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

Project 2: Local Search and Constraint Satisfaction Problems

Issued: 16 September 2022 Due: 16 October 2022

Overview

In this project, you will implement local search and Constraint Satisfaction Problem (CSP) algorithms to find valid goal states in a game. Specifically, you are tasked to implement the following algorithms:

1. Implement the **Local Search Algorithm (Hill-Climbing**.
2. Implement the **CSP Backtracking Algorithm**.

This project is worth 10% of your module grade.

General Project Requirements

The general project requirements are as follows:

- **Individual** project
- Python Version: **3.7**
- Submission deadline: **16 October 2022 (Wednesday) 11:59pm**
- Submission folder: **LumiNUS** > **CS3243** > **Projects** > **Project 2 Submission folder**
- Submission format: One standard (non-encrypted) **zip file**[1] containing only the necessary project files. In particular, it should unzip to give one folder with two `.py` files: `Local.py` and `CSP.py`. More information is given in the Submission Details section below.

As to the specific project requirements, you must complete and submit the following:

- Task 1 (Local Search): Implement your Local Search algorithm in `Local.py`.
- Task 2 (CSP): Implement your CSP Backtracking algorithm in `CSP.py`.

Note that you are tasked to implement your own search space and state structure for the game. **You are strongly encouraged to reuse your search space and state definition from Project 1.**

---

[1] Note that it is the responsibility of the students to ensure that this file may be accessed by conventional means.

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 materials between individuals is also strictly not allowed. Students found plagiarising or sharing their code will be dealt with seriously. Additionally, sharing of code on any public repository is strictly not allowed and any students found sharing 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%.

Background: n-Queens Puzzle

The n-queens puzzle is the problem of placing n chess queens on an $n \times n$ chessboard so that no two queens threaten each other; thus, a solution requires that no two queens share the same row, column, or diagonal. The problem was first posed in the mid-19th century. In the modern era, it is often used as an example problem for various computer programming techniques. Solutions exist for all natural numbers n with the exception of $n = 2$ and $n = 3$. Although the exact number of solutions is only known for $n \leq 28$, the work of Simkin in 2021 established that the asymptotic growth rate of the number of solutions is $(0.143n)^n$.

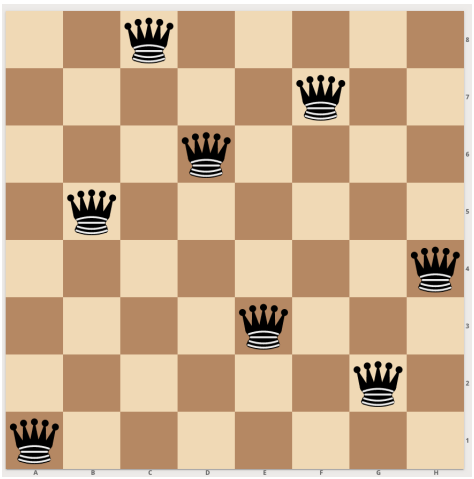

Figure 1: A layout of the 8-queens puzzle.

In this project, we will be solving a generalised $n - pieces$ puzzle. Instead of using only Queen pieces, we will be using all the different chess pieces (King, Queen, Bishop, Rook, Knight, Ferz, Princess, Empress) as stated in Project 1 (excluding Pawn piece). Additionally, there may also be obstacles placed on the board (Same as Project 1). The objective of the game is to find a goal state where no pieces threaten each other. As with Project 1, **obstacles prevent pieces from threatening another piece**, i.e., the threatened cells of a piece stop at an obstacle in the path. More details will be given in the later sections. For this project, the board will **not** have any action costs as we are not concerned with determining a sequence of moves. Instead, we are only interested in the goal state, i.e., the final positions of each piece in the board.

As the chess pieces and the game board are the same as Project 1, you may refer to the Project 1 specifications PDF for the movement rules of each piece as well as the basic board layout.

Hint: If you implemented the game board and pieces correctly in Project 1, you may reuse them as there should be minimal/no change to it for Project 2.

Input test file

Similar to Project 1, for each test case, you will be given an input text file containing the board configuration (e.g., number of rows/columns, type of game pieces, starting position of game pieces, etc.). A parser function has been defined for you to take in the input text file and generate the required variables.

In Project 2, there are two different types of test case files (e.g., `Local1.txt`, `Local2.txt`, ... and `CSP1.txt`, `CSP2.txt`, ...) – one type for Local Search and one type for CSP.

Getting Started

You are given two python files (`Local.py, CSP.py`) with the recommended empty functions/classes to be implemented. Specifically, the following are given to you:

1. `run_local/CSP()`: **DO NOT REMOVE THIS FUNCTION.** When called, this function must return a **Goal State**. The returned output will be evaluated by the autograder for correctness. More details on the output will be given in the later sections.

2. `parse(testcase)`: **DO NOT REMOVE THIS FUNCTION.** When called, this function parses the testcase text file given in its parameters and outputs the required information needed to formulate the search space.

3. `search()`: Implement the corresponding search/backtracking algorithm here. (You may change/remove this function as desired, as long as you keep the `run_XXX()` function and return the required values.)
4. `State`: A class storing information of the game state. (You may change/remove this function as desired, as long as you keep the `run_XXX()` function and return the required values.)
5. `Board`: A class storing information of the board. (You may change/remove this function as desired, as long as you keep the `run_XXX()` function and return the required values.)
6. `Piece`: A class storing information of a game piece. (You may change/remove this function as desired, as long as you keep the `run_XXX()` function and return the required values.)

You are encouraged to write other helper functions to aid you in implementing the algorithms. You are also encouraged, for both python files, to use the same classes to design the game board and pieces (i.e., `State, Board, Piece`); only the implementation of `search()` should differ. During tests, the input filename will be given as the first command line argument (use sys.argv[1] to get the input filename) and thereafter only the method `run_XXX()` will be called. The output should be a valid Goal State.

Finding a Valid Goal using Local Search/Backtracking

The objectives of this part of the project are:

1. To gain exposure in the implementation of algorithms taught in class.
2. To learn to recognise the differences and utility of each search algorithm.
3. To learn to recognise the effects that the different methods have on the performance and efficiency.

Project 2 Task 1: Overpopulation

The King of CS3243 has successfully escaped from the Kingdom of CS9999 and eliminated the CS9999 King, leading to the downfall of CS9999. Many years have passed and the kingdom of CS3243 has prospered due to the lack of competition from CS9999. More and more people from all over the world have migrated to the CS3243 Kingdom due to the opportunities and wealth that the kingdom offers. However, this has led to overpopulation in the Kingdom due to the lack of space to accommodate the surge in the number of citizens. To tackle this issue, the Council of CS3243 has decided to exile some of its citizens...

Background:

We are given a game board of varying size. Each such board is completely filled with chess pieces (and obstacles). The positions of the chess pieces and obstacles are given in the input test file. The pieces and obstacles will remain static at all times (i.e., they remain in the same position throughout the search). Each chess piece will threaten its surrounding position based on the type of chess piece it is. Note that whilst the chess pieces can threaten each other, they cannot capture/eliminate other pieces. An example of the initial state of the board is shown below:

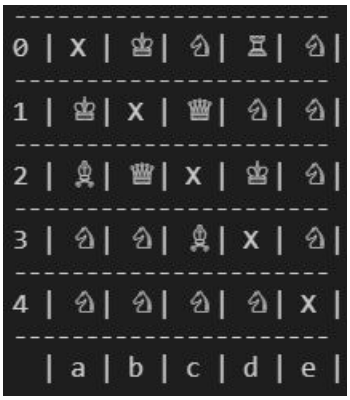

Figure 2: An initial state of the game with the the board being completely filled up .

We are also given a positive integer k (where $k \leq n$, and given that initially, there are n pieces on the board). The objective of the game is thus to remove at most $n - k$ pieces such that no two pieces on the board threaten each other. Hence, the goal is to find a board with at least k original pieces (i.e., pieces in their starting positions) on it, such that among these k original pieces, no piece is threatening another. The diagram below shows an example of a goal state of the game:

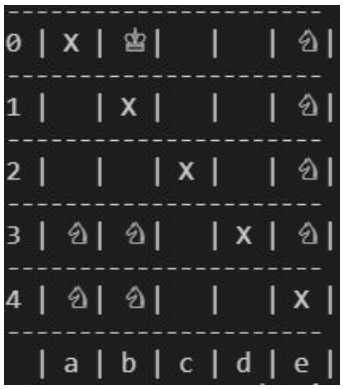

Figure 3: The goal state of the game with 8 pieces left on the board where they are not threatening one another.

Task 1: Local Search

Implement a **Local Search** algorithm (i.e., Hill-Climbing) to solve the problem stated above in `Local.py`. This can be done through random restarts or other variants of the algorithms specified.

Test file input: Use sys.argv[1] to obtain the name of the test file within the run_local() function. This has already been added for you in the template code.

When the `run_local()` function is executed, your code should return a **dictionary of valid positions** such that no two chess piece threaten one another on the board and each chess piece is on the board in the following format:
$\{pos_1 : piece_1, pos_2 : piece_2, ..., pos_n : piece_n\}$

Each position pos_i and piece $piece_i$, is a `(key:value)` pair. The `key` represents a grid index tuple, (x, y), where x is the column index (i.e., a string), and y is the row index (i.e., an integer), such that (x, y) corresponds to a specific grid cell on the board that we wish to reference. The `value` corresponds to a string identifying a particular type of chess pieces (i.e., "King", "Queen", "Bishop", "Rook" or "Knight" - do note that the first letter is capitalised).

More specifically, the positions of the pieces at its goal state should be returned in this format:
$\{(x_1, y_1) : piece_1, (x_2, y_2) : piece_2, (x_3, y_3) : piece_3, ..., (x_j, y_j) : piece_j\}$

An example of the function printed and its output is shown below:
`print(run_local())`

Sample Output:
$\{('a', 0) : Queen, ('b', 3) : King, ('c', 7) : Rook, ..., ('d', 5) : Knight\}$

Project 2 Task 2: Repopulation

In order to reduce the probability of a revolt, the council from the CS3243 Kingdom has decided to relocate the citizens it has exiled. Being a new settlement, the exiled population would have had no difficulties with the relocation. However, as fate would have it, they are stricken by an infectious disease (Covid-99). The citizens must now practise safe distancing amidst their relocation to reduce the chance of transmission.

Background:

We are given a game board of varying size. Each board is initially empty (except for any obstacles). The number of each chess piece are given in the input test file. An example of the initial state of the board is shown below:

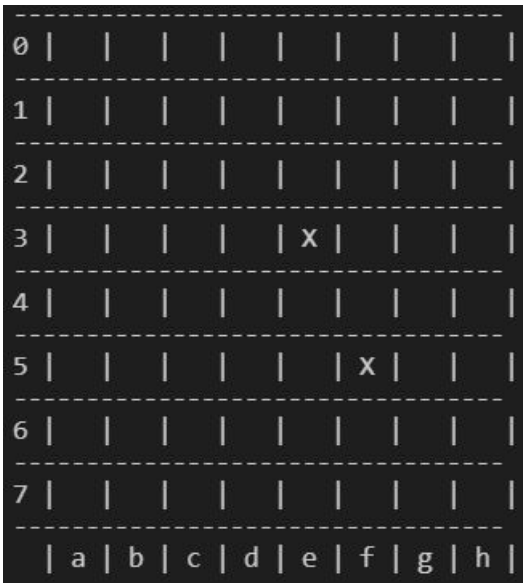

Figure 4: An initial state of the game with the the board being completely empty and two obstacles labelled 'X'.

We are also given a set of chess pieces to populate the board with (i.e., the input test file specifies how many pieces of each kind must be placed). The objective of the game is to find positions on the given board for each piece specified in the given set of chess pieces such that no two pieces on the board threaten each other. This implies that the goal is to find a board where all chess pieces specified in the initial set are placed such that no piece is threatening another. The diagram below shows an example of a goal state of the game:

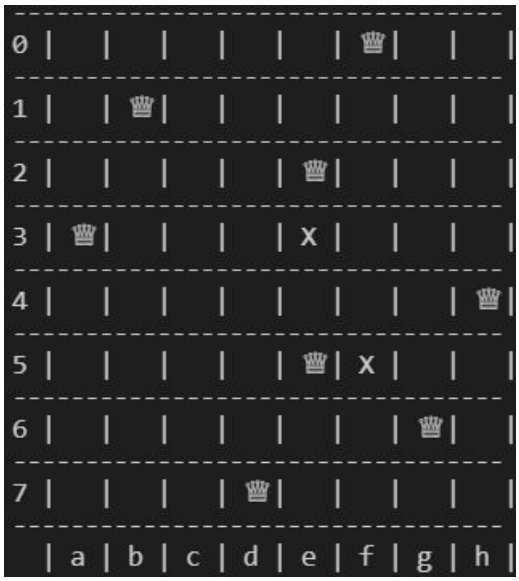

Figure 5: The goal state of the game with no pieces threatening one another and all pieces (8 Queens) stated in the input test file are placed in the goal state.

Task 2: Backtracking

Implement the **Backtracking** algorithm to solve the problem stated above in `CSP.py`. You are free to use any heuristics and inference algorithm to speed up the search.

Test file input: Use sys.argv[1] to obtain the name of the test file within the run_CSP() function. This has already been added for you in the template code.

When the `run_CSP()` function is executed, your code should return a **dictionary of valid positions** such that no two chess piece threaten one another and all the given chess pieces have been placed on the board. The required format is:
$\{pos_1 : piece_1, pos_2 : piece_2, ..., pos_n : piece_n\}$

Each position pos_i and piece $piece_i$, is a `(key:value)` pair. The `key` represents a grid index tuple, (x, y), where x is the column index (i.e., a string), and y is the row index (i.e., an integer), such that (x, y) corresponds to a specific grid cell on the board that we wish to reference. The `value` corresponds to a string identifying a particular type of chess pieces (i.e., "King", "Queen", "Bishop", "Rook" or "Knight" - do note that the first letter is capitalised).

More specifically, the positions of the pieces at its goal state should be returned in this format:
$\{(x_1, y_1) : piece_1, (x_2, y_2) : piece_2, (x_3, y_3) : piece_3, ..., (x_j, y_j) : piece_j\}$

An example of the function printed and its output is shown below:
`print(run_CSP())`

Sample Output:
$\{('a', 0) : Queen, ('b', 3) : King, ('c', 7) : Rook, ..., ('d', 5) : Knight\}$

Grading Rubrics (Total: 10 marks)

Requirements (Marks Allocated)	Total Marks
• Correct implementation of Local Search Algorithm evaluated by passing all public test cases and hidden test cases (4m). • Correct implementation of Backtracking Algorithm evaluated by passing all public test cases and hidden test cases (6m).	10

Grading Details

Your code will be graded for technical correctness. Please do not change the name of the function `run_XXX()`.

There are 3 configuration files released to you via LumiNUS (together with the skeleton implementation files). These files serve as public test cases. There are additional private test cases that we will also run using your code (on CodePost). This is to prevent students from applying brute-force and simply returning the results without implementing the algorithms. Hence, to get the full credit for each task, you are required to pass all the public test cases and private test cases.

For each test case, the correctness of your code will be tested (e.g., returning the valid goal state in the specific format) and the time taken for the execution of your code will be measured as well. The time taken for your search algorithms will be compared to our solution's benchmark. If the time taken for your code is above the time limit, it will be considered a failure of the test case. As such, you are encouraged to use efficient data structures when implementing your code (e.g. using a set/dictionary instead of a list). To account for the execution of code on different systems, we have provided additional buffers for the time limit.

In summary, to pass a test case, two conditions have to be fulfilled for Local/CSP. They are:

1. Valid goal state comprising of the chess pieces in valid positions.
2. Time taken to run your search is within the time limit.

CodePost (Platform for Running Autograder)

We will be using CodePost as our platform to run the autograder and test your code. You should have already joined the Course's group on CodePost from Project 1. If you need any assistance for CodePost signups, please feel free to reach out and email any of the TAs.

Subsequent access: https://codepost.io/student/CS3243%20AY2223%20S1/Fall%202022/

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. **Note that you should not print any output in your Python files but only return the required values as this may interfere with the Autograder, e.g., dictionary mapping of positions to pieces.**
3. After the submission has been processed, refresh the page and select "View feedback".
4. Ideally, if your implementation is correct and is within the runtime threshold, the output will look like this:

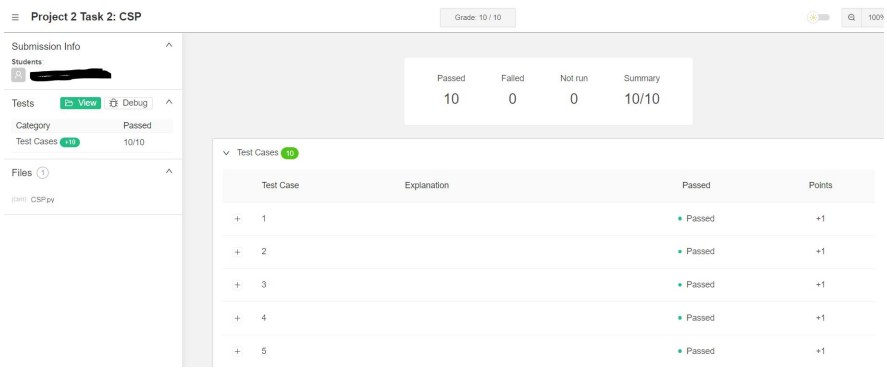

Figure 6: Codepost output

5. As mentioned in the previous section, if your solution is incorrect or takes too long, the autograder on CodePost will deem it as incorrect.

6. You will receive full credit for the search algorithm when you pass all testcases in CodePost. However, **random checks** will be carried out to inspect your code to ensure that the implementation is correct. **Moreover, we will check for any plagiarism and students found plagiarising will be dealt with seriously.**

You may submit your files as many times as you like. Note that this platform is run by an external organisation – we are not responsible for any downtime.

Other details

Allowed Libraries:

To aid in your implementation, you are allowed to use these Python libraries:

- CLI arguments: sys
- Datastructures: queue, collections, heapq, array, copy, enum, string
- Math: numbers, math, decimal, fractions, random
- Functional: itertools, functools, operators
- Types: types, typing

Testcases Debugging:

When submitting on Codepost, you should remove any print() functions and rename any functions/variables/string that contain the "print" word in your code to prevent the Autograder from terminating. Commenting out the print() functions may not guarantee the Autograder will pass check.

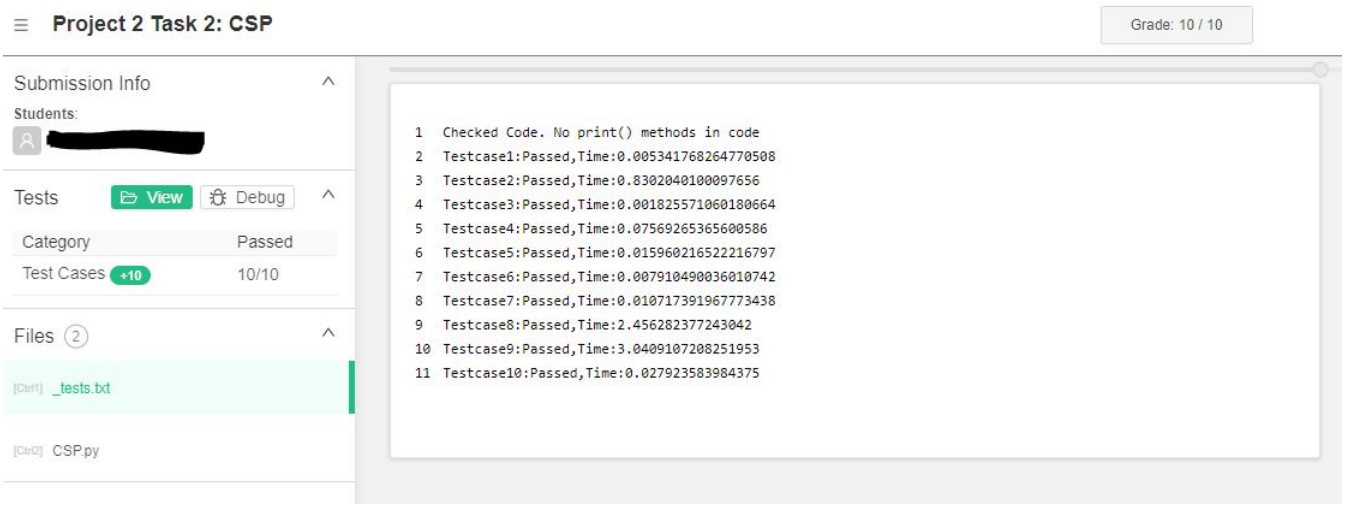

Figure 7: Codepost Debug Output

Submission Details via LumiNUS

- For this project, you will need to submit 1 zip file containing 2 Python files: `Local.py`, `CSP.py`.
- Place the 2 files in a folder, name the folder as `studentNo` and zip it as `studentNo.zip`. An example will be: `A0123456X.zip`.
- The format of submission is the same as Project 1
- In summary, your submission file should be a zip file and when unzipped, it will contain the 2 files in a folder: **A0123456Z.zip** $\rightarrow$ **unzip** $\rightarrow$ **A0123456Z folder** $\rightarrow$ **Local.py, CSP.py files**. There should not be any subfolders.
- Do not modify the file names.
- Make only one submission on LumiNUS.
- 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 testcases as your code cannot be tested.
- Submission folder: **LumiNUS** $>$ **CS3243** $>$ **Projects** $>$ **Project 2 Submission Folder**