Programming Assignment 1 — Exhaustive Search

Many real-world problems can be formulated as *Constraint Satisfaction Problems* (CSPs), where we have a set of constraints that a valid solution must satisfy. Examples of such CSPs include planning and scheduling problems, but also the n-Queens problem, crossword puzzles, and sudokus. In this assignment, you will program a recursive exhaustive search algorithm called that aims to find solutions for the CSP described below.

Description

We have a 2D grid consisting of empty cells (with a default value of 0) that need to be filled out with non-zero values from a set of k permissible numbers $N = \{n_1, n_2, \ldots n_k\}$. You can think of these numbers as the weights of various types of objects that need to be placed on the 2D surface. The goal is to fill the entire 2D grid whilst satisfying the *group constraints* that are specified by the user. Every group constraint holds for a given group G, which is a set of (y, x) locations consists of two components/coordinates. For a given group G_i, the following items are specified by the problem:

- **Group members**: The group members $M(G_i)$ that are tied to the constraints below. The group members are a set of locations on the 2D grid.
- **Maximum sum**: This constraint specifies that the sum of values of all group members may not exceed a threshold $S(G_i)$ within group G_i.
- **Maximum count**: This constraint specifies that any value n_i may not occur more than $C(G_i)$ times in group G_i.

(0,0)	(0,1)	(0,2)
(1,0)	(1,1)	(1,2)
(2,0)	(2,1)	(2,2)

(a) The coordinate system

1	2	2
0	0	0
0	0	0

(b) Partially filled in grid

Figure 1: An example of a 2D grid with 2 groups, indicated by the two different colors. The 2 bottom-left cells are not part of a group. Empty cells contain 0s by default and still have to be filled in. A given cell can be part of multiple groups.

An example is shown in Figure 1, where empty cells in the right subfigure contain zeros. In this example, there are two groups G_1 and G_2 as indicated by the two colors. The members of these groups are denoted as $M(G_1) = \{(0,0), (0,1), (0,2)\}$, and $M(G_2) = \{(1,0), (1,1), (1,2), (2,2)\}$. The sum constraints of the groups are indicated by $S(G_1)$ and $S(G_2)$. This means that the sum of the values of $M(G_1)$ and $M(G_2)$ may not exceed $S(G_1)$ and $S(G_2)$, respectively. Moreover, the count constraints of the groups are indicated by $C(G_1)$ and $C(G_2)$. They tell us that it is

not allowed that any value n_i can occur more than $C(G_1)$ times in group 1 and no more than $C(G_2)$ times in G_2. **A grid is a valid solution if it is completely filled with non-zero values and all of the group constraints are satisfied.** Note that in general, a cell may belong to an arbitrary number of groups.

Suppose that $C(G_1) = 1$. In that case, the right subfigure can never be a solution because the value 2 occurs twice in group G_1. If $C(G_1) = 2$, however, it is still possible that the right subfigure can lead to a valid solution. Similarly, if $S(G_1) = 3$, that means that the right subfigure can never be a valid solution as the sum of the members (1+2+2=5) is larger than the maximum allowed sum (3).

Programming

For the project description, you are given an empty or partially filled grid, together with the specification of the groups and the respective constraints. The members of every group represented as a list of locations that constitute members of the group. The sum and count constraints are also represented as a list of tuples in the form [($S(G_1)$,$C(G_1)$), ($S(G_2)$,$C(G_2)$),...]. The goal is to find a completed grid (fully filled) that satisfies all of the constraints for all of the groups. When there is no solution your program should return None.

We provided a template program. It consists of two files:

- `csp.py`: The main file in which you will implement the solution.
- `test_usecases.py`: A Unit Test file with various test cases that help you check whether your implementations of the different functions are correct.

The file `csp.py` contains various functions that are not implemented yet. Note that you are not required to implement `start_search` as this function is already completed. Read the written documentation about these functions, and implement these functions. Most of these functions require less than 10 lines of code. **Do not change the headers of the functions provided.** You are allowed to add functions yourself, if you feel that that makes it easier. Note, however, that points are deducted if we think that they are unnecessary. Make sure to document these consistently.

The file `test_usecases.py` consists of several test functions. We provided these functions to help programming and testing the code. By running the following command, the unit tests can be executed:

```
python -m unittest test_usecases.py
```

Make sure to have the right packages installed. Do not use other packages than those present in the template. Feel free to add more unit tests. Do not alter the unit tests that are provided. If your program does not succeed on all unit tests that are provided, it is likely that there is still a problem in your code. Make sure that all other unit tests succeed, before submitting the code.

Additionally, also hand in a file named `test_private.py`. In here, you can create additional unit tests to verify the working of your program. Write at least 3 additional unit tests, and hand these in along with the assignment.

Also keep in mind that all unit tests should be able to run within a matter of seconds on any computer.

Report

Write a report in LaTeX(at most 3 pages) using the provided template (see Brightspace), addressing the following points / research questions:

- Introduction: describe the problem. Describe a state and action **explicitly** in the context of this problem.
- Explain the difference between exhaustive search and backtracking. What makes the implemented approach exhaustive search? How would you change to make it do backtracking instead?
- Draw the search tree of a small example that exhaustive search traverses. Additionally, draw the same search tree, but then for backtracking. What can we conclude? Is backtracking optimal (guaranteed to find a solution if it exists)? You are allowed to draw this on paper and scan the result.
- Think of a greedy approach to search for a solution for a given CSP. How well does this greedy approach work? Is your approach optimal? Explain why it is or why it is not. Include an adversarial test-case in your report where the greedy approach does not yield an optimal schedule. If your approach is optimal, you do not need to do this.
- Summary and Discussion. What was the goal of the assignment? What have you done and observed? (think about: backtracking vs exhaustive search, optimality, running time, greedy approach vs exhaustive search and backtracking). Do not write about your personal experience and stories. Keep it scientific and simply summarize the report, making observations about the algorithms.

Work distribution

At the end of the report, include a distribution of the work: who did what? By default, we give both group members the same grade, but if there is a serious imbalance in the workload distribution, we may have to take action. The work distribution does not count towards the page limit.

Submission

Submit your assignment through Brightspace by submitting the following files:

- report.pdf (the report)
- csp.py (your solution code)
- test_private.py (your unit tests)

The deadline for this assignment is **the 17th of March 2023, 23:59 CET.**