Algorithms and Data Structures

BACKTRACKING ALGORITHM
programming project

Sudoku creation
Sudoku solvation

Students:
Marta Ortigas
Núria Mitjavila

Teacher Supervisor:
José Luis Balcázar

Algorithms and Data Structures
Second term, Second year, 2022 - 2023
Bioinformatics degree, ESCI - UPF

1. Introduction

For the programming project of the Algorithms and Data Structures, we decide to do a Backtracking Algorithm to create and solve a Sudoku with a size of 9x9. (more introduction)

The Backtracking Algorithm is an algorithmic technique for solving problems recursively by trying to build a solution incrementally, one piece at a time, removing those solutions that fail to satisfy the constraints of the problem at any point in time (by time, here, is referred to the time elapsed till reaching any level of the search tree) - GeeksforGeeks. The backtracking is related to Depth-First Search, except that the graph remains implicit, being a vertex of a graph representing a subproblem and the edges representing the decisions to go from one vertex to another. Recursion is one of the mains concepts to take into account in backtracking, because the backtracking algorithms are mostly solved by calling the function itself, directly or indirectly, looking for all possible solutions to select the best one according to some criterion.

The Sudoku, from the Japanese expression: "Sūji wa dokushin ni kagiru" (numbers can only appear once), is a mathematical game that, despite having its origins around the 18th century, did not gain popularity until the 1980s in Japan and became famous internationally in 2005. The objective of aKalyazinSudoku is to fill a grid of 9 $\times$ 9 cells (81 cells) divided into 3 $\times$ 3 subgrids (also called "boxes" or "regions") with the numbers from 1 to 9 starting from some numbers already arranged in some of the cells. The rules of the game are that the numbers cannot be repeated in the same row, column or sub-grid. A Sudoku only have one solution.

During the programming process, we faced some problems. Our main objective was to create a code that solves a 9x9 Sudoku, but once we finish it, we decide to amplify our code, and it was at this point when this problems came out. First, we look at the possibility of solving problems of different sizes, like a 6x6 or a 12x12 Sudoku. At this point, we find the problem that it was so difficult to create a class that was able to solve games of different sizes, because for example the 6x6 and the 12x12 are not squared Sudoku's and when we look at the 16x16 with our code it takes so much time if we solve it by using backtracking. Another problem that we had to deal during the code is explained later, in the SudokuCreator Class section.

Code Explanation

SudokuSolver Class

Our code, is separated by three main parts, solves a Sudoku, creates a Sudoku and the main. The first created class is named SudokuSolver with the goal to solve a given 9x9 Sudoku. The first function we found in the class is the printInput, that takes the given Sudoku with numbers and blank spaces (given as points) and transforms it in a more understandable way, so that we can observe the given problem in the terminal. As we know that the Sudoku size is 9x9, we print the input with little squares of 3x3 and in the blank spaces, instead of having points, we change them into little squares: "□". This function, will be executed in the solution of the Sudoku, so that the input that you give can be seen clear but also when we will be creating a Sudoku, so that the problem that we create can also be seen on a better way. This is the code:

```
def printInput(self, table):
    print()
    print("This is the given Sudoku:")
    print("-------------------------")
    count = 0
    for x in table:
        if count == 3:
            print("--------+-------+--------")
            count = 0
        count += 1
        for y in range(0, 9):
            if y == 0 or y == 3 or y == 6:
                print("|", end=" ")
            if x[y] == '.':
                print("[]", end=" ")
            else:
                print(x[y], end=" ")
        print("|", end=" ")
        print()
    print("-------------------------")
```

The next functions are the important ones, that will check if there is or not a number and if not, if it is possible to put a number by looking at the row, the column and in the 3x3 squares. The point function checks if, for every place in the matrix, we have a number or do not. If we have a point (no number), we return the coordinates and if we already have a number, we return -1, to know that there is no need to search for a number to put there. The row and column functions are really similar, we just check with a for loop if a given value between one and nine is or not in a row or a column, returning True if it is valid to go there or False if it is not. For the squareCheck function, we "create" the square by selecting the row and the column that are in the middle of the square to then check if the given value is located between this row, a row less and a row more and this column, a column less and a column more. In that way, we can check if the value is in the square (return False) or is not in the square (return True). Here is the code:

```
def rowCheck(self, table, row, value):
    for x in range(0, 9):
        if value == table[row][x]:
            return False
    return True

def columnCheck(self, table, column, value):
    for x in range(0, 9):
        if value == table[x][column]:
            return False
    return True

def squareCheck(self, table, row, column, value):
    row = (row // 3) * 3 + 1
    column = (column // 3) * 3 + 1
    for x in [-1, 0, 1]:
        for y in [-1, 0, 1]:
            if table[row + x][column + y] == value:
                return False
    return True

def point(self, table):
    for x in range(0, 9):
        for y in range(0, 9):
            if table[x][y] == ".":
                return x, y
    return -1, -1
```

The solution function, is the key one, the one that solves the Sudoku using the previous functions and backtracks in case that we have an error. Remember that we are solving this Sudoku by starting at the first row, first column and checking for the lowest number if it can go there by looking at the row, the column and the square. If it fits there, we put it, but maybe there is another number that could also fit there but that we are not taking into account. If later we cannot put any number to a certain position after checking for all numbers in row, column and square, then we will backtrack to the previous numbers that we have been putting and look if there was another possible solution that is valid for the other empty spaces. By doing it, we will find the proper solution of the Sudoku. I found a good animation on YouTube that shows the Backtracking Algorithm in Sudoku, and in there is clearly seen how this code works.

Our solution function, takes a point in the matrix with two coordinates (x, y) and checks with the point function, if there is a number or not on there. If there is a point, we just return True and keep looking for another position. If there is not a number, we loop for all numbers between one and nine, if they can go at that position by looking at row, column and square functions. If we return True for all the functions, then we put the number in that position and run again the function to check for new numbers in new positions. If at a certain point we do not have any number that fits a row, column and square, then we put again a point, and we backtrack to find another valid number. Here is the code for the solution function:

```
def solution(self, table):
    x, y = self.point(table)
    if x == -1 and y == -1:
        return True
    for number in ['1', '2', '3', '4', '5', '6', '7', '8', '9']:
        if self.rowCheck(table, x, number) and self.columnCheck(table, y,
   number) and self.squareCheck(table, x, y, number):
            table[x][y] = number
            if self.solution(table):
                return True
            table[x][y] = "."
    return False
```

Finally, as we did with the printInput function, we did another one that prints the solution of the Sudoku once we have it. The only thing that changes in the code is that instead of substituting the points with squares, as we did before, we just put the number that we find in the previous function. Here is the code, and then you can observe how this function of the input and output format would be printed in the terminal in a more understandable way.

```
def printTable(self, table):
    print()
    print("This is the solution:")
    print("-------------------------")
    count = 0
    for x in table:
        if count == 3:
            print("--------+-------+--------")
            count = 0
        count += 1
        for y in range(0, 9):
            if y == 0 or y == 3 or y == 6:
                print("|", end=" ")
            print(x[y], end=" ")
        print("|", end=" ")
        print()
    print("-------------------------")
    return ''
```

```
-------------------------
| 4 8 3 | □ 5 □ | □ 2 □ |
| 1 □ □ | 4 8 3 | 5 □ 6 |
| □ 5 □ | □ 7 2 | □ 3 □ |
--------+-------+--------
| 9 6 7 | □ 1 □ | 4 □ □ |
| □ □ □ | □ 6 □ | 1 8 □ |
| □ □ 8 | 3 □ 5 | 6 7 □ |
--------+-------+--------
| 8 □ □ | 5 □ □ | 9 □ □ |
| □ 9 1 | 8 2 □ | 3 6 5 |
| □ 3 □ | □ □ 1 | □ □ 8 |
-------------------------
```

This is an example of how would the input of a Sudoku look like. In this case, the input is:

```
483.5..2.
1..4835.6
.5..72.3.
967.1.4..
....6.18.
..83.567.
8..5..9..
.9182.365
.3...1..8
```

SudokuCreator Class

The second created class is named SudokuCreator with the goal to create a 9x9 Sudoku. This class has fewer functions than the previous one, that was the main objective of our project. This class was created to amplify our work and generate Sudoku's that can later be solved by the other class. The first two functions are the generate and the random. In first one, we generate a base pattern for the Sudoku board. This function takes two arguments, two integers representing the row and column numbers, and we return a number between 0 and 8 based on the row and column values. The second one, shuffles the list "llista" and returns it. We use the random.sample() function to generate a new list with the same elements that are in "llista", but in a random order. This random.sample() function is from the random library.

In these points we meet one of the problems we could not deal with when we were doing the code. With this generate and random functions, we are generating a Sudoku that can be solved, but we cannot be sure that this Sudoku is meeting all the Sudoku requirements like that the created problem is having a unique solution. At the end of the page, there is a generated Sudoku with this class, where you can observe that multiple solutions are possible.

```
def generate(self, row, column):
    return (3 * (row % 3) + row // 3 + column) % 9

def random(self, llista):
    return random.sample(llista, len(llista))
```

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

Figure 1: Created Sudoku

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

Figure 2: First solution

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

Figure 3: Second solution

Finally, the main function of this class is the create_board, the function that will create the Sudoku using the previous functions. This function takes a single argument, difficulty, that is representing the different levels that the created Sudoku can have, easy, medium or hard. First, we generate a random order for the rows and the columns of the Sudoku board, and then we assign a random order for the Sudoku numbers, between one and nine, that will be placed in the different spaces. Then, we fill the spaces of the board with the numbers generated by the generate function and, depending on the difficulty level, we remove more or less numbers and substitute them with a point, indicating an empty space. Finally, we return the created Sudoku with some numbers and some empty spaces, so that you can solve the problem.

```
def create_board(self, difficulty):
    row = []
    for x in self.random(range(3)):
        for y in self.random(range(3)):
            row.append(x * 3 + y)
    column = []
    for x in self.random(range(3)):
        for y in self.random(range(3)):
            column.append(x * 3 + y)
    number = self.random(range(1, 10))
    table = []
    for r in row:
        row_list = []
        for c in column:
            row_list.append(number[self.generate(r, c)])
        table.append(row_list)
    squares = 9 * 9
    if difficulty == 'easy':
        empty = squares * 1 // 4
    elif difficulty == 'medium':
        empty = squares * 2 // 4
    else:
        empty = squares * 3 // 4
    for x in random.sample(range(squares), empty):
        table[x // 9][x % 9] = '.'
    return table
```

Main Function

The last function that we have in our program is, obviously, the main. Its principal role is to display on the terminal some questions to know if we are interested in solving or creating a Sudoku, if we are interested in solving it, take the input and return the Sudoku solved, and if we are interested in creating one, ask for the level of difficulty and return a Sudoku to be solved.

It is, for this reason, that we start the code by asking the executor if is interested in creating (1) or solving (2) the Sudoku. Once we know what we are interested in, we have a simple conditional statement. If we want to create, to solve or if the input was wrong. In the last case, it returns a message "Please, choose a valid option" and then it executes again the main function, so that the executor can enter again a valid option to solve or create the Sudoku.

If the choice is to create a Sudoku, then we ask for the level of difficulty (easy, medium, hard), we store this level, and we store the SudokuCreator() class in a variable. If the level is easy, then in goes to the create_board function of the SudokuCreator() class, returns a Sudoku with 20 empty spaces and 60 numbers and then goes to the printInput function that we create in the SudokuSolver() class to print the Sudoku in an understandable way. It does the same for the medium and hard difficulty, returning 40 empty spaces and 40 numbers for the medium level and 60 empty spaces and 20 numbers for the hard level. Finally, there is a condition where, if the level name was not a valid input, it returns "Please, choose a valid level of difficulty".

If the choice is to solve a Sudoku, then we ask to input the game, using a dot for the blank spaces, as we can observe in the example of one of the previous pages (at the bottom of page 6). With this input, we create a list, table, where we store all the given values and points (empty values) as rows (with lists of lists). Then, we print the given Sudoku in a better way, and we check if we have a solution for the problem by checking the solution function in the SudokuSolver() class. If there is a solution, we print this solution in the terminal with the printTable function and if there is no solution, we return "there is no solution for this Sudoku!". It is at this point the place where our code ends after creating two classes, one for creating a Sudoku and one for solving it using backtracking and a main function that executes them.

```
def main():
    print("Choose if you want the program to create a sudoku or you want to
   solve one")
    print("Write 1 to create a sudoku")
    print("Write 2 to solve a sudoku")
    option = input()
    sol = SudokuSolver()

    if option == '1':
        sudoku = SudokuCreator()
        print("Please, input a level of difficulty (easy, medium, hard):")
        level = input()
        if level == 'easy':
            board = sudoku.create_board('easy')
            sol.printInput(board)
        elif level == 'medium':
            board = sudoku.create_board('medium')
            sol.printInput(board)
        elif level == 'hard':
            board = sudoku.create_board('hard')
            sol.printInput(board)
        else:
            print("Please, choose a valid level of difficulty")

    elif option == '2':
        table = []
        print("Please, input the sudoku (use a dot for the blank spaces):")
        for x in range(0, 9):
            table.append(list(input()))
        sol.printInput(table)

        if sol.solution(table):
            print(sol.printTable(table))
        else:
            print()
            print("There is no solution for this Sudoku!")

    else:
        print("Please, choose a valid option")
        main()

if __name__ == '__main__':
    main()
```