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

Project 2.2: Local Search and Constraint Satisfaction Problems

Issued: 19 February 2024 Due: 24 March 2024

1 Overview

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

1. Implement a **Local Search** algorithm
2. Implement a **CSP** solver

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, but you are *allowed to consult P2ST (ChatGPT) App*.
- Python Version: $\geq$ **3.10**
- Deadline: **24 March 2024**
- Submission: Via Coursemology. Refer to Canvas → CS3243 → Files → Projects → coursemology_guide.pdf for details.

1.2 Academic Integrity and Late Submissions

Note that any material that does not originate from you (e.g., is taken from another source, with the exception of the P2ST (ChatGPT) app) should not be used directly. You should do up the solutions on your own. Failure to do so constitutes plagiarism. Sharing of materials between individuals is also strictly not allowed. Students found plagiarising or sharing their code 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 less 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 above. The objective is to return a list of subsets that fulfils the requirements specified in 2.1.2. In particular, you should return a `list`, where each element in the `list` is a `list`, and each element every inner `list` is a `int`. The order of integers in the inner list, and the order of the inner lists do not matter.

An example is given below in 2.1.6.

2.1.5 Assumptions

You may assume the following:

- **At least** 1 solution exists. Hence, you may assume the sum of all values is a multiple of `count`.
- All values are positive integers. Values in the list are not unique, and the same value might appear more than once.
- You will not be given any extra values. Specifically, `len(values) == count * size`

It is crucial that your implementation uses local search to arrive at a solution. Failure to do so (eg: implementing as a CSP) will result in 0 marks given for this task.

2.1.6 Example

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

```
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]
}
```

Two possible solutions are shown.

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

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

You may be able to find other answers 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 being in charge of one region. More experienced teams would 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 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 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 2.2.2. In particular, 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 2.2.6.

2.2.5 Assumptions

You may assume the following:

- **At least** 1 solution exists. Hence, 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

In this question, all coordinates are given in matrix coordinates, (row, col). A 4x4 board is labelled below with their 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: 4x4 square labelled with matrix coordinates

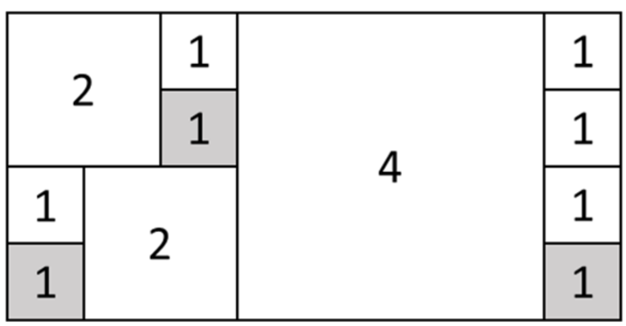
The below input requires a rectangle with 4 rows and 8 columns to be filled with 5 squares of side length 1, 2 squares of side length 2 and 1 square of side length 4. The obstacles are at positions (3,0), (1,2) and, (3,7).

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

One possible solution is shown.

```
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 corresponds to the following arrangement (darkened squares are obstacles):



3 Grading

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

Requirements (Marks Allocated)	Total Marks
• Correct implementation of Hill Climbing 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 regular 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 **ALL** the projects, including projects 1 and 3.

4 Submission

Refer to Canvas → CS3243 → Files → Projects → coursemology_guide.pdf for details.

Note that although the bonus test cases exist as a seperate task on coursemology, you are to submit the same code for the regular CSP task and the bonus task.

4.1 P2ST (ChatGPT) App Usage

The P2ST (ChatGPT) app has been developed for CS3243 (this course). It is a tool that can be used to generate code from natural language descriptions. The objective is to help you better understand the contents of the course while skipping some of the more tedious parts of coding.

You are **permitted** to use the P2ST (ChatGPT) app to generate code to help you with this project. No other app or AI-generated code may be used.

The plagiarism-checking phase will be conducted using a tool that can identify code that has been copied from other sources. If your code is flagged as duplicated, you may appeal to the teaching team only if the code was generated from or with the help of the P2ST (ChatGPT) app. If the teaching team finds that the code was not generated using the P2ST (ChatGPT) app and is instead generated via other means like using other AI assistance tools, plagiarising other people's work, and more, the appeal will not be successful.

Note that the correctness and efficiency of the code generated by the app are not guaranteed. You are responsible for any submissions made.

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