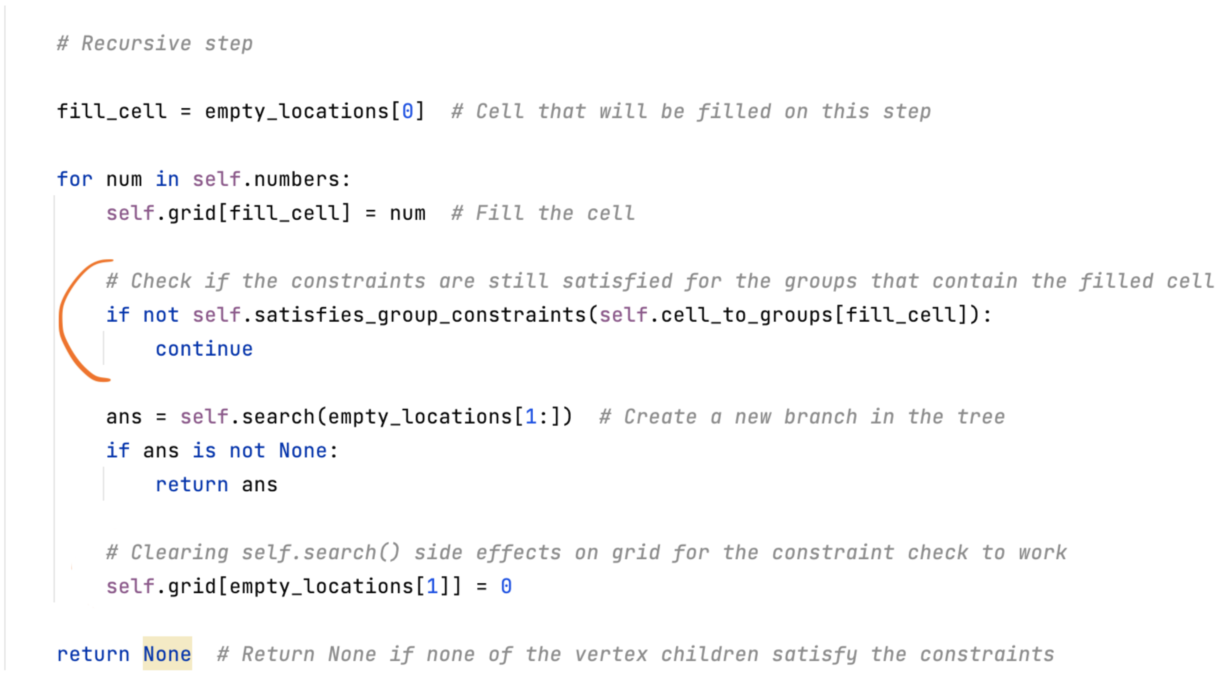
Algorithms and Data Structures - Assignment 1

Introduction

The problem itself reminds a Sudoku game: you have to fill out the grid with the given constraints using the numbers from the given set. But here the constraints are not the rules of the game, but the input of the user. These restrictions are applied to the arbitrary group of cells and fix the maximum sum of all elements and the number of repetitions.
An exhaustive search approach (with backtracking) was chosen to solve this challenge. Such an algorithm has to iterate through all possible combinations of numbers in the grid to find a possible solution. To structure our search we will use state spaces. In our case, the **state** is represented by the grid, which is uniquely filled with numbers. So, the starting point is the grid, given at the very beginning. It may be a 0 matrix or some numbers may be already there. An **action** is a substitution of one zero with a number from the set of allowed numbers. So by action, we traverse to another uniquely filled matrix.
Also, we use backtracking. It means, that when we see that the constraints are already violated, we rewrite the last number written with another number. So, we are going to another branch of possible states without trying the current one to the very end. That's because there is no reason to continue on this branch, as we already break the restrictions.

The difference between exhaustive search and backtracking

The difference between these two options is very simple. When we are using exhaustive search, we try every possible state (or combination, if you prefer). So we are filling the grid up to the very end, and only afterward check the constraints. On the other hand, the backtracking method checks restrictions on every action. Why is that? - To save resources! Imagine you have put your first number and this option already doesn't fit with limitations. As human, you probably understand, that there is no reason to continue putting numbers, by no means you won't get to the right answer. But a computer, which uses exhaustive search will continue trying. Such attempts will cost us time. So let's tell the computer to check the restrictions every time it makes a move (action) so we don't fall into useless work. And that's called backtracking.



```python
# Recursive step

fill_cell = empty_locations[0]  # Cell that will be filled on this step

for num in self.numbers:
    self.grid[fill_cell] = num  # Fill the cell

    # Check if the constraints are still satisfied for the groups that contain the filled cell
    if not self.satisfies_group_constraints(self.cell_to_groups[fill_cell]):
        continue

    ans = self.search(empty_locations[1:])  # Create a new branch in the tree
    if ans is not None:
        return ans

    # Clearing self.search() side effects on grid for the constraint check to work
    self.grid[empty_locations[1]] = 0

return None  # Return None if none of the vertex children satisfy the constraints
```

Figure 1: Implementation of backtracking

In our case backtracking is implemented. You can see it in Figure 1. The block of code highlighted is responsible for picking another number in the column if this one breaks the restrictions.

Drawing trees

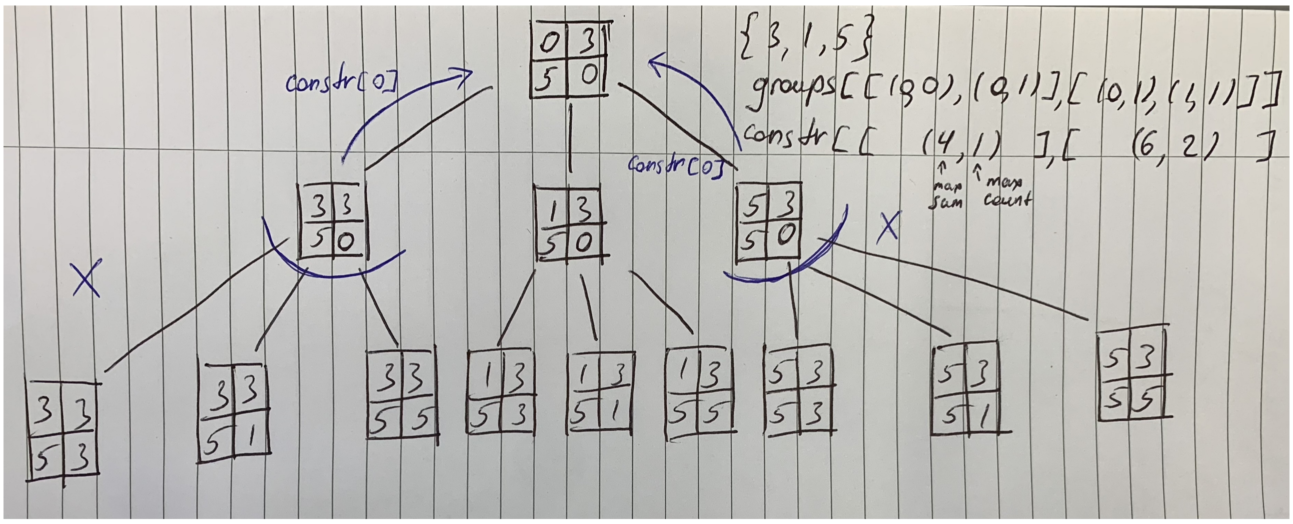

{3, 1, 5}
groups[[(0,0),(0,1)],[(0,1),(1,1)]]
constr[[(4,1)],[(6,2)]
max sum
max count
constr[0]
constr[0]

Figure 2: State tree

In Figure 2 you may see a state tree for a small grid. The groups and limitations are written at the top right corner. The black pen represents the traversal of the exhaustive search (it just visits every node in the tree), meanwhile, the blue one denotes the backtracking route. It can be observed, that backtracking doesn't visit 2 out of 3 subtrees of the root, only the roots of these subtrees. As a result, we almost halved the number of nodes! At the same time, backtracking is still optimal, as we surely will find a solution if it exists. That's because we cut off only the solutions, which is certainly wrong. As you can assure from the example in Figure 2, we cut only inappropriate cases.

Greedy approach

A greedy approach to this problem would be implemented following way:

1. You take groups one by one as in the list
2. In every group you take cells one by one as in the list
3. In every cell you put the minimum possible value, which doesn't violate constraints

Now let's consider an example presented in Figure 3. The algorithm is going to the following actions:

1. Go to group 0
2. Put 1 in the cell (0, 0) as it's the minimal possible value
3. Put 3 in the cell (0, 1) as it's the minimal possible value
4. Go to group 1
5. 1st cell already filled
6. Put 3 in the cell (1, 1) as it's the minimal possible value

And that's where the problem arises. It would be better to put 3 into (0, 0) and 1 into (0, 1) and (1, 1), but the greedy approach is not able to find such a solution. Hereby, it is not optimal.

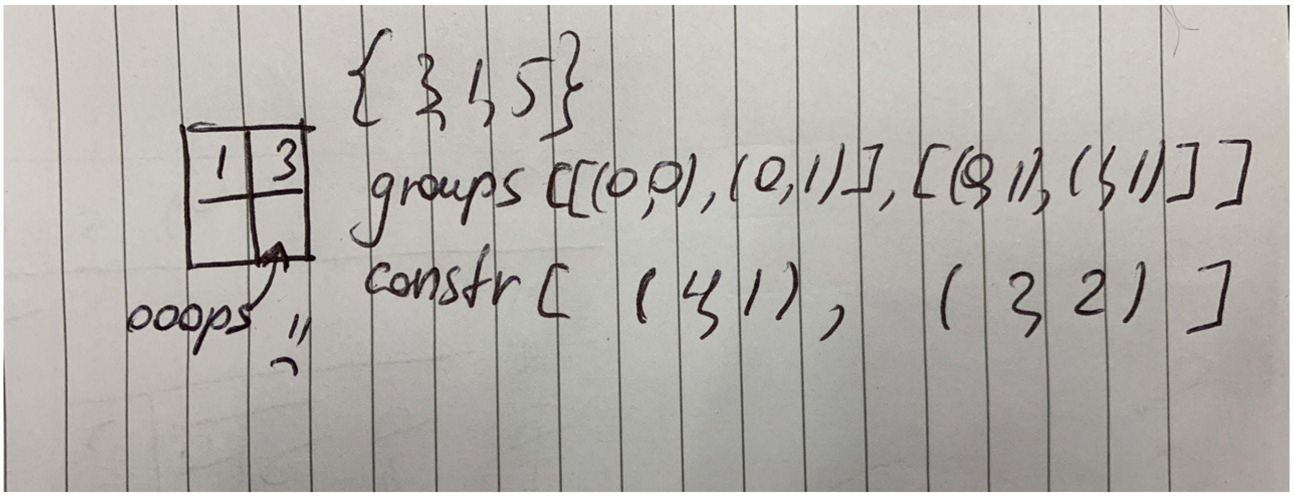


Figure 3: Counterexample for greedy approach

Summary and Discussion

Exhaustive search is a great algorithm when something more advanced is either not needed or not possible. In our example of a Sudoku-like game, it fits well. However, its upper-bound complexity is usually above $\mathcal{O}(n)$. So, it should be avoided where it's possible. The better variant of exhaustive search is exhaustive search with backtracking. We are adding the feature of checking constraints on every step, not only on the last one, as it is in the classical exhaustive search. Such a small check can save us a huge amount of resources. For instance, in this report, we have already seen a small example (Figure 2) where backtracking cut off almost half of the space tree. Finally, we tried to use a greedy approach for the task given. But proved, that it will not be optimal.

Contributions

Student Maksym Lytovka (s3705609): Code (all functions), Report
Student Ivan Banny (s3647951): Code (all functions), Report