Algorithms and Data Structures - Assignment 1

Introduction

The task is to fill a 2D grid of the form *n*x*n* with non-zero values from a set of k different permissible numbers while satisfying the constraints. The group members are a set of locations on the 2D grid. Two constraints need to be taken into consideration. The maximum sum constraint determines the maximum value of all group members added together and the maximum count constraint determines the maximum number multiplicity of a value within a group. The group members are a set of locations on the 2D grid. State: A partially filled 2D grid, where some cells have been assigned a non-zero value from the permissible set of numbers, and some cells are still empty (filled with zeros). Assigning a non-zero value from the permissible set of numbers to an empty cell in the 2D grid. In order to have a valid grid all constraints have to be fulfilled.

Difference between exhaustive search (ES) and backtracking (BT)

For the explanation we refer to Figure 1 that contains all possible combinations of filling the given numbers into the grid. Using the exhaustive search approach you would recursively search the tree until you find a leaf of the tree. Only when you found the leaf, you check if the completely filled grid violates any constraints.
In the backtracking approach on the other hand, you would check the constraints at every node on the way towards the leaf. If the constraints are already violated at a node, the algorithm goes back to the node where the constraints were not violated and continues from there. Thus one saves much processing time by skipping filling the nodes that are anyway not considered.
The implemented code in the function "search(...)" is a exhaustive search algorithm, since the only time when the constraints are checked, is when "len(empty_locations)==0". That corresponds to checking all leafs for constraints. Within the for loop the constraints are not checked.
The "search_backtracking(...)" function on the other hand checks the constraints only within the for-loop directly after the tryout-number is assigned to the grid. If the constraint is not satisfied the recursive call is not performed and none of the children of the violated node that do not fulfill the constraints are checked. In short this means that the timing of checking the group constraints determines which algorithm is used.
Furthermore, for the exhaustive search, all permutations of grids are computed, and only after that are the grids (states) checked for the group constraints.
For the backtracking, the constraints get checked after every empty cell is filled (action), and if the constraints are not fulfilled the follow-up actions are discarded and a follow-up action for a valid grid (state) is computed.
Thus, the ES algorithm can be converted to a BT algorithm by applying the constraint check directly after every action.

This should result in major time improvements of the backtracking algorithm compared to the exhaustive search algorithm. To check this we implemented a test measuring the runtime of the different algorithms using the " time.monotonic_ns() " and " time.perf_counter_ns() " methods. We compare the exhaustive serch, backtracking and greedy backtracking to each other. The improvements of the greedy backtracking algorithm are explained in Section Greedy approach to search for solution for given CSP. Table 1 displays the runtime differences of the different algorithms for three test runs and its average. We use a 4x4 grid, to properly measure the times of the backtracking and greedy backtracking algorithm that would be low otherways.

	Exhaustive Search	**Backtracking**	**Greedy BT**
Run 1 [ms]	45361.068	0.093	0.038
Run 2 [ms]	45498.132	0.055	0.030
Run 3 [ms]	45881.657	0.129	0.034
Average [ms]	45580.286	0.092	0.034

Table 1: Runtime comparison of the exhaustive search, backtracking and greedy backtracking algorithm.

The runtime analysis shows that the backtracking algorithms are in the order of 100000 times faster than the

exhaustive search algorithm for our test run, which was expected but which is still very high. The optimizations of the greedy approach make the algorithm around three times faster than solely the backtracking algorithm.
An interesting observation was that which algorithm was the best depended on how the time was measured. The exhaustive search is in any case the worst, but the backtracking algorithm was in some cases faster than the greedy backtracking. When we used the "timeit" module instead of the "time" module, which repeats the measurement multiple times and calculates the average time out of it, the backtracking was slightly faster than the greedy backtracking algorithm. Possible reasons for that is that the cost of the complement calculation as described below is too high to have a significant improvement. Since this is way over the requested task, we don't further investigate on this.

Search trees: ES vs BT

We use an example for the search tree with the following constraints:
k-permissible numbers: 1,2
sum constraint: 3
count constraint: 1
groups: $[(0,0),(0,1)], [(0,0),(1,0)], [(1,0),(1,1)], [(0,1),(1,1)]$

Search tree description

The parse tree shows the states and actions of the grid. Every square shows the state of the grid and each vertex embodies an action adding a number to an empty cell from the k permissible numbers. We find that only nodes 7 and 10 are valid grids.

1. Exhaustive search algorithm

In Figure 1 2, all the possible permutations of the grid (disregarding the constraints) are represented by the children from 1-16. Only after computing all the possible grids with the k-permissible numbers do we apply the constraints that are given above.

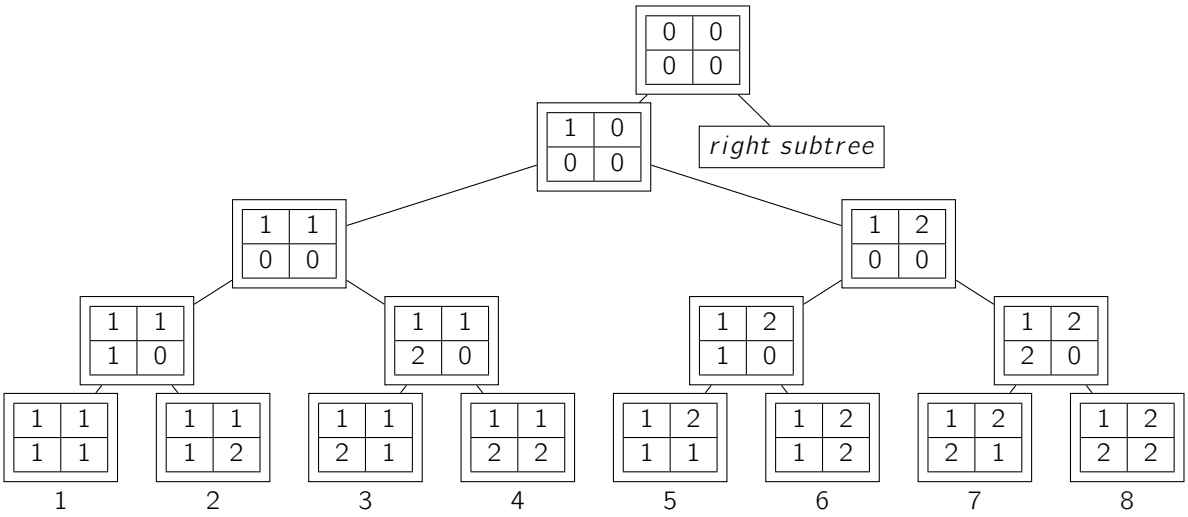

Figure 1: ES

2. Backtracking search algorithm

In Figure 3, after the computation of each new node, the constraints are checked instantly. If the constraints are fulfilled, the next two children are computed and if the constraints are violated, the further computation for the children of the node is discarded. This way 18 computations (about 55% of computations) are saved in this scenario. The ES uses 32 and the BT uses 14 computations.

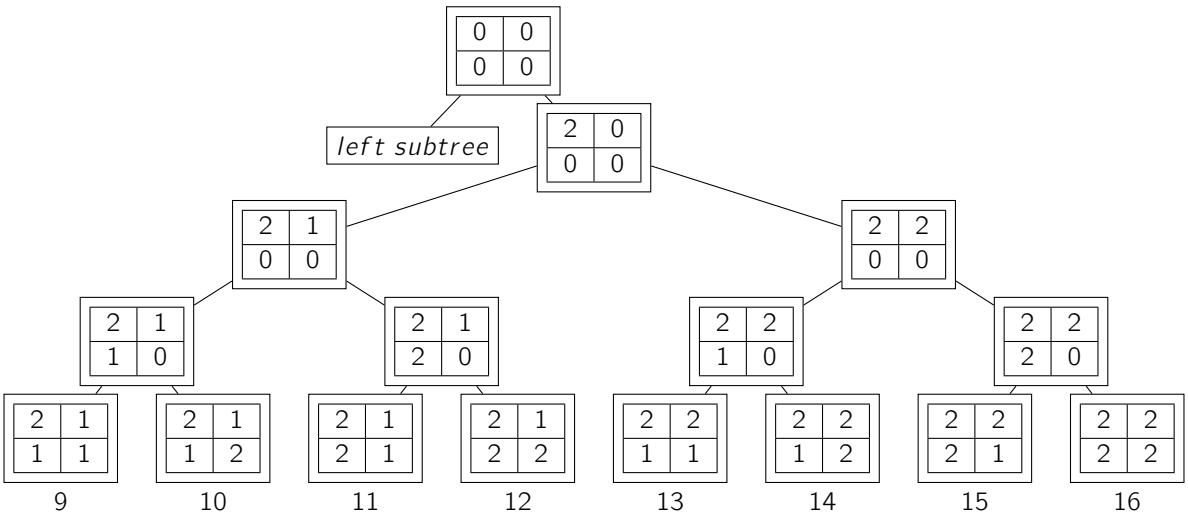

Figure 2: ES

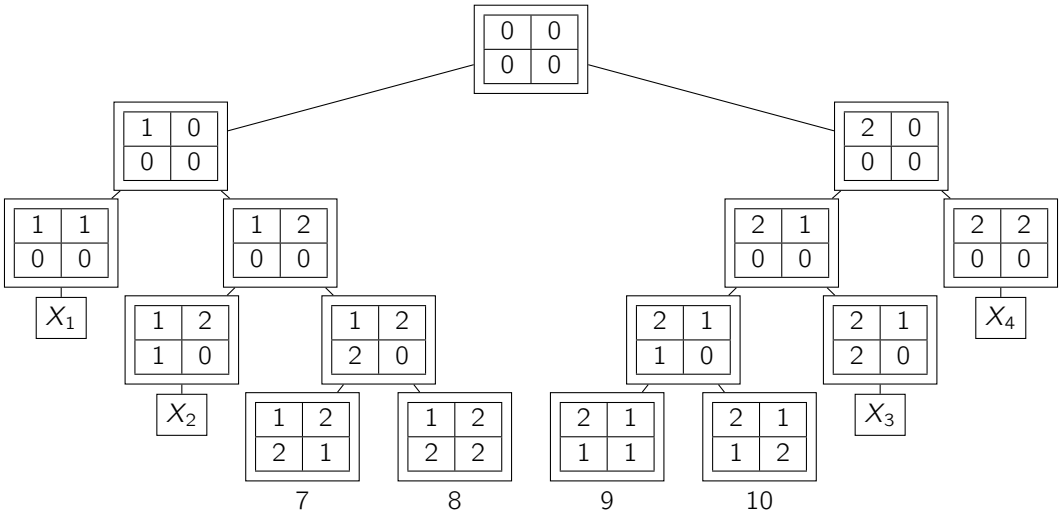

Figure 3: BT

Greedy approach to search for solution for given CSP

The greedy approach implemented in "search_greedy_backtracking(...)" uses the basic code of "search_backtracking(...)" and has two major improvements compared to the predecessor.
Firstly the number set is first sorted before being used. This has the advantage that the algorithm begins with the lowest number. This way, it is less likely that the algorithm runs into the sum constraint. Speaking in terms of the search tree, we use a binary search tree instead of a binary tree.
Secondly, we can only take the numbers that are lower than the difference between the sum constraint and the sum of the already-filled locations within the group into account. This is because all numbers that are bigger than that threshold would not pass the constraint check anyway. As explained in the binary search tree image, we neglect all nodes that are right and below from the highest possible element.
This should yield better results in the case that there way more numbers than actually needed numbers to solve the CSP. Then the sorting makes it way less likely to run into sum constraint and the filtering filters out all the impossible numbers.
In the case that there are exactly as many numbers as needed, the improvement to filter out too big elements causes additional execution cost, although there are no elements filtered out. In this case the greedy approach could be slower than the simple backtracking algorithm.

Summary and Discussion

Goal of the assignment

In the assignment, we aimed to investigate the ES-, and BT-algorihm and compared both under the criteria:

- methodology
- effectiveness
- efficiency (runtime)

Furthermore we thrived to create a greedy algorithm for an optimal approach

Results

We found that:

1. The BT algorithm way more efficient than the ES algorithm
    - Theoretically from the example search tree, it is obvious that the BT algorithm is faster for big data sets.
    - In our test run around the backtracking algorithm is 100000 times faster than the exhaustive search algorithm.
    - The theoretical observation match the experimental obersavation, although a time difference of 100000 is way more than expected
2. The greedy approach is around three times faster than the backtracking algorithm
    - From the runtime table 1 it can be seen that the greedy approach is the best solution.
    - It is optimal because there are no calculations used for impossible grids, which is secured by the difference of the sum and count constraint combined with the backtracking. Also, the ordering gives a higher probability for faster computations.
3. The results depend on the method of measuring the runtime
4. Best running time using the method described in Difference between exhaustive search (ES) and backtracking (BT): Greedy Backtracking, Backtracking, Exhaustive Search

All in all, all goals have been met and we found answers to all three investigation questions.

Contributions

- David Wünsch (s3665534):
    - Differences between exhaustive search (ES) and backtracking (BT)
    - Greedy approach to search for solution for given CSP
    - Discussion on Runtime
    - Main contribution of code (Helper methods, Search methods, Runtime, Tests)
- Jesse Kroll (s3666778):
    - Introduction
    - Search trees: ES vs BT
    - Summary and discussion
    - Contribution to code (Helper methods, Search methods, Tests)