

REPORT OF SUDOKU USING BACKTRACKING

As a project work for Course

ARTIFICIAL INTELLIGENCE (INT 404)

Name : Saurabh Santosh

Registration Number : 11809739 Roll no:-13

Name : Bhanu pratap

Registration Number : 11803008 Roll no.:-22

Program : CSE B.Tech

School : School of Computer Science

Name of the University : Lovely Professional University

Date of submission : 31st March 2020



ACKNOWLEDGEMENT

The success and final outcome of this project required a lot of guidance and assistance from many people and we are extremely privileged to have got this all along the completion of my project. All that we have done is only due to such supervision and assistance and we would not forget to thank them.

We respect and thank Prof. Sagar Pande for providing us an opportunity to do the project work and giving us all support and guidance which made us complete the project duly. We are extremely thankful sir for providing such a nice support and guidance, although he had busy schedule.

CONTEXT

This project has been done as part of my course for the INT404 at Lovely Professional University . Supervised by Prof. Sagar Pande, We had three months to fulfill the requirements in order to succeed the module.

INTRODUCTION

SUDOKU:-

Sudoku is a number-placement puzzle where the objective is to fill a square grid of size 'n' with numbers between 1 to 'n'. The numbers must be placed so that each column, each row, and each of the sub-grids (if any) contains all of the numbers from 1 to 'n'.

The most common Sudoku puzzles use a 9x9 grid. The grids are partially filled (with hints) to ensure a solution can be reached.

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

And here's the solution. Notice how each row, each column and each sub-grid have all numbers from 1 to 9. Some puzzles may even have multiple solutions.

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

ASIDE: SOLVING A SUDOKU PUZZLE

Sudoku is a logic-based puzzle. Needless to say, solving one requires a series of logical moves and might require a bit of guesswork. Since this isn't an article to explore how to solve a Sudoku puzzle, I'll just leave a link to one that helped me getting started: kristanix.com/sudokuepic/sudoku-solving-techniques.

BACKTRACKING:-

Backtracking is an algorithm for finding all (or some) of the solutions to a problem that incrementally builds candidates to the solution(s). As soon as it determines that a candidate cannot possibly lead to a valid solution, it abandons the candidate. Backtracking is all about choices and consequences.

Abandoning a candidate typically results in visiting a previous stage of the problem-solving-process. This is what it means to “backtrack”—visit a previous stage and explore new possibilities from thereon.

Usually, apart from the original problem and the end goal, we also have a set of constraints that the solution must satisfy.

The simplest (read ‘dumbest’) implementations often use little to no “logic” or “insight” to the problem. Instead, they frantically try to find a solution by guesswork.

A backtracking algorithm can be thought of as a *tree of possibilities*. In this tree, the root node is the original problem, each node is a candidate and the leaf nodes are possible solution candidates. We traverse the tree depth-first from root to a leaf node to solve the problem.

SUDOKU & BACKTRACKING

PROBLEM

Given a, possibly, partially filled grid of size 'n', completely fill the grid with number between 1 and 'n'.

GOAL

Goal is defined for verifying the solution. Once the goal is reached, searching terminates. A fully filled grid is a solution if:

1. Each row has all numbers form 1 to 'n'.
2. Each column has all numbers form 1 to 'n'.
3. Each sub-grid (if any) has all numbers form 1 to 'n'.

CONSTRAINTS

Constraints are defined for verifying each candidate. A candidate is valid if:

1. Each row has unique numbers form 1 to 'n' or empty spaces.
2. Each column has unique numbers form 1 to 'n' or empty spaces.
3. Each sub-grid (if any) has unique numbers form 1 to 'n' or empty spaces.

TERMINATION CONDITIONS

Typically, backtracking algorithms have termination conditions other than reaching goal. These help with failures in solving the problem and special cases of the problem itself.

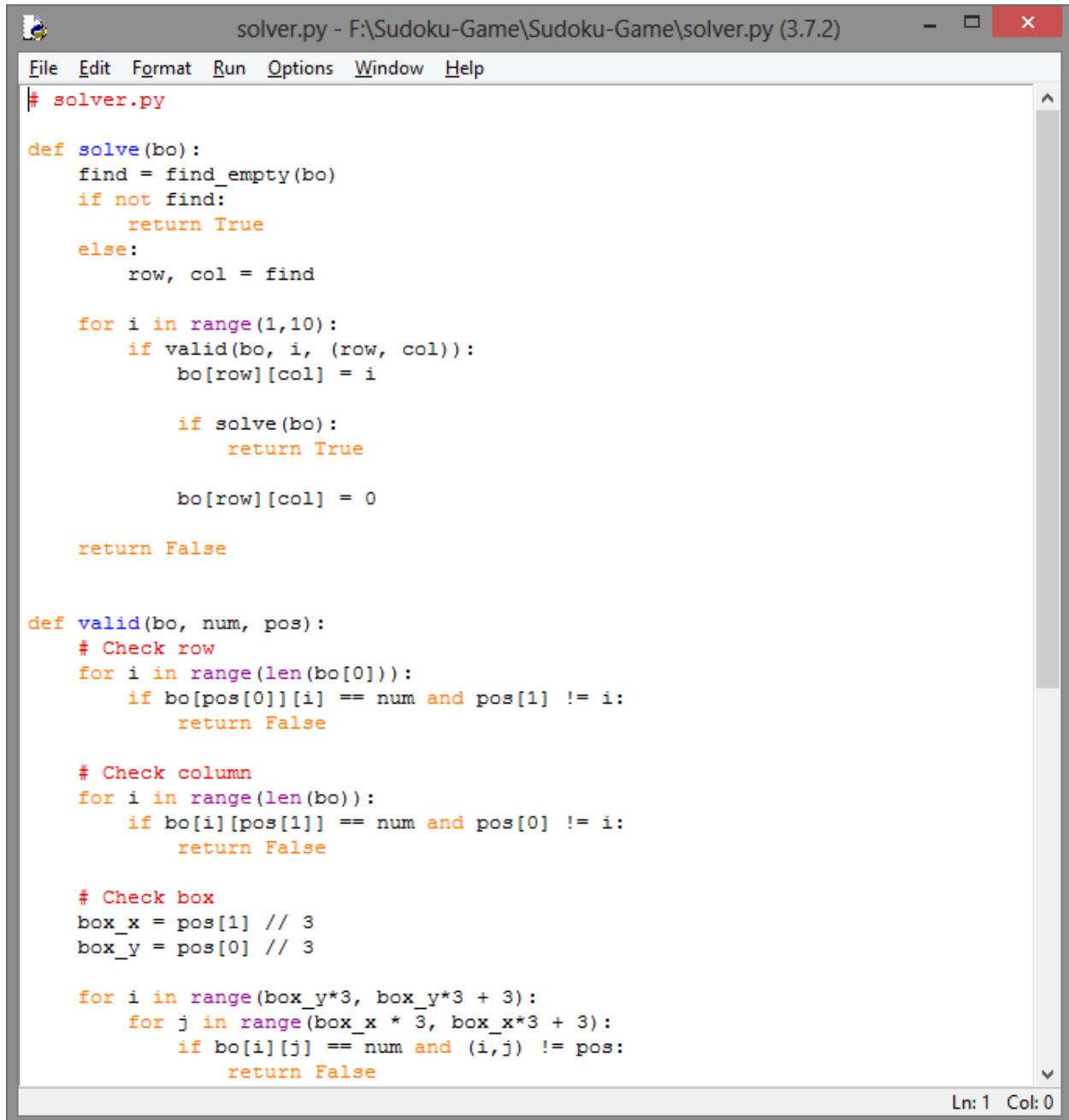
1. There are no empty spots left to fill and the candidate still doesn't qualify as a the solution.
2. There are no empty spots to begin with, i.e., the grid is already fully filled.

STEP-BY-STEP ALGORITHM

Here's how our code will "guess" at each step, all the way to the final solution:

1. Make a list of all the empty spots.
2. Select a spot and place a number, between 1 and 'n', in it and validate the candidate grid.
3. If any of the constraints fails, *abandon candidate and repeat step 2 with the next number*. Otherwise, check if the goal is reached.
4. If a solution is found, stop searching. Otherwise, repeat steps 2 to 4.

PROGRAM



```
# solver.py

def solve(bo):
    find = find_empty(bo)
    if not find:
        return True
    else:
        row, col = find

        for i in range(1,10):
            if valid(bo, i, (row, col)):
                bo[row][col] = i

                if solve(bo):
                    return True

                bo[row][col] = 0

        return False

def valid(bo, num, pos):
    # Check row
    for i in range(len(bo[0])):
        if bo[pos[0]][i] == num and pos[1] != i:
            return False

    # Check column
    for i in range(len(bo)):
        if bo[i][pos[1]] == num and pos[0] != i:
            return False

    # Check box
    box_x = pos[1] // 3
    box_y = pos[0] // 3

    for i in range(box_y*3, box_y*3 + 3):
        for j in range(box_x * 3, box_x*3 + 3):
            if bo[i][j] == num and (i,j) != pos:
                return False
```

Ln: 1 Col: 0


```
solver.py - F:\Sudoku-Game\Sudoku-Game\solver.py (3.7.2)
File Edit Format Run Options Window Help
# solver.py

def solve(bo):
    find = find_empty(bo)
    if not find:
        return True
    else:
        row, col = find

        for i in range(1,10):
            if valid(bo, i, (row, col)):
                bo[row][col] = i

                if solve(bo):
                    return True

                bo[row][col] = 0

        return False

def valid(bo, num, pos):
    # Check row
    for i in range(len(bo[0])):
        if bo[pos[0]][i] == num and pos[1] != i:
            return False


    # Check column
    for i in range(len(bo)):
        if bo[i][pos[1]] == num and pos[0] != i:
            return False

    # Check box
    box_x = pos[1] // 3
    box_y = pos[0] // 3

    for i in range(box_y*3, box_y*3 + 3):
        for j in range(box_x * 3, box_x*3 + 3):
            if bo[i][j] == num and (i,j) != pos:
                return False

Ln: 1 Col: 0
```

OUTPUT

 Sudoku - □ ×

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

Time: 0:13

SAMPLE PROBLEMS

We can try to solve some other problems by basing our approach on our current understanding.

The **Binary Watch** problem is a close enough relative to the Sudoku solver. Two more problems that rank similar are **Combination Sum III** and **Permutation Sequence**.

There are a lot more problems that you can try on.

However, trying to mold them into the Sudoku solver pattern might not always be trivial. It's best to try to formulate your approach in the following pattern:

1. Problem
2. Goal
3. Constraints
4. Termination conditions
5. Step-by-step algorithm