GE103 Project Report

*Sudoku Solver*



*Aayush Sabharwal*

*Aditya Patel*

# Table of Contents

Table of Contents

[Acknowledgements](#_Toc15067_WPSOffice_Level1) [2](#_Toc15067_WPSOffice_Level1)

[Introduction](#_Toc24770_WPSOffice_Level1) [3](#_Toc24770_WPSOffice_Level1)

[Aim](#_Toc24770_WPSOffice_Level2) [3](#_Toc24770_WPSOffice_Level2)

[What is Sudoku?](#_Toc5980_WPSOffice_Level2) [3](#_Toc5980_WPSOffice_Level2)

[Our approach to solving the problem](#_Toc5980_WPSOffice_Level1) [3](#_Toc5980_WPSOffice_Level1)

[Optimisation of brute-force method](#_Toc7227_WPSOffice_Level2) [3](#_Toc7227_WPSOffice_Level2)

[Code](#_Toc24770_WPSOffice_Level3) [4](#_Toc24770_WPSOffice_Level3)

[An alternative idea](#_Toc32012_WPSOffice_Level2) [5](#_Toc32012_WPSOffice_Level2)

[Code](#_Toc5980_WPSOffice_Level3) [5](#_Toc5980_WPSOffice_Level3)

[Solution Checker](#_Toc7227_WPSOffice_Level1) [9](#_Toc7227_WPSOffice_Level1)

[Bibliography](#_Toc32012_WPSOffice_Level1) [10](#_Toc32012_WPSOffice_Level1)

Acknowledgements

I would like to express my gratitude towards all those who made this project a possibility. First and foremost, I would like to thank Prof. Sudarshan Iyengar who through his extensive guidance and motivation enabled me to take up this project. I would also like to thank all the TAs who provided constant assistance whenever required.

# Introduction

## Aim

This project aims to create a python program which can take as input an unsolved sudoku grid, and print the solved grid. This report aims to illustrate our approach to solving the problem, including our logic and code samples.

## What is Sudoku?

Before creating a program to play the game, we needed to familiarize ourselves with it. Sudoku is a game played on a 9x9 grid which is composed of 3x3 sub-grid (hereon referred to as blocks) arranged in a 3x3 pattern. Each cell in the grid can take a single integer value from 1 to 9, both inclusive. Some cells are pre-filled in the grid. The game aims to fill the remaining cells in a manner such that no number appears twice in its row, column or block. The difficulty of a particular problem generally varies with the number of pre-filled cells. With more cells filled beforehand, the difficulty is drastically reduced as a larger number of possible values for each cell are ruled out. As the number of pre-filled cells increases, the problem becomes more difficult and requires more extensive analytical techniques to be used.

# Our approach to solving the problem

The most obvious solution to any problem is brute force. That is, trying every solution to find the one that satisfies all the given conditions. It is easy to see that this is practically impossible for sudoku, since each cell can have up to 9 possible values, and there are 81 total cells. Considering a starting grid with half the cells filled (this is a fairly optimistic assumption since most grids are significantly harder) the total number of possibilities totals to 940 which is of the order of 1038. This is too hard to brute-force even for an easy grid.

## Optimisation of brute-force method

Our line of thought then progressed to methods of optimising the brute-force algorithm. We can see that checking whether assigning a particular number to a cell is valid is a constant-time operation since there are a fixed number of cells to check. Each empty cell will have a finite number of possible values at the beginning of each puzzle, and only one of these values is its final solution (assuming that the grid has only one possible solution). Hence, we can sequentially try putting values in cells, testing whether it is valid, and moving to the next empty cell if it is. If a value is invalid for a particular cell, we test the next one until we either come across a valid value or exhaust the list. If the list is exhausted, it means that one of our previous assumptions must be incorrect. Hence, in such a situation, we go to the previous cell and try out its next possible value. This proceeds until either we return to the first cell and exhaust all its values (the case where the grid cannot be solved), or we reach the last cell and assign it its only value (the case where the grid has one solution), or we reach the last cell and have more than one valid options to fill it (the case where the grid has multiple solutions). This is a simple modification of the depth-first-search algorithm in graph theory. Each node is a state of the grid with edges representing values being assigned to the next cell in the grid. It may be interesting to note that the resultant graph should form a tree. The root node will be the initial, unsolved grid and each branch outward from it representing a distinct way of filling the first empty cell, and each subsequent branch outward being likewise a distinct way of filling the next cell. The DFS algorithm runs in O(V+E) time complexity. Since the graph is a tree, E = V-1 and time complexity becomes O(2V-1)=O(V). However, one aspect to note is that this assumes we need to traverse the entire tree to arrive at our solution. This is not the case since we stop once we have reached the end. It may also be worth noting that for every time we assign a cell a particular number, the possible valid values for the other blank cells in its row, column and block decrease by one. Hence, on average, the number of children per node decreases the deeper we go into the tree. The worst-case appears for an unsolvable grid when we try all possibilities.

### Code

|  |
| --- |
| # returns list containing elements in given column of grid g def col(x, g):  return [g[r][x] for r in range(9)] # return list of all elements at index i in each row   # returns list containing elements in given box of grid g, box indexes are 0-indexed from top left def block(r, c, g):  ret = []  for rr in range(3):  for cc in range(3):  ret.append(g[r \* 3 + rr][c \* 3 + cc])  return ret   # check if it is possible to put i in (r, c) cell of grid g def is\_possible(r, c, g, i):  if i in g[r]: # if i exists in the row  return False   if i in col(c, g): # or in the column  return False   if i in block(r // 3, c // 3, g): # or in the block  return False # it isn't possible to put it here   return True # it isn't in row/col/block so it is possible for i to be here   # prints grid def pretty\_print\_grid(g):  for i in range(9):  for j in range(9):  print(g[i][j], end=' ')  print()   # given an unsolved grid g and a list of coordinates of empty cells, this function returns whether grid is solvable # and solves it def solve(g, empty):  if len(empty) == 0: # if we have no empty cells, we have assigned every cell a valid value  pretty\_print\_grid(g) # print the finished grid  return True # this grid is solvable   for i in range(1, 10): # iterating through 1 to 9  if is\_possible(empty[0][0], empty[0][1], g, i): # if it is possible for i to be in this position  g[empty[0][0]][empty[0][1]] = i # assign it here  if solve(g, empty[1:]): # if solving the resultant grid with this one less empty cell is possible  return True # this grid is solvable  g[empty[0][0]][empty[0][1]] = 0 # making cell empty again, in case it was assigned before but didn't lead to  # a solution   return False # if we reached here, we have tried all the numbers from 1 to 9 and none resulted in a solution,  # so the grid is unsolvable in its current state   grid = [] print("Enter 9x9 unsolved grid. Write 0 for a blank cell. Cells should be space separated") for i in range(9):  grid.append([int(x) for x in input().split()])  # to store coordinates of all empty cells in grid empty = [] for row in range(9):  for column in range(9):  if grid[row][column] == 0:  empty.append((row, column))  ans = solve(grid, empty)  if not ans: # if we couldn't solve the grid  print("Grid is unsolvable") |

## An alternative idea

A simple google search for the term “sudoku techniques” reveals a comprehensive list of various techniques for solving sudoku puzzles. The next idea we had for solving a puzzle is to write a code that can perform as many as possible of the analytical checks, and thus narrow down the possibilities for all the cells. In the case where a cell has only one possibility, it then takes that value. Hence, we should be able to solve the grid using this method.

Finding/updating possible values for each cell takes constant time, as mentioned previously. Hence, for the entire grid, this takes O(n) time, for n empty cells. We aim to perform a series of analytical tests, narrowing down the possibilities until at least one cell has only one possible value. In such a case, we assign it that value and re-update the possibilities for each cell. Since in one iteration of our tests we want to assign a minimum of one cell its final value, the grid should be solved in n iterations. Each iteration takes O(n) time to update possibilities and O(n) time for each test. Hence the resultant time complexity is O(n\*n)=O(n2). This is a drastic improvement.

A major challenge arises in performing each test, and in an order that makes sense. Simple tests such as ruling out numbers already occurring in the same row/column/block are easy to implement. Other, more complicated tests are considerably harder such as checking if a cell is the only one that can take a particular value in its row/column/block and assigning it that value. Hence, this algorithm can only solve grids that can be solved with the analytical techniques implemented. In other words, it is not possible to solve all grids but the grids that can be solved are solved faster than the brute-force optimisation algorithm.

### Code

|  |
| --- |
| grid = []  # sudoku grid  possibles = {}  # maps tuple(r, c) to list of possibilities of that cell  invalid\_grid = False  # keeps track of whether our current grid is invalid (unsolvable or no distinct solution)  assigned\_cell = False  # keeps track of if a cell got it final value this iteration. Updated every iteration  # returns list of xth column elements in grid, x is 0-indexed  def col(x):     return [grid[r][x] for r in range(9)]  # return list of all elements at index i in each row  # returns list containing elements in given box, box indexes are 0-indexed from top left  def block(r, c):     ret = []     for rr in range(3):         for cc in range(3):             ret.append(grid[r \* 3 + rr][c \* 3 + cc])     return ret  # returns a list of possible entries in given cell  def find\_possibles(r, c):     global assigned\_cell     global invalid\_grid     if grid[r][c] != 0:  # we should be checking for possibilities in an empty cell         print("ERR: Non-empty cell check (", r, ',', c, ')')         return     current = [1, 2, 3, 4, 5, 6, 7, 8, 9]  # all the possibilities     for num in grid[r]:  # checking against elements in row         if num in current:             current.remove(num)     for num in col(c):  # checking against elements in column         if num in current:             current.remove(num)     for num in block(r // 3, c // 3):  # checking against elements in block, r//3,c//3 gives block indices         if num in current:             current.remove(num)     possibles[(r, c)] = current  # assigning the possibility of given cell in dict     if len(current) == 0:  # if there are no possibilities, we have an invalid grid         invalid\_grid = True  # checks if a cell is the only one that can possibly take a value required in its row/col/block  def check\_only\_possible(r, c):     global assigned\_cell     row\_leftovers = []  # elements we need in this row     for i in range(9):         if grid[r][i] == 0 and i != c:             row\_leftovers.extend(possibles[(r, i)])     col\_leftovers = []  # elements we need in this column     for j in range(9):         if grid[j][c] == 0 and j != r:             col\_leftovers.extend(possibles[(j, c)])     block\_leftovers = []  # elements we need in this block     for i in range(r // 3 \* 3, r // 3 \* 3 + 3):         for j in range(c // 3 \* 3, c // 3 \* 3 + 3):             if grid[i][j] == 0 and (i, j) != (r, c):                 block\_leftovers.extend(possibles[(i, j)])     for poss in possibles[(r, c)]:  # considering each value the current cell can take         if poss not in row\_leftovers:   # if this value cannot be taken by any other cell in the row             grid[r][c] = poss   # current cell takes this value             del possibles[(r, c)]             assigned\_cell = True             break         if poss not in col\_leftovers:   # similarly for column             grid[r][c] = poss             del possibles[(r, c)]             assigned\_cell = True             break         if poss not in block\_leftovers: # similarly for block             grid[r][c] = poss             del possibles[(r, c)]             assigned\_cell = True             break  # returns whether grid is unsolved i.e. contains a 0  def is\_solved():     for i in range(9):         for j in range(9):             if grid[i][j] == 0:                 return False     return True  # checks entire grid for cells with only one possibility and assigns it  def check\_singletons():     global assigned\_cell     global invalid\_grid     for i in range(9):         for j in range(9):             if grid[i][j] == 0 and len(possibles[(i, j)]) == 1:                 grid[i][j] = possibles[(i, j)][0]                 del possibles[(i, j)]                 assigned\_cell = True             elif grid[i][j] == 0 and len(possibles[(i, j)]) == 0:                 invalid\_grid = True                 break         if invalid\_grid:             break  print("Enter 9x9 unsolved grid. Write 0 for a blank cell. Cells should be space separated")  for i in range(9):  # taking input     grid.append([int(x) for x in input().split()])  niters = 0  # to keep track of number of times the loop has run, to prevent infinite loops  while not is\_solved() and niters < 200:  # while our grid isn't solved     niters += 1     for i in range(9):  # finding possibilities for all 0s         for j in range(9):             if grid[i][j] == 0:  # if the cell is empty                 find\_possibles(i, j)  # find its possibilities     check\_singletons()  # check if any cells can take only one value     if invalid\_grid:  # if the grid turns out to be invalid         print("ERR: Invalid grid")  # send an error message         break  # stop trying to solve it     if assigned\_cell:  # if we assigned a cell its value         assigned\_cell = False  # reset         continue  # reiterate, since we need to update the possibilities of all other cells before further checks     for i in range(9):  # checking all cells         for j in range(9):             if grid[i][j] == 0:                 # we didn't assign a value based off of it having only one possibility, but                 # it may be the only one which can take a given value in row/col/block                 check\_only\_possible(i, j)     if invalid\_grid:         print("ERR: Invalid grid")         break     if assigned\_cell:  # if we assigned a cell its value         assigned\_cell = False  # reset         continue  # reiterate, since we need to update the possibilities of all other cells before further checks  for row in grid:     for cell in row:         print(cell, end=' ')     print() |

# Solution Checker

Apart from the programs to solve a sudoku grid, we needed one to check solutions outputted by our code, since verifying by hand is a very tedious task and unfeasible for a large number of test cases. The following simple code takes an unsolved grid and its proposed solution, and verifies the solution by checking if a number has any duplicates in its row, column of block.

|  |
| --- |
| # returns list of xth column elements in grid but makes rr, cc element -1, x is 0-indexed def colm(rr, cc, grid):  return [grid[r][cc] if r != rr else -1 for r in range(9)] # return list of all elements at index i in each row   # returns list containing elements in given box but makes i, j element -1, box indexes are 0-indexed from top left def blockm(r, c, grid, i, j):  ret = []  for rr in range(3):  for cc in range(3):  ret.append(grid[r \* 3 + rr][c \* 3 + cc] if (r \* 3 + rr, c \* 3 + cc) != (i, j) else -1)  return ret   unsol\_grid = [] print("Enter 9x9 unsolved grid. Write 0 for a blank cell. Cells should be space separated") for i in range(9): # taking input  unsol\_grid.append([int(x) for x in input().split()])  sol\_grid = [] print("Enter 9x9 solved grid. Write 0 for a blank cell. Cells should be space separated")  for i in range(9): # taking input  sol\_grid.append([int(x) for x in input().split()])  solved = True for i in range(9):  for j in range(9):  # if there is a cell where value was given but it is not the same in the answer  if unsol\_grid[i][j] != 0 and unsol\_grid[i][j] != sol\_grid[i][j]:  print("ERR: Changed grid at (", i, ',', j, ')') # print error message  solved = False # the grid isn't solved   if (sol\_grid[i][j] in sol\_grid[i][0:j] + sol\_grid[i][  j + 1:] # if the current cell value is duplicate in its row  or sol\_grid[i][j] in colm(i, j, sol\_grid) # or it is duplicate in its column  or sol\_grid in blockm(i // 3, j // 3, sol\_grid, i, j)): # or it is duplicate in its block  print("ERR: Repetition at (", i, ',', j, ')') # print error message  solved = False # the grid isn't solved   if not solved:  break  if not solved:  break  if solved:  print("SOLVED") |

# Bibliography

1. <https://www.conceptispuzzles.com/index.aspx?uri=puzzle/sudoku/techniques> - A fairly comprehensive list of commmon sudoku solving techniques
2. <https://sudoku.com/> - A sudoku puzzle generator used to test our programs