions(self.state)] # filtering
```

Figure 8: Node object characteristics.

Implement DFS to the solver:

The search would first choose the first empty node. Then, it starts filling in the result to the board representing the state variable. Furthermore, every visited node will be pushed to the stack. The value will be pop out of the stack if the number does not meet the requirement anymore.

Lastly, the program will stop when there is no empty stack available. Moreover, the algorithm also need to handle the case when the given problem is already violating the rule. For instance, the given numbers in the board is repeated. In this case, the algorithm will stop and return false when no node can be popped from the stack.



```
def DFS(problem):
    start = Node(problem.initial)
    if problem.goal_test(start.state):
        return start.state

    stack = []
    stack.append(start)

    while stack:
        node = stack.pop()
        if problem.goal_test(node.state):
            return node.state
        stack.extend(node.expand(problem))

    return None
```

Figure 9: DFS implementation.

III. Analysis and Results

DFS Complexity

DFS time complexity is O(n+m). Where n is the number of vertices of edges in the graph. However, O(m) may vary between O(1) and $O(n^2)$, depending on how dense the graph is.

Solver Complexity

For the Sudoku problem, the complexity also depends on the number of empty cells and the type of board (6x6 vs 9x9). According to the *Table 1 below,* it can easily be observed that the complexity increases as the number of empty cells increase. Furthermore, the bigger the map, the longer the time the algorithm can find the solution.

	6x6	9x9
Unsolvable	0.02198600769	0.6849379539
Hardest	0.01053786278	2.115872145
Easiest	0.001165151596	0.007792711258
Filled	2.98E-05	3.41E-05

Table 1: The comparison elapsed time between 6x6 and 9x9 board.

As shown in *Figure 10* below, the process time for 9x9 is significantly greater than 6x6. Especially for the hardest case, the 9x9 board process time is almost 99% bigger compared to 6x6.

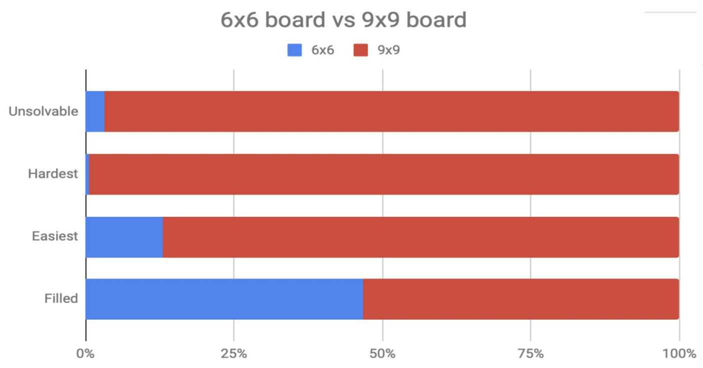


Figure 10: The percentage between two board types.

IV. Conclusion

Overall, the solver algorithm were successful since it was able to find solutions for both the easiest to the hardest cases as well as different type of board. Timing is one of the most challenging that we have to face while we are working on the project. Since everyone in the group has a different schedule. As first the puzzle was not working properly due to the difficulty in detecting the type of board as well as dealing with corner cases such as when the board layout is violating the rule. After several times of debugging and redefining the function, we were able to get the solver to work the way we expect.

Last but not least, Our group would like to thank you Professor Gokay for giving us a challenging but fun semester. Thank you Professor for pushing us to meet new friends as well as embracing the knowledge of algorithm and the discipline in the future programming career.

V. References

Awerbuch, B. (1985). A new distributed depth-first-search algorithm. *Information Processing Letters*, *20*(3), 147-150.

Provan, J. S. (2009). Sudoku: strategy versus structure. *The American Mathematical Monthly*, *116*(8), 702-707.

Sampe Sudoku difficulty levels: https://www.sudokuconquest.com/9x9/easy