Algorithms and Data Structures - Assignment 1

Daniël Zee, s2063131
Joris Nouwens, s1278002

March 25, 2023

Introduction

For this assignment we had to implement a recursive exhaustive search algorithm to solve a speficic *Constraint Satisfaction Problem* (CSP). In this problem we have a 2D grid of any size that is either filled with either zeros or with values from a set of k permissible numbers $N = \{n_1, n_2, ...n_k\}$. The goal is to replace all the zero cells with values from set N while satisfying certain *group constraints*. Groups are a set of cell coordinates specified for which the cell values together must follow the following rules:

- The sum of the values of the group members must not exceed a specified threshold.
- Any value in k must not occur in the group more than a specified amount of times.

Figure 1 show an example of a grid to use for this CSP. We will use this example to explain and analyse our algorithm and other approaches. This example uses the following groups $M(G_1) = \{(0,0),(0,1)\}$, $M(G_2) = \{(1,0),(1,1)\}$, $M(G_3) = \{(0,0),(1,0)\}$ and $M(G_4) = \{(0,1),(1,1)\}$, so every cell is in a group with its horizontal and vertical neighbor. All groups in this example follow the same set of constraints, the sum constraint being $S(G_i) = 3$ and the count constaint being $C(G_i) = 1$.

(0,0)	(0,1)
(1,0)	(1,1)

(a) The coordinate system

1	0
0	0

(b) Partially filled in grid

1	2
2	1

(c) Solution grid

Figure 1: An example of a 2D grid

Exhaustive search and backtracking approach

For this assignment we only had to implement an exhaustive search approach for our algorithm. Algorithm 1 shows our recursive implementation in pseudocode. The algorithm first replaces every zero cell in the grid with a value from N. After filling everything it checks if the group constraints still satisfy for the groups which the filled in celles are part of, starting from the last filled in cell. If this is not the case, the algorithm will stop checking the group constraints any further and change the zero cells to a different configuration and starts checking again. The algorithm stops if a solution grid has been found or if all configurations have been tried.

This approach can be classified as exhaustive search because we fill in all the zero cells in the grid before we start checking the group constraints. A more efficient approach would be to check the relevant group constraints after every zero cell we replace in the grid, because we are not interested in filling in the grid any further after one of the groups no longer satisfies the constraints. This would be classified as a backtracking approach because we immediately discard the candidate solution when we know that it will never be valid. So to change our current implementation to backtracking we would have to check for the group constraints before recursively calling our function again and return *None* if we already know the grid so far is invalid.

Algorithm 1 Recursive exhaustive search approach

Input: A list of the cells filled with zero, μ
Output: The current grid solution or *None*

if the length of μ greater than 0 **then**
    for every number in N **do**
        Replace the grid value of the first cell in μ with the current n_i
        Run the algorithm with the first cell in μ excluded and save the output in λ
        if λ is not *None* and the group constraints still hold after changing the value of the first cell in μ **then**
            return the current grid solution
        end if
    end for
    Change the grid value of the first cell in μ back to zero
    return *None*
else
    return the current grid solution
end if

Search trees

Figure 2 shows the search tree for our example in figure 1 using the exhaustive search approach. You can see in red that the algorithm only discards solutions after every zero cell has been filled in, even if we could have known earlier that the group constraints would not satisfy.

Figure 3 shows the search tree for the described backtracking approach. You can see that this approach spends significantly less time in the left branch because it immediately knows that those solutions are a waste of our time. From this we can conclude that backtracking is an optimal approach for solving this problem in the least amount of steps.

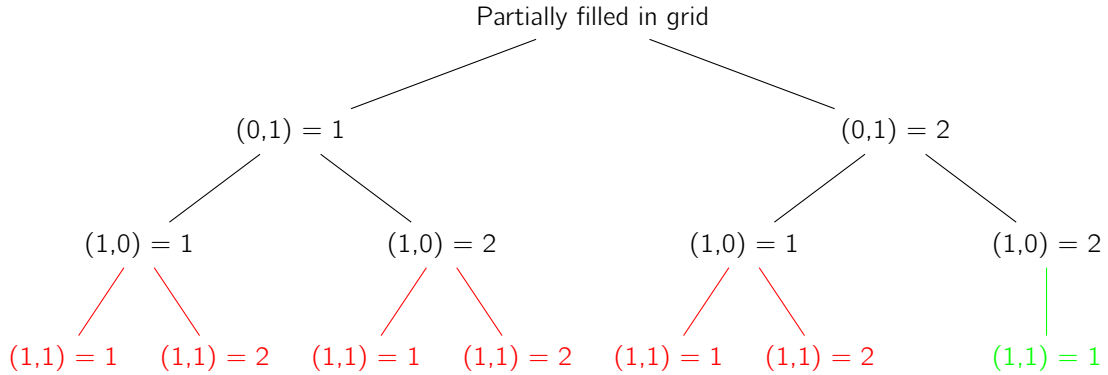

Figure 2: Search tree for exhaustive search traversal of figure 1

Greedy approach

A greedy approach for solving this CSP would try to make the best local decision for filling in every zero cell but would be unable to reverse a decision. This would mean that the algorithm would not terminate when one of the group constraints does not satisfy but it would continue while locally trying to minimize the number of groups that would not satisfy. This approach would not be optimal because it is not guaranteed to find a solution if it exists.

The search tree for this approach on our example in figure 1 would funnily enough be the same as our backtracking example in figure 3 and would in this case find the solution. This is because we never encounter a local decision where multiple options would still result in a valid grid. For an example where it would not find the solution, we change the groups to $M(G_1) = \{(0,0),(0,1)\}$, $M(G_2) = \{(0,1),(1,1)\}$ and $M(G_3) = \{(1,0),(1,1)\}$. Figure 4 shows the search tree for this example. Here we see that the algorithm has two options at $(1,0)$ that still result in a valid grid so it just pick the one with the lowest number and by doing so gets stuck in a situation where it it unable to find the solution.

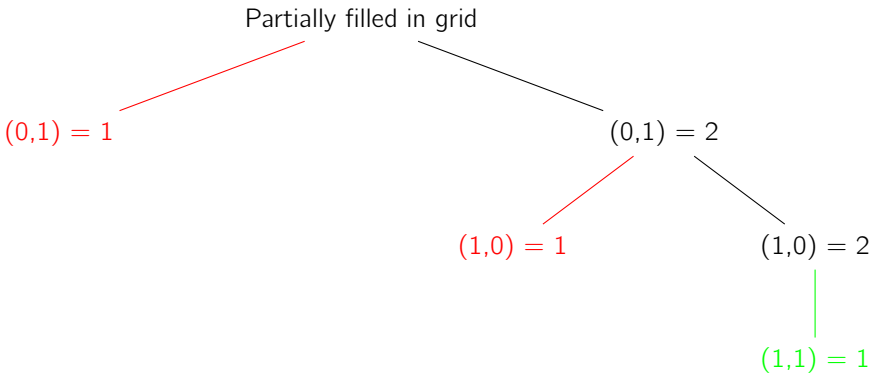

Figure 3: Search tree for backtracking traversal of figure 1

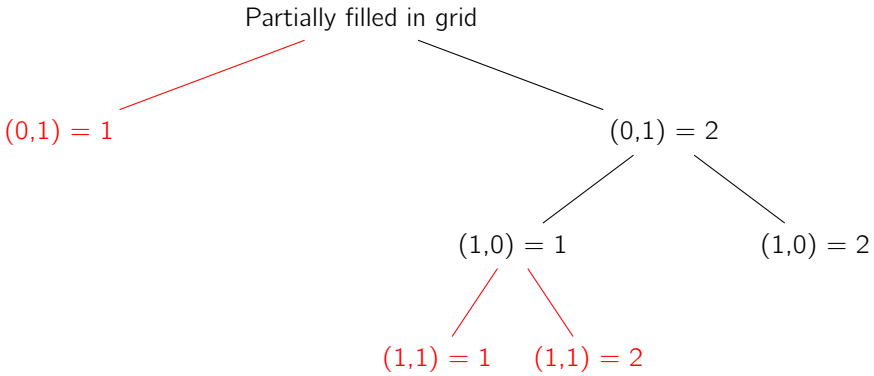

Figure 4: Search tree for greedy traversal of figure 1
with different groups

Summary and Discussion

The goal of this assignment was to get a good understanding of the difference between algorithmic strategies for solving problems. By implementing a exhaustive search function for this relatively simple problem and comparing it to backtracking and greedy approaches we were able to experience the nuances of each of them first hand in a practical way. We have observed that both exhaustive search and backtracking are optimal and therefore always find a solution if it exists but backtracking is more efficient in running time. The greedy approach on the other hand does not always find a solution if it exists and would be better suited for situations where exhaustive search or backtracking would take too long and it is not critical to find the optimal solution every time.

Contributions

Daniël Zee (s2063131):
Report: Exhaustive search and backtracking approach, Search trees, Summary and Discussion
Code for functions: satisfies_sum_constraint, satisfies_count_constraint, satisfies_group_constraints, search
Joris Nouwens (s1278002):
Report: Introduction Report, Greedy approach, Summary and Discussion
Code for functions fill_cell_to_groups, search and test_private.py