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

Project 2.2: Local Search and Constraint Satisfaction Problems

Issued: 18 September 2023 Due: 22 October 2023

1 Overview

In this project, you will **implement local search and constraint satisfaction problem (CSP)** algorithms to find valid states in a game. 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 2% 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: **22 October 2023**
- Submission folder: **Canvas** $>$ **CS3243** $>$ **Assignments** $>$ **Project 2.2** More details can be found in Section 4.

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: Patchwork Blanket

The air-conditioning in a school is often set to very low temperatures. To keep warm, some of the students had the idea to collaborate and create a blanket. One day, each student brought some leftover pieces of fabric from home. However, due to a oversight, the pieces of fabric brought were of varying sizes. The students had no scissors with them, and being environmentally conscious, they did not want to waste any material. Being the top programmer in the group, you have been entrusted with the task of arranging the fabric pieces to form a rectangular blanket that utilises all the materials.

2.1 Task 1: Local Search

2.1.1 Functionality

You will be given the dimensions of a target rectangle which you need to fill. You will also be given the numbers and dimensions of the input squares which you can use to fill the target rectangle. The objective is to find an arrangement of the input squares such that they fill the target rectangle completely without overlaps.

2.1.2 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.

2.1.3 Requirements

You will define a python function, `run_local(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 2.1.1. 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 2.1.5.

2.1.4 Assumptions

You may assume the following:

- **At least** 1 solution exists. Hence, you may assume the sum of the areas of the input squares 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.

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.

2.1.5 Example

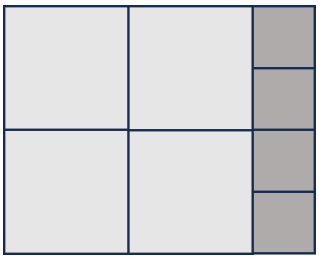
The below input requires a rectangle with 4 rows and 5 columns to be filled with 4 squares of side length 1 and 4 squares of side length 2.

```
input = {
    'rows' : 4,
    'cols' : 5,
    'input_squares' : {
        1 : 4,
        2 : 4
    }
}
```

One possible solution is shown.

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

The output above corresponds to the following arrangement of squares.



2.2 Task 2: CSP

2.2.1 Functionality

The requirements for this task are the same as the task specified in 2.1.1, with one other inclusion: a list of obstacle squares will be given that must not be filled.
Specifically, you will be given the dimensions of the target rectangle to be filled, the numbers and dimensions of input squares, and 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.

2.2.2 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)**.

2.2.3 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 2.2.1. 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 2.2.5.

2.2.4 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.

2.2.5 Example

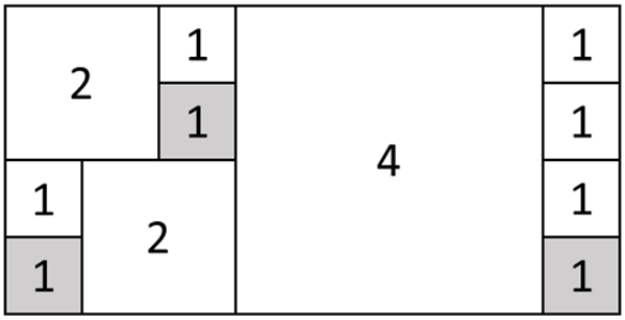
The below input requires a rectangle with 4 rows and 8 columns to be filled with 5 squares of side length 1, 2 squares of side length 2 and 1 square of side length 4. The obstacles are at positions (3,0), (1,2) and, (3,7).

```
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 + 2 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 Backtracking Algorithm evaluated by passing all bonus test cases (2m).	2

3.2 Grading Details

3.2.1 Correctness

We will run your code on a set of public and private test cases. There are 4 possible outputs when grading an implementation on a given test case:

- **Accepted (AC):** You have returned a solution passing all the requirements within the given time limit. You have **passed** the test case.
- **Wrong Answer (WA):** You have returned a solution failing at least one requirement within the given time limit. You have **not passed** the test case.
- **Time Limit Exceeded (TLE):** Your code runs beyond the given time limit. You have **not passed** the test case.
- **Runtime Error (RTE):** Your code has thrown an exception or caused an exception to be thrown. You have **not passed** the test case.

Different test cases may have different weights. You will get full credit for the implementation only if you obtain AC for all test cases using that implementation. For the specific points distribution, refer to Section 5.2.

Note that due to the **random nature of local search**, occasional failing of the test cases is acceptable. Your local search solution is therefore required to pass the test cases **at least half the time** ($\geq$

50%), a relaxation of our usual requirements of 100% of the time. Unfortunately, this condition *will not be checked by CodePost*.

Do not submit the same code multiple times e.g. submitting your CSP implementation as a local search solution. It is fairly easy to identify such cases. **Any violation of this rule will result in 0 marks for the affected implementations**.

3.3 CodePost (Platform for Code Testing)

We will be using CodePost as our standardised platform for you to run and test your code on public and private testcases. You should have used CodePost to test your implementations for Project 1. Do inform the course staff if you face any issues accessing or uploading to CodePost.

1. For each task (Local/CSP), click on "Upload assignment", and upload your Python file `local.py/csp.py` (DO NOT rename the files).
2. After the submission has been processed, refresh the page and select "**View feedback**".
3. Ideally, if your implementation is correct and is within the runtime threshold, the output will look like this:

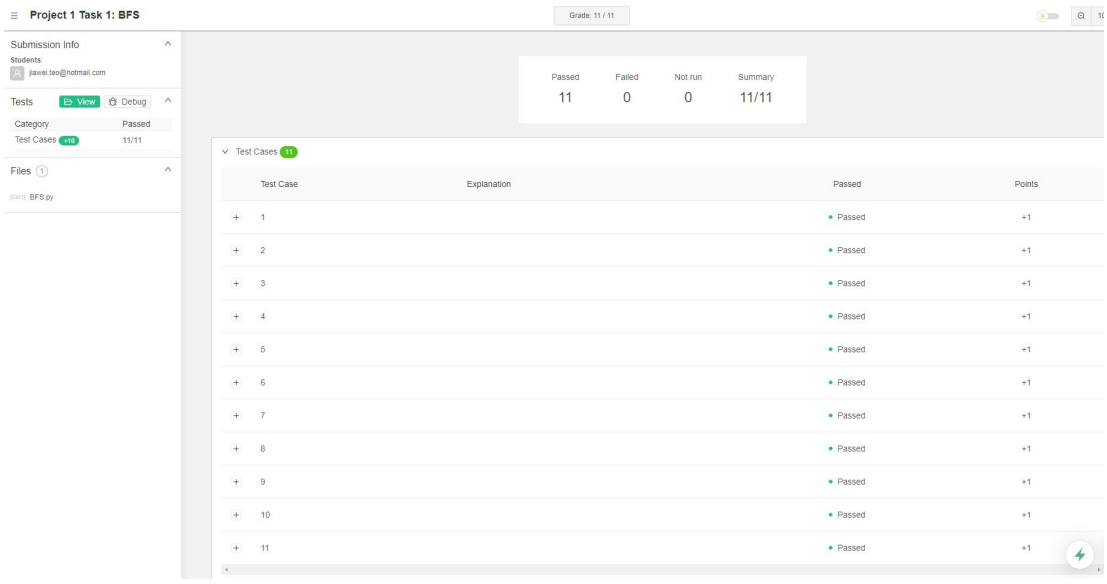


Figure 1: Codepost output

3.3.1 CodePost for Testing

CodePost hosts both the public test cases (which have been released via Canvas together with the skeleton implementation files) and the private test cases. The output in CodePost has been purposefully sanitised to prevent attempts at finding out the private test cases. As such, you are **expected to check your implementations thoroughly** via **custom-made test cases run locally**, as you will not find CodePost useful for debugging purposes.

Note that CodePost is for you to run and test your code on the test cases. Your solution will **not be graded** using CodePost; instead, your Canvas submission will be used and run on the same environment as CodePost. Therefore, passing all test cases in CodePost **does not guarantee** full credit. Some possible reasons for not obtaining full credit despite passing all test cases in CodePost include: being lucky in CodePost due to randomness in the algorithm, being caught plagiarising, etc. **We will check for any plagiarism and students found plagiarising will be dealt with seriously.**

Note that CodePost is run by an external organisation – we are not responsible for any downtime.

---

4 Submission

4.1 Submission Details via Canvas

- For this project, you will need to submit 2 Python files: `local.py` and `csp.py`.
- Place all files required in a folder, name the folder as `studentNo` and zip it as `studentNo.zip`. An example will be: `A0123456Z.zip`.
- When unzipped, your submission file must contain the 2 files in a folder: **A0123456Z.zip** $\rightarrow$ **unzip** $\rightarrow$ **A0123456Z folder** $\rightarrow$ **local.py, csp.py**. There should not be any subfolders.
- Do not modify the file names.
- Please follow the instructions closely. **If your files cannot be opened or if the grader cannot execute your code, this will be considered failing the test cases as your code cannot be tested.**
- Submission folder: **Canvas** $>$ **CS3243** $>$ **Assignments** $>$ **Project 2.2**

You may submit your files as many times as you like, but only the latest one will be graded, **even if it means incurring a late penalty**.

4.2 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

5.2 Points Distribution

5.2.1 Local Search

1. Correctness: each test case is worth 0.25 points. Total is 1 point.
2. Efficiency: each test case is worth 0.25 points. Total is 2 points.

5.2.2 CSP

1. Correctness: each test case is worth 0.25 points. Total is 1 point.
2. Efficiency: each test case is worth 0.5 points. Total is 3 points.
3. Bonus: bonus test cases 1 and 2 are worth 0.5 points each. Bonus test case 3 is worth 1 point. Total is 2 points.