National University of Singapore
School of Computing
CS3243 Introduction to Artificial Intelligence

Project 2.2: Local Search and Constraint Satisfaction Problems

Issued: 16 September 2024 Due: 20 October 2024, 2359hrs

1 Overview

In this project, you will **implement local search and constraint satisfaction problem (CSP)** algorithms to find valid goals for various tasks. Specifically, you must implement the following.

1. A **Local Search** algorithm (e.g., hill-climbing)
2. A **CSP** solver (e.g., backtracking)

This project is worth 7% of your module grade, with an extra 3% bonus.

1.1 General Project Requirements

The general project requirements are as follows:

- **Individual** project: Discussion within a team is allowed, but **no code should be shared**
- Python Version: ≥ **3.12**
- Deadline: **20 October 2024, 2359hrs**
- Submission: Via **Coursemology** – for details, refer to the Coursemology Guide on Canvas

1.2 Academic Integrity and Late Submissions

Note that any material that does not originate from you (e.g., is taken from another source) should not be used directly. You should do up the solutions on your own. Failure to do so constitutes plagiarism. Sharing of code between individuals is also strictly not allowed. Students found plagiarising will be dealt with seriously.

For late submissions, there will be a 20% penalty for submissions received within 24 hours after the deadline, 50% penalty for submissions received between 24-48 hours after the deadline, and 100% penalty for submissions received after 48 hours after the deadline. For example, if you submit the project 30 hours after the deadline and obtain a score of 92%, a 50% penalty applies, and you will only be awarded 46%.

2 Project 2.2: Facing the Music

2.1 Task 1: Local Search

2.1.1 Description

You have been hired as a concert manager for an upcoming music festival. One of your jobs is to schedule performances. However, you are required to schedule performances in a balanced manner; scheduling all the popular performers on the same day will mean more people attending that day while other days will see fewer festival goers. To make matters worse, the number of performers scheduled to perform on each day of the festival must be the same to ensure that the attendees on any day do not feel cheated.

To determine the popularity of the performers, you conducted extensive surveys. The results from these surveys show the number of fans supporting each performer, and thus determine their popularity. Your job is now to satisfy the attendees by dividing the number of performers equally across all days, while ensuring the number of fans supporting the performers are also the same across all days.

2.1.2 Functionality

You will be given a list of values to be partitioned. In addition, you will also be given the number of subsets required to partition the list into, and the required size of each subset. You are required to partition the values in the list such that the total sum of the values in each subset and the size of each subset is the same, and every value is to be used exactly once.

2.1.3 Input Constraints

You will be given a `dict` with the following keys:

- `count`: The number of subsets required. Type is `int`.
- `size`: The required size of each subset. Type is `int`.
- `values`: The values that need to be partitioned. Type is `list`. The elements in this list will be of type `int`.

2.1.4 Requirements

You will define a Python function, `search(input)`, that takes in the input dictionary described in Section 2.1.3. The objective is to return a list of subsets that fulfils the requirements specified in Section 2.1.2. More specifically, the objective is to return a `list`, where each element in this outer `list` is a `list`, and each element in each nested (i.e., inner) `list` is an `int`. The order of integers in each nested list and the order of the nested lists do not matter.

2.1.5 Assumptions

You may assume the following:

- **At least 1 (ONE) solution exists**. Hence, you may assume the sum of all values is a multiple of `count`.
- Each element in `values` is a positive `int`; these element values may not be unique – i.e., some elements may share the same `int` value as other elements.
- The size of `values` is exactly the product of `count` and `size` – i.e., we have: `len(values) == count * size`.

It is crucial that your implementation adopts a local search to arrive at a solution. Failure to do so (e.g., implementing a CSP solver instead) will result in 0 marks given for this task.

2.1.6 Example

The input below requires the 20 values to be divided into 5 subsets, each of size 4.

```
input = {

'count' : 5,

'size' : 4,

'values' :  [  7,  8,  4, 13, 12, 19,  9,  6,  6,  1,
               3, 13, 10, 14,  3, 19,  8, 10,  32, 3 ]

}
```

The following are two possible solutions.

```
output1 = [
            [  3, 10, 13, 14 ], [  3,  8, 10, 19 ],
            [  6,  7,  8, 19 ], [  6,  9, 12, 13 ],
            [ 32,  1,  4,  3 ]
]

output2 = [
            [  3, 10, 13, 14 ], [  3,  6, 12, 19 ],
            [  6,  7,  8, 19 ], [  8,  9, 10, 13 ],
            [ 32,  4,  3,  1 ]
]
```

Do note that there may be other solutions as well!

2.2 Task 2: CSP

2.2.1 Description

Following your success in scheduling performers in the concert, you are now faced with another division problem. To facilitate the assignment of ushers and security personnel, the concert hall must be divided into various square regions, with each team overseeing one region. More experienced teams will be responsible for a larger area, while newer teams will be responsible for a smaller area. There are also some emergency escapes within the concert area, where no guests are allowed to sit. Naturally, it would be redundant to ask ushers or security personnel to patrol in those areas.

2.2.2 Functionality

You will be given the dimensions of a target rectangle (i.e., the size of the concert hall). You will also be given the number and dimensions of several input squares, as well as a list of obstacle coordinates. You must arrange the input squares a way such that the whole rectangle is covered except at the positions of the obstacles. No squares are to be placed over the obstacles.

An example is given in Section 2.2.6 below.

2.2.3 Input Constraints

You will be given a `dict` with the following keys:

- `rows`: The number of rows in the target rectangle. Type is `int`.
- `cols`: The number of columns in the target rectangle. Type is `int`.
- `input_squares`: The side lengths and numbers of input squares. Type is `dict`. The keys of each element in this dict is an `int`, representing the side length of the input square. The values of this dictionary will be another `int`, representing the number of squares of that particular size that are available.
- `obstacles`: A list of coordinates that must not be covered by any squares. Type is `list`. The elements in this list will be a `tuple[int, int]`, representing the location of the obstacle as **(row, col)**.

2.2.4 Requirements

You will define a python function, `solve_CSP(input)`, that takes in the input dictionary described above. The objective is to return an arrangement of input squares that fulfils the requirements specified in Section 2.2.2. More specifically, you should return a `list`, where each element in the `list` is a `tuple[int, int, int]`. The first element in the tuple will represent the size of a square in the final arrangement, and the next two elements represent the **row and**

column of the **top-left corner of the same square**.

An example is given below in Section 2.2.6.

2.2.5 Assumptions

You may assume the following.

- **At least** 1 (ONE) solution exists. Further, you may assume the sum of the areas of the input squares and the obstacles is exactly equal to the area of the target rectangle.
- All dimensions (of the target rectangle and the input squares) are given as integers. Hence, you do not have to consider cases where a square is placed at non-integral coordinates.
- If a key does not appear in the `input squares` dict, no squares of that size are to be used to tile the target rectangle.

2.2.6 Example

For this problem, all coordinates are given in matrix coordinates, (row, col). Figure 1 below corresponds to a 4×4 board with the respective coordinates.

(0, 0)	(0, 1)	(0, 2)	(0,3)
(1, 0)	(1, 1)	(1, 2)	(1, 3)
(2, 0)	(2, 1)	(2, 2)	(2, 3)
(3, 0)	(3, 1)	(3, 2)	(3, 3)

Figure 1: 4×4 square labelled with matrix coordinates

Suppose that we are given the input specified below.

```
input = {   'rows' : 4
            'cols' : 8
            'input_squares': { 1:5, 2:2, 4:1 }
            'obstacles'    : [ (3,0), (1,2), (3,7) ]    }
```

In the above example input, the specified target rectangle has 4 rows and 8 columns. Further, it states that the target rectangle must be filled with 5 squares of side length 1, 2 squares of side length 2 and

1 square of side length 4. There are also 3 obstacles are at positions (3, 0), (1, 2), and (3, 7).

One possible solution is given below.

```
output = [ (2,0,0), (1,0,2), (4,0,3), (1,0,7),
           (1,1,7), (1,2,0), (2,2,1), (1,2,7) ]
```

This output corresponds to the arrangement shown in Figure 2 below.

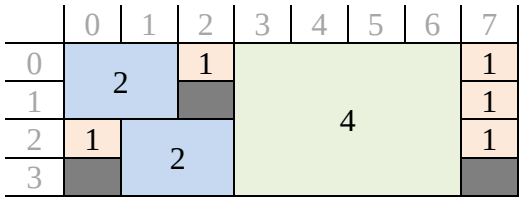

Figure 2: A depiction of **`output`**. Notice that the two squares of length 2, which are highlighted in blue, correspond to `(2,0,0)`, and `(2,2,1)` – i.e., the squares at (row, col) coordinates (0, 0) and (2,1) respectively. Also note that the darkened squares are obstacles.

3 Grading

3.1 Grading Rubrics (Total: 7 marks + 3 bonus marks)

Requirements (Marks Allocated)	Total Marks
• Correct implementation of the Local Search algorithm evaluated by passing all public test cases and hidden test cases (3m). • Correct implementation of Backtracking algorithm evaluated by passing all public test cases and hidden test cases (4m).	7
• Highly optimised implementation of Local Search algorithm evaluated by passing all bonus test cases (1m). • Highly optimised implementation of Backtracking algorithm evaluated by passing all bonus test cases (2m).	3

3.2 Grading details

3.2.1 Local search

There are a total of fourteen correctness test cases, eight efficiency test cases, and two bonus test cases. Of the regular test cases, four test cases are public and ten are private. All efficiency and bonus test cases are private test cases.

- Public test cases 1 – 4 are worth 0.1m each.
- Private test cases 1 – 10 are worth 0.1m each.
- Efficiency test cases 1 – 8 are worth 0.2m each.
- Bonus test cases 1 – 2 is worth 0.5m each.

3.2.2 CSP

There are a total of ten regular test cases and three bonus test cases. Of the regular test cases, six test cases are public, and the rest are private. All three bonus test cases are private test cases.

- Public test cases 1 – 4 are worth 0.25m each.
- Public test cases 5 – 6 are worth 0.5m each.
- Private test cases 1 – 4 are worth 0.5m each.
- Bonus test cases 1 – 2 are worth 0.5m each.
- Bonus test case 3 is worth 1m.

Note that the bonus test cases are not required for you to achieve full credit in the project. Any marks obtained from the bonus test cases can be used to make up for marks lost across any of the projects (i.e., the bonus marks may only be applied to other projects (i.e., Project 1.1, 1.2, 2.1, 2.2, or 3), and not to other forms of (non-project) assessments (e.g., midterm or final)).

---

4 Submission

4.1 Details for Submission via Coursemology

Refer to **Canvas** > **CS3243** > **Files** > **Projects** > **Coursemology_guide.pdf** for submission details.

---

5 Appendix

5.1 Allowed Libraries

The following libraries are allowed:

- Data structures: queue, collections, heapq, array, copy, enum, string
- Math: numbers, math, decimal, fractions, random, numpy
- Functional: itertools, functools, operators
- Types: types, typing