Killer Sudoku

Sudoku is one of the most popular puzzle games of all time. The goal of Sudoku is to fill a 9×9 grid with numbers so that each row, column and 3×3 section contain all of the digits between 1 and 9.

We take this game one step further and introduce a slightly more difficult version termed as “Killer Sudoku” (https://en.wikipedia.org/wiki/Killer_sudoku). “Killer Sudoku” is a regular Sudoku with additional constraints on some particular cells summing up to specific numbers.

Here are some definitions as pertaining to the Killer Sudoku puzzle which we will refer to from now onwards:
Cell: A single square that contains one number in the grid
Row: A horizontal line of 9 cells
Column: A vertical line of 9 cells
Nonet: A 3 by 3 grid of cells (outlined by the bold lines in the diagram above) also called a box.
Cage: The grouping of cells denoted by individual colours.
House: Any non-repeating set of 9 cells (a general term for "row, column, or nonet")

The objective is to fill the grid with numbers from 1 to 9 in a way that the following conditions are met:

- Each row, column, and nonet contains each number from 1-9 exactly once.

- The sum of all numbers in a cage must be equal to the small number printed in its corner.
- No number appears more than once in a cage.

To explain this more clearly, consider the image of the empty killer Sudoku puzzle provided above, solving this entails that you place the digits 1 to 9 in all the blocks such that each row, column and 3 by 3 section contain all of the digits between 1 and 9 and additionally the sum of all the digits in every coloured region should be equal to the small number provided in that region. Also each coloured region should contain all different digits.

For example the first 2 digits that need to be filled in the first row (corresponding to the region in yellow colour) should sum upto 3. The full solution of the above killer Sudoku is as follows:

This project aims to program an automatic Killer Sudoku playing agent which when given an empty Killer Sudoku grid, is able to achieve the assignment of numbers from 1 to 9 as per the rules of the Killer Sudoku.

Solution Strategy - Backtracking

What is a backtracking algorithm ? (https://en.wikipedia.org/wiki/Backtracking)
In backtracking algorithms you try to build a solution one step at a time. If at some step it becomes clear that the current path that you are on cannot lead to a solution you go back to the previous step (backtrack) and choose a different path. Briefly, once you exhaust all your

options at a certain step you go back. Think of a labyrinth or maze - how do you find a way from an entrance to an exit? Once you reach a dead end, you must backtrack. But backtrack to where? - to the previous choice point. Backtracking is also known as depth-first search.

Approach for solving sudoku using recursive backtracking algorithm
Like all other Backtracking problems, we can solve Sudoku by one by one assigning numbers to empty cells. Before assigning a number, we need to confirm that the same number is not present in current row, current column and current 3 by 3 subgrid. If number is not present in respective row, column or subgrid, we can assign the number, and recursively check if this assignment leads to a solution or not. If the assignment does not lead to a solution, then we try next number for the current empty cell. And if none of numbers (1 to 9) lead to solution, we will return false.

How to extend this logic to Killer Sudoku?
The solution to a Killer Sudoku uses the backtracking algorithm as described above but with additional constraint checks whenever a number is assigned to a cell. If all of the following constraints are satisfied, we can assign a number to a cell:

- The same number is not present in current row, current column and current nonet.
- Assigning this number to the cell does not violate the sum of that cage.
- This number is not already present in its cage at some other cell.

If any one of these constraints is violated, then proceed with assigning a different number to that cell. If all different assignments 1 to 9 violate the constraints, then backtrack one step and try out a different assignment to the cell that was assigned previously.

Implementation and Input/Output Details

Input Format
The input will consist of an unsolved killer sudoku in the following format:
A cell will be represented in standard matrix index notation. E.g. (0, 0) will denote the cell in the first row and first column. (4, 5) will denote the cell in the 5th row and 6th column.
First line will consist of the number of cages in the entire sudoku (say C).
This is followed by C sets of lines as follows:

a. First line consists 2 space separated integers denoting the number of cells in the cage (say c_n) and the required sum of cells in that cage (say S_n).
b. Next c_n lines will contain 2 space separated integers x and y denoting the index of the cells that are part of the current cage.

Output Format
The output must be a completely solved Killer Sudoku - 9 rows containing 9 space separated integers (between 1-9) each.

Example
Input:
The above killer sudoku will be represented in the input as follows (Comments only for clarity):

```
29 // Total number of cages is 29
2 3 // The first cage will contain 2 numbers which must sum to 3.
0 0 // (0, 0) is a cell in the first cage.
0 1 // (0, 1) is a cell in the first cage.
3 15 // Next cage
0 2
0 3
0 4
4 22 // Next cage
0 5
1 4
1 5
2 4
2 4 // Next cage
0 6
1 6
2 16 // Next cage
0 7
1 7
4 15 // Next cage
0 8
1 8
2 8
3 8
4 25 // Next cage
1 0
1 1
2 0
2 1
2 17 // Next cage
1 2
1 3
3 9 // Next cage
2 2
2 3
3 3
3 8 // Next cage
2 5
3 5
4 5
3 20 // Next cage
2 6
2 7
3 6
2 6 // Next cage
3 0
```

```
4 0
2 14 // Next cage
3 1
3 2
3 17 // Next cage
3 4
4 4
5 4
3 17 // Next cage
3 7
4 7
4 6
3 13 // Next cage
4 1
4 2
5 1
3 20 // Next cage
4 3
5 3
6 3
2 12 // Next cage
4 8
5 8
4 27 // Next cage
5 0
6 0
7 0
8 0
3 6 // Next cage
5 2
6 1
6 2
3 20 // Next cage
5 5
6 5
6 6
2 6 // Next cage
5 6
5 7
4 10 // Next cage
6 4
7 3
7 4
8 3
4 14 // Next cage
6 7
```

```
6 8
7 7
7 8
2 8 // Next cage
7 1
8 1
2 16 // Next cage
7 2
8 2
2 15 // Next cage
7 5
7 6
3 13 // Next cage
8 4
8 5
8 6
2 17 // Next cage
8 7
8 8
```

Output:

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