

**CSE2003: DATA STRUCTURES AND ALGORITHMS
PROJECT REVIEW – 3**

INSTRUCTOR: PROF. ASHWIN GANESAN

PROJECT NAME: SUDOKU PUZZLES

TEAM MEMBERS:

18BCE0265 ISHAAN OHRI

18BCE0283 ANMOL PANT

INTRODUCTION:

WHAT IS SUDOKU PUZZLE?

A Sudoku puzzle is defined as a logic-based, number-placement puzzle. The objective is to fill a 9×9 grid with digits in such a way that the following three conditions are always fulfilled:

- i. Each row must always contain all digits from 1 – 9.
- ii. Each column must always contain all digits from 1 – 9.
- iii. Each 3×3 grid must always contain all digits from 1 – 9.

HOW TO BEGIN?

Each Sudoku puzzle begins with some cells filled in. The player uses these seed numbers as a launching point toward finding the unique solution.

RULES FOR SOLVING A SUDOKU PUZZLE

While solving Sudoku puzzles can be significant challenge, the rules for traditional solution finding are quite straight forward:

- i. Each row, column, and nonet can contain each number (typically 1 to 9) exactly once.
- ii. The sum of all numbers in any nonet, row, or column must match the small number printed in its corner. For traditional Sudoku puzzles featuring the numbers 1 to 9, this sum is equal to 45.

ABSTRACT:

In this project we solve Sudoku puzzles using backtracking method

WAYS OF SOLVING THE SUDOKU PUZZLE:

WHAT ARE THE DIFFERENT WAYS OF SOLVING A SUDOKU PUZZLE?

A Sudoku puzzle can be solved by the following two methods:

- i. Backtracking.
- ii. Constraint propagation.

BACKTRACKING

- Backtracking is an algorithmic-technique for solving problems recursively by trying to build a solution incrementally.
- It involves removing those solutions (partial candidates) that fail to satisfy the constraints of the problem at any point of time.
- These partial candidates are represented as the nodes of a search tree.
- The backtracking algorithm traverses this search tree recursively, from the root down, in depth-first order.
- Backtracking is an important tool for solving constraint satisfaction problems, such as crosswords, verbal arithmetic, Sudoku etc.

CONSTRAINT PROPAGATION

- Constraint propagation is an algorithm that solves Sudoku problems by using possibility array and multiple iterations.
- We will find all the possible numbers for a particular blank location and we will have an array for each blank location
- If for a particular blank location there is only one possibility then we will directly insert the value in that location
- In the other case we will have multiple iterations following to complete the Sudoku problem

EXAMPLE OF SOLVING A SUDOKU PUZZLE:

For solving the puzzle we will have to follow a set of rules. They are:

- i. Assign numbers one by one to empty cell.
- ii. Before assigning we check if it is safe or not.

- iii. If in a cell there is no other possibility then we go to the previous cell, empty it and then try going for some other possibility.
- iv. We repeat the above three steps recursively till we get the solution to the puzzle.

Given is a to be solved Sudoku Puzzle:

1		3	
		2	1
	1		2
2	4		

STEP 1:

For cell two:

We cannot place 1, hence we place 2.

1	2	3	
		2	1
	1		2
2	4		

STEP 2:

For cell four:

We cannot place any numbers from 1 to 3, hence we place 4.

1	2	3	4
		2	1
	1		2
2	4		

STEP 3:

For cell five:

We cannot place either of 1 or 2, hence we place 3.

1	2	3	4
3		2	1
	1		2
2	4		

STEP 4:

For cell six:

There is no possibility of filling any numbers so we back to cell five, remove 3 and fill it with its next possibility i.e.

1	2	3	4
3	1	2	1
	1		2
2	4		

STEP 5:

For cell five:

We fill 4 which is the next possibility after 3.

1	2	3	4
4		2	1
	1		2
2	4		

STEP 6:

For cell six:

We place 3, which is the only possibility.

1	2	3	4
4	3	2	1
	1		2
2	4		

STEP 7:

For cell nine:

We place 3, which is the only possibility.

1	2	3	4
4	3	2	1
3	1		2
2	4		

STEP 8:

For cell eleven:

We cannot place 1 to 3, hence we place 4.

1	2	3	4
4	3	2	1
3	1	4	2
2	4		

STEP 9:

For cell fifteen:

We place 1.

1	2	3	4
4	3	2	1
3	1	4	2
2	4	1	

STEP 10:

For cell sixteen:

We cannot place either of 1 or 2, therefore we place 3.

1	2	3	4
4	3	2	1
3	1	4	2
2	4	1	3

Final Solution:

1		3	
		2	1
	1		2
2	4		



1	2	3	4
4	3	2	1
3	1	4	2
2	4	1	3

ALGORITHM:

- i. Input is taken in the form of 9 rows.
- ii. Every row of input consists of numbers separated by a space.
- iii. Every row is appended to the list sudoku.
- iv. Call the function solveSudoku().
- v. Find the empty location in the grid by calling function findEmptyLocation().
- vi. Use a loop to generate numbers from 1 to 9.
- vii. Call a function check() to check if the number is safe to enter or not.
- viii. Inside the function check(), the rows, columns and the sub grid is checked for any repetition using the function usedInRow(), usedInCol(), usedInMiniGrid() respectively.
- ix. If the number assigned satisfies all the three conditions, then it is assigned to that cell.
- x. If the Sudoku is solved, then print the Sudoku using printGrid() function else print "No solution exists".

CODE (Python 3.6):

```
#TO PRINT SUDOKU
def printGrid(A):
    print("\nSOLUTION IS")
    for i in range(9):
        print(" ".join(str(x) for x in A[i]))

#TO FIND ALL EMPTY LOCATIONS MARKED BY
'0' def findEmptyLocation(A,l):
    for row in range(9):
        for col in range(9):
            if(A[row][col]==0):
                l[0]=row
                l[1]=col
                return True
    return False

#CHECKS WHETHER NUMBER ALREADY EXISTS IN ROW
def usedInRow(A,row,num):
    for i in range(9):
        if(A[row][i] == num):
            return True
    return False

#CHECKS WHETHER NUMBER ALREADY EXISTS IN
COLUMN def usedInCol(A,col,num):
    for i in range(9):
```

```

        if(A[i][col] == num):
            return True
    return False

#CHECKS WHETHER NUMBER ALREADY EXISTS IN MINIGRID
def usedInMiniGrid(A,row,col,num):
    for i in range(3):
        for j in range(3):
            if(A[i+row][j+col] == num):
                return True
    return False

#CHECKS IF A LOCATION IS SAFE
def check(A,row,col,num):

    return not usedInRow(A,row,num) and not
    usedInCol(A,col,num) and not usedInMiniGrid(A,row -
    row%3,col - col%3,num)

#MAIN SOLVING FUNCTION
def solveSudoku(A):

    l=[0,0]

    if(not findEmptyLocation(A,l)):
        return True

    row=l[0]
    col=l[1]

    for num in range(1,10):

        if(check(A,row,col,num)):

            A[row][col]=num

            if(solveSudoku(A)):
                return True

            A[row][col] = 0

    return False

sudoku = []

for i in range(9):
    temp = list(map(int,input().split()))
    sudoku.append(temp)

```

```
#IF SUCCESS THEN PRINT SUDOKU
if(solveSudoku(sudoku)):
    printGrid(sudoku)
else:
    print ("No solution exists")
```

TESTING PLAN:

Input:

```
306508400
520000000
087000031
003010080
900863005
050090600
130000250
000000074
005206300
```

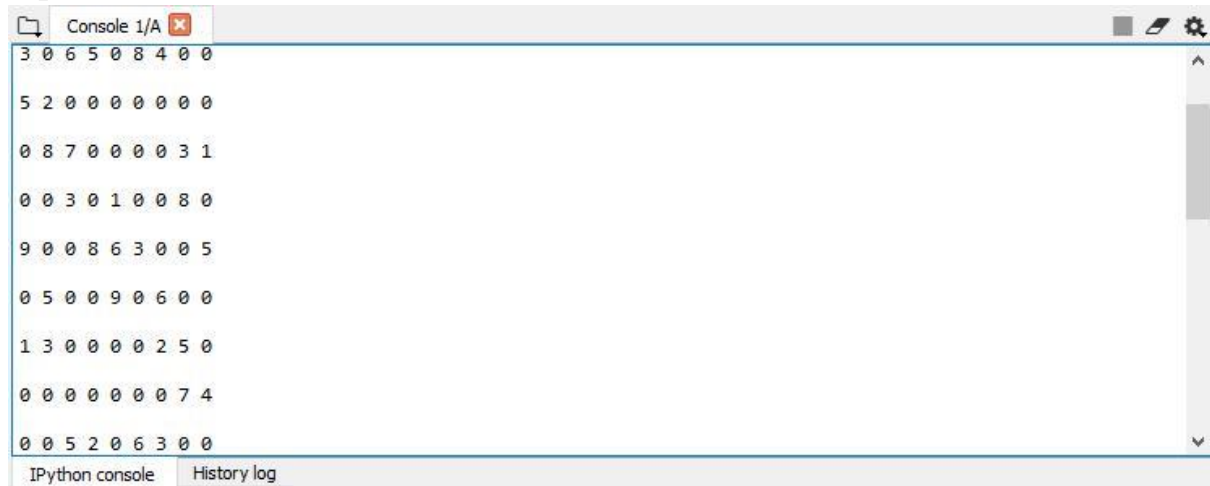
Output:

SOLUTION IS:

```
316578492
529134768
487629531
263415987
974863125
851792643
138947256
692351874
745286319
```

INPUT/OUTPUT (Using Spyder):

Input:



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

Output:



```
SOLUTION IS
3 1 6 5 7 8 4 9 2
5 2 9 1 3 4 7 6 8
4 8 7 6 2 9 5 3 1
2 6 3 4 1 5 9 8 7
9 7 4 8 6 3 1 2 5
8 5 1 7 9 2 6 4 3
1 3 8 9 4 7 2 5 6
6 9 2 3 5 1 8 7 4
7 4 5 2 8 6 3 1 9

In [17]: |
```

ALGORITHM ANALYSIS:

Backtracking:

- i. Backtracking is an optimization technique.
- ii. We start with a possible solution which satisfies all the required conditions, then we move to the next level and if that does not produces satisfactory solution, we return back to the previous level and start with a new option.
- iii. Example: Sudoku solver, going through mare, eight queen problem
- iv. The backtracking algorithm enumerates a set of partial candidates that, in principle, could be completed in various ways to give all the possible solutions to the given problem.

- v. The completion is done incrementally, by a sequence of candidate extension steps.
- vi. Conceptually, the partial candidates are represented as the nodes of a tree structure, the potential search tree.
- vii. Each partial candidate is the parent of the candidates that differ from it by a single extension step; the leaves of the tree are the partial candidates that cannot be extended any further.
- viii. The backtracking algorithm traverses this search tree recursively, from the root in depth-first order.
- ix. At each node x , the algorithm checks whether x can be completed to a valid solution.
- x. If it cannot, the whole sub-tree rooted at x is skipped.
- xi. Otherwise, the algorithm checks whether x itself is a valid solution, and if so reports it to the user and recursively enumerates all sub-trees of x .
- xii. The two tests and the children of each node are defined by user-given procedures.

Divide and Conquer:

- i. Divide and Conquer is an algorithmic paradigm.
- ii. A typical Divide and Conquer algorithm solves a problem using following three steps.
 - a. Divide: Break the given problem into sub problems of same type.
 - b. Conquer: Recursively solve these sub problems.
 - c. Combine: Appropriately combine the answers

ADVANTAGE OF DEPTH FIRST SEARCH

- i. The advantage of depth-first Search is that memory requirement is only linear with respect to the search graph.
- ii. This is in contrast with breadth-first search which requires more space.
- iii. The reason is that the algorithm only needs to store a stack of nodes on the path from the root to the current node.
- iv. Practically depth first search is time-limited rather than space-limited.
- v. If depth-first search finds solution without exploring much in a path then the time and space it takes will be very less.

DISADVANTAGE OF DEPTH FIRST SEARCH

- i. The disadvantage of Depth-First Search is that there is a possibility that it may go down the left-most path forever. Even a finite graph can generate an infinite tree.

- ii. One solution to this problem is to impose a cut off depth on the search.
- iii. Although the ideal cut off is the solution depth d and this value is rarely known in advance of actually solving the problem. If the chosen cut off depth is less than d , the algorithm will fail to find a solution, whereas if the cut off depth is greater than d , a large price is paid in execution time, and the first solution found may not be an optimal one.
