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

Project 1: Path Planning Search

Issued: 24 August 2022 Due: 18 September 2022

Overview

In this project, you will formulate a search problem and through that, implement search algorithms and heuristics to find valid paths to goals in a game. Specifically, you are tasked to implement the following algorithms to find a solution path in a maze:

1. Implement the **Breadth-First Search** (BFS).
2. Implement the **Depth-First Search** (DFS).
3. Implement **Uniform-Cost Search** (UCS).
4. Implement **A* Search** (A*).

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: **18 September 2022 (Sunday) 11:59pm**
- Submission folder: **LumiNUS** > **CS3243** > **Projects** > **Project 1 Submission Folder**
- Submission format: One standard (non-encrypted) **zip file**[1] containing only the necessary project files.

  In particular, it should contain 1 folder with 4 `.py` files:
  (`BFS.py, DFS.py, UCS.py, AStar.py`)

  More info is given in the Submission Details section.

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

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

- Task 1 (BFS): Implement your BFS algorithm in `BFS.py`.
- Task 2 (DFS): Implement your DFS algorithm in `DFS.py`.
- Task 3 (UCS): Implement your UCS algorithm in `UCS.py`.
- Task 4 (A*): Implement your A* algorithm in `AStar.py`.

Note that you are tasked to implement your own search space and state structure for the game. You can reuse the search space/state structure in all 4 search algorithms.

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 48 hours or more 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: Chess

Chess is an abstract strategy game and involves no hidden information. It is played on a square chessboard with 64 squares arranged in an 8x8 grid among two opposing players. At the start, each player (one controlling the white pieces, the other controlling the black pieces) controls sixteen pieces: one king, one queen, two rooks, two knights, two bishops, and eight pawns. The objective of the game is to checkmate the opponent's king, whereby the king is under immediate attack (in "check") and there is no way for it to escape. There are also several ways a game could end in a draw.

Figure 1: A layout of a chess board.

In this project, we will **not** be designing a game of chess. Instead, we will create a one-player game variant that utilitises a re-sizable chess board and a subset of the chess pieces, whilst introducing **new objectives, obstacles and new pieces to the game**. Also, note that we will not be using the pawn[1] piece in this project.

In the section below, the movement rules of each piece in the game is introduced and they should be strictly followed in the implementation of the game. **Note that we are only concerned with the movement of each piece and not any other complex rules (e.g., castling)**

Game Pieces

Each piece has its unique way of moving. In the diagrams below, the dots mark the squares to which the piece can move if there are no intervening piece(s) obstructing its path (except the knight, which leaps over any intervening pieces).

Classic Chess Pieces:

- **King**: The king can move one square in any direction.
- **Rook**: A rook can move any number of squares along a rank or file, but cannot leap over other pieces.
- **Bishop**: A bishop can move any number of squares diagonally, but cannot leap over other pieces.

[1] https://en.wikipedia.org/wiki/Pawn_(chess)

- **Queen**: A queen combines the power of a rook and bishop and can move any number of squares along a rank, file, or diagonal, but cannot leap over other pieces.
- **Knight**: A knight moves to any of the closest squares that are not on the same rank, file, or diagonal (thus the move forms an ”L”-shape: two squares vertically and one square horizontally, or two squares horizontally and one square vertically). The knight is the only piece that can leap over other pieces.

Obstacle: An obstacle in the game has no moves and it takes up a square in the board. No other pieces can occupy the same position as the obstacle and they cannot leap over the obstacle (except the knight).

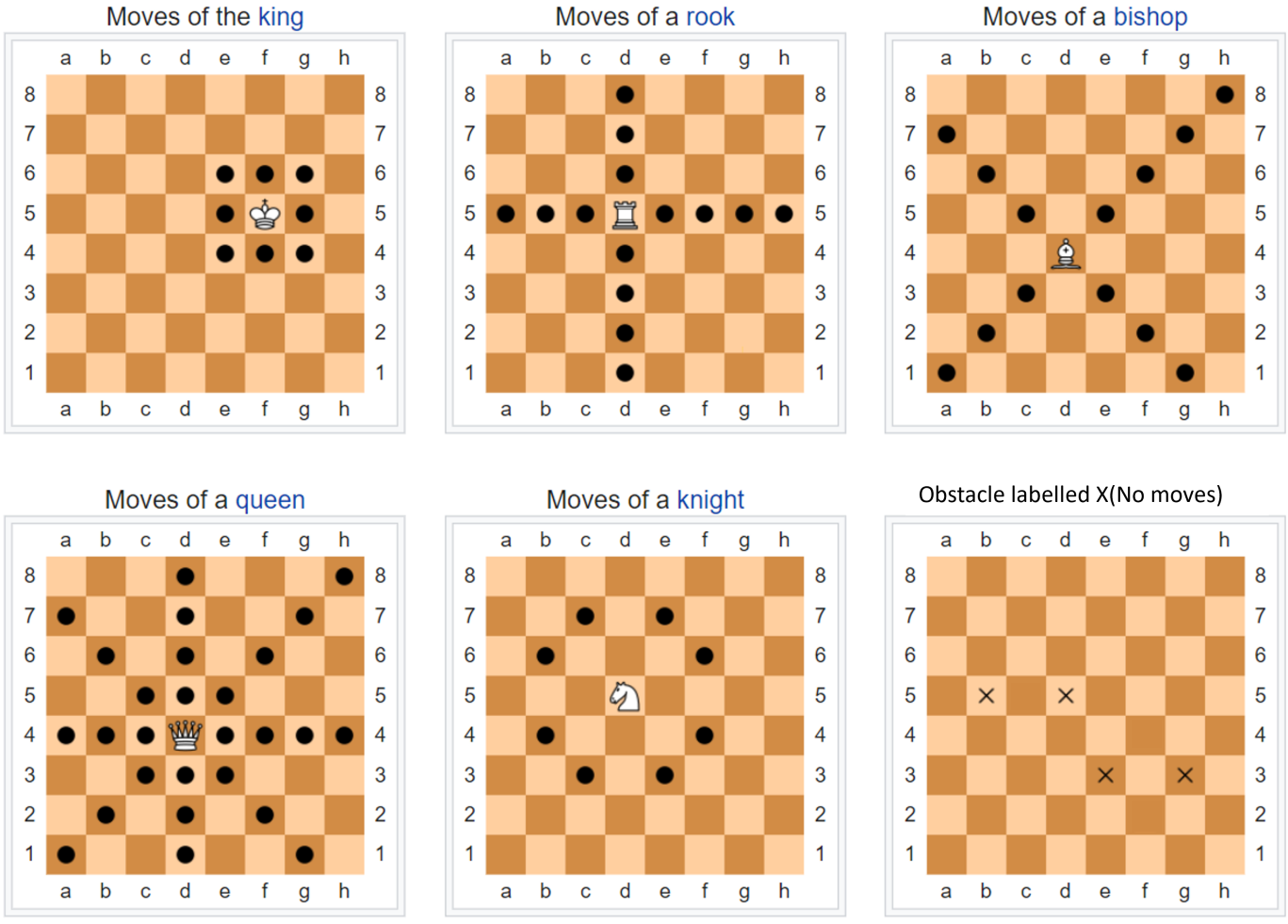

Figure 2: Movement of Chess Pieces

Fairy Chess Pieces:

- **Ferz**: The ferz is a fairy chess piece that may move one square diagonally.
- **Princess**: The princess is a fairy chess piece that can move like a bishop or a knight. It cannot jump over other pieces when moving as a bishop but may do so when moving as a knight.
- **Empress**: The empress is a fairy chess piece that can move like a rook or a knight. It cannot jump over other pieces when moving as a rook but may do so when moving as a knight.

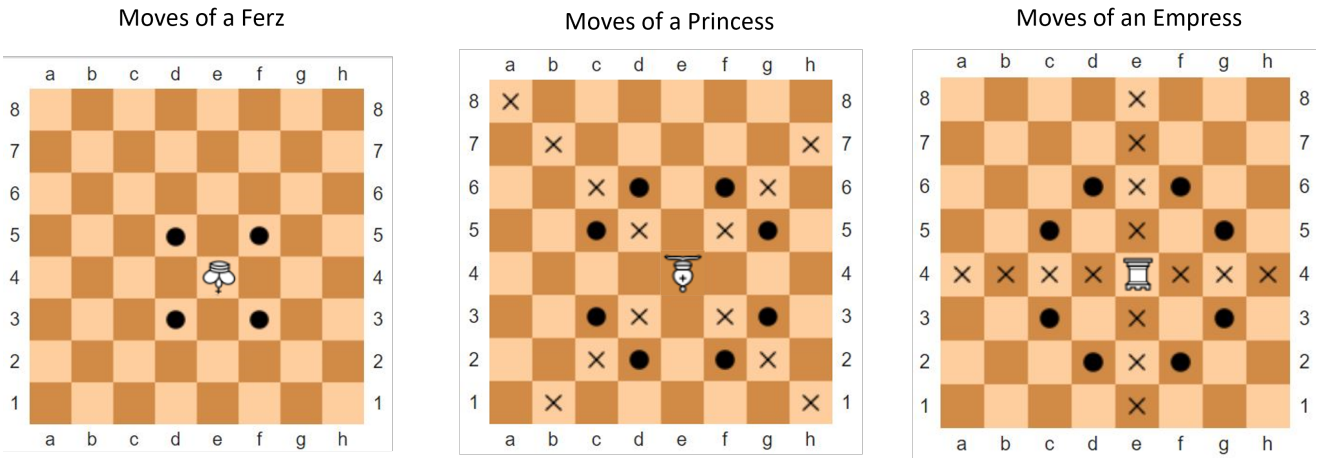

Figure 3: Movement of Fairy Chess Pieces

Game Board

A typical chess game is played on a square board with 64 squares arranged in an 8x8 grid. In this project, we relax the constraints on the board and allow the board size to be adjustable (e.g., we can create a 5x5 board, 7x6 board, 20x20 board).

The X-axis follows the ASCII character ordering staring from 'a' from the left, while the Y-axis follows the numerical ordering starting from 0 from the top. E.g., for an 10x10 grid, the X-axis is indexed as: 'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j' from the left to right, while the Y-axis is indexed as: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 from the top to bottom. These indices are to be used to represent the squares on the game board, e.g., the square on column 'a' and row 3 is represented as a3 in this text and $(a, 3)$ in the implementation. For this Project, we limit the maximum length of the X-axis to 26.

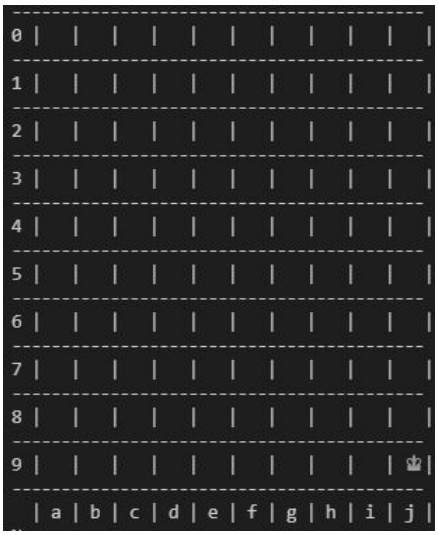

Figure 4: A 10x10 Board with a King at j9.

Action Costs

Additionally, we introduce different costs on each square of the board. By default, a move to any square costs 1 unit. However, in this project, we may define different costs on different squares and hence moving to a square on the board may cost more than 1 unit.

Specifically, to define costs, each square on the board will be assigned a cost, c. For any move made to that square, that move would thus incur cost c.

Implementation of Game Board and Game Pieces

Before implementing the search algorithms in the project, you will be required to design and implement your own game board and game pieces, including the movement of the pieces, the game state and the search space. This section encapsulates the problem formulation of the project.

For each test case, you will be given an input text file (e.g. `1.txt`) containing the board configuration (e.g., number of rows/columns, type of game pieces, starting position of game pieces, ending position of game pieces). To aid you in implementing the state space, a parser function has been defined for you to read the text file and process the information required.

Note that this step is important as your search algorithms in this project AND subsequent projects will make use of the design of your game board and game pieces.

Getting Started

You are given 4 python files (`BFS.py, DFS.py, UCS.py, AStar.py`) with recommended empty functions/classes to be implemented. Specifically, the following are given to you:

1. `run_BFS/DFS/UCS/AStar()`: **DO NOT REMOVE THIS FUNCTION.** When called, this function must return a **List of Moves**. The returned output will be evaluated by the autograder to check correctness. For `run_UCS()` and `run_AStar()`, the function will also return the **Path Cost** on top of the list of moves.

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 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 all 4 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, When each of the 4 python files is executed, the method `run_XXX()` will be called. The output should be a list of valid moves for the game (as well as the path cost for **UCS** and **AStar**).

Finding a Valid Path using Search Algorithms

The objectives of this part of the project are:

1. To gain exposure in formulating a search problem.

2. To gain exposure in the implementation of search algorithms taught in class.

3. To learn to recognise the differences and utility of each search algorithm.

4. To learn to recognise the effects that the different search methods (uninformed and informed search) have on the performance and efficiency.

Project 1: King's Maze

Oh no! The King of CS3243 is captured and trapped in a dungeon by his enemy kingdom, CS9999. The dungeon is filled with many obstacles and traps, with guards overlooking the place to prevent the King from escaping. However, as night approaches, the guards fall asleep in their positions, leaving the King an opportunity to escape the dungeon.

Now it's time to write full-fledged generic search functions to help the King find his way out! Remember that a search node must contain not only a state but also the information necessary to reconstruct the path (plan) which gets to that state.

Important note: All of your search functions need to return **a list of actions** that will lead the agent (King) from the start to the goal. These actions all have to be **legal moves** (valid movement rules of each piece, no movement through walls/obstacles).

Hint: Each algorithm is very similar. Algorithms for BFS, DFS, UCS, and A* should differ only in the details of how the frontier is managed. So, concentrate on getting BFS right and the rest should be relatively straightforward.

Background:

We are given a King chess piece in a game board of varying sizes and each board contains varying number of **obstacles** as well as **enemy game pieces**. The positions of the obstacles and enemy chess pieces are given in the input file and they **remain static at all times** (remain in the same position throughout the gameplay). The enemy chess pieces can threaten its surrounding positions based on the type of chess piece it is. This implies that our King piece cannot venture to the threatened positions due to the enemy chess pieces. For example, in the diagram below, squares a2, b2, b3, b4 and a4 are the threatened positions due to the enemy King at a3. Hence, our King piece is **not allowed** to move to these threatened positions.

Furthermore, our King's aim is to escape the dungeon and not fight the guards. Hence, our King piece cannot capture (i.e., remove) other enemy pieces on the board and it cannot visit squares inhabited by other pieces (of course, including obstacles).

We are given the starting position of our King and the goal position(s) in the input file. To escape the dungeon, the King has to reach any of the goal position(s). The diagrams below show an

example of an initial state and goal state of a game. The input file will also include information about costs.

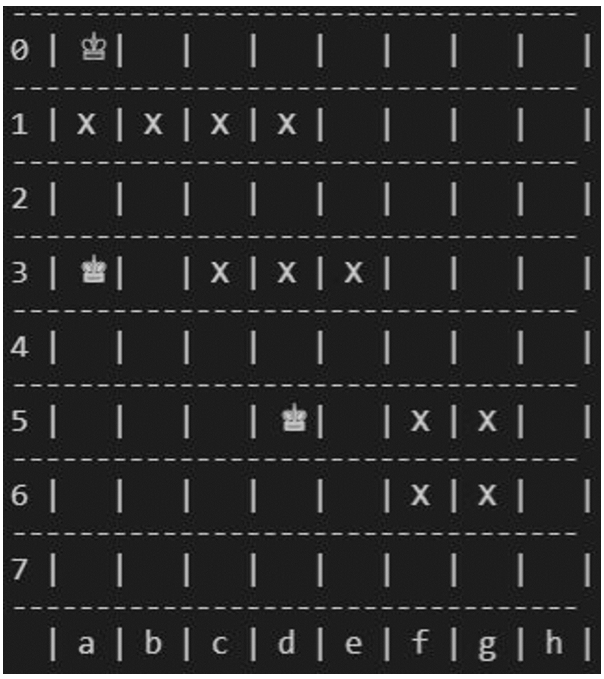

Figure 5: An initial state of the game with the King at a0 as its starting position, with two enemy Kings at a3 and d5 and obstacles labelled as X. In this game, the given goal position is h7.

Figure 6: The goal state of the game with our King at h7.

Task 1: Breadth First Search

Implement the **breadth-first search (BFS)** algorithm in `BFS.py`.

When the `run_BFS()` function is executed in your code, your code should return a list of valid moves in the following format:
$[move_1, move_2, move_3, ..., move_{last}]$

where $move_i$ is a **list containing two tuples**: `[Current Position Tuple, Next Position Tuple]` and in each position tuple, it will contain 2 elements `(x,y)` where `x` is the column index (i.e. a string) and `y` is the row index (i.e. an integer).

More specifically, the list of moves should be returned in this format:
$[[(x_1, y_1), (x_2, y_2)], [(x_2, y_2), (x_3, y_3)], ...[(x_i, y_i), (x_j, y_j)]]$

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

Output:
$[[('a', 0), ('b', 0)], [('b', 0), ('c', 0)], [('c', 0), ('d', 0)], [('d', 0), ('e', 1)], [('e', 1), ('d', 2)]]$

Task 2: Depth First Search

Implement the **depth-first search (DFS)** algorithm in `DFS.py`.

When the `run_DFS()` function is executed in your code, your code should return a list of valid moves in the following format:
$[move_1, move_2, move_3, ..., move_{last}]$

Details of $move_i$ are as explained in Task 1.

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

Output:
$[[('a', 0), ('b', 0)], [('b', 0), ('c', 0)], [('c', 0), ('d', 0)], [('d', 0), ('e', 1)], [('e', 1), ('d', 2)]]$

Task 3: Uniform Cost Search

While BFS finds the fewest-moves path to the goal, at times we may wish to find paths that are "best" in other sense. In Task 3 and 4, we consider a cost function and our objective is to search for a valid path that is of the least cost.

Implement the **uniform-cost search (UCS)** algorithm in `UCS.py`. Note that your implementation should only require a few adjustments, taking into consideration the action cost and path cost.

When the `run_UCS()` function is executed in your code, your code should return a list of valid moves and the path cost in the following format:
$[move_1, move_2, move_3, ..., move_{last}]$`, pathCost`

where $move_i$ is a **list containing two tuples**: `[Current Position tuple, Next Position tuple]` and `pathCost` is an **integer** representing the sum of the action cost of each move from the initial position to the goal position.

For each position tuple, it will contain 2 elements `(x,y)` where `x` is the column index (i.e. a string) and `y` is the row index (i.e. an integer).

More specifically, the list of moves and path cost should be returned in this format:
$[[(x_1, y_1), (x_2, y_2)], [(x_2, y_2), (x_3, y_3)], ...[(x_i, y_i), (x_j, y_j)]]$, `pathCost`

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

Output:
$([[('a', 0), ('b', 0)], [('b', 0), ('c', 0)], [('c', 0), ('d', 0)], [('d', 0), ('e', 1)], [('e', 1), ('d', 2)]], 64)$

Task 4: A* Search

Implement **A* search** algorithm in `AStar.py`. Again, this should just be an extension of your previous search algorithms, with the addition of heuristics design and computation of h-cost, g-cost and f-cost.

When the `run_AStar()` function is executed in your code, your code should return a list of valid moves and the path cost in the following format:
$[move_1, move_2, move_3, ..., move_{last}]$, `pathCost`

Details of $move_i$, and `pathCost` are as explained in Task 3.

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

Output:
$([[('a', 0), ('b', 0)], [('b', 0), ('c', 0)], [('c', 0), ('d', 0)], [('d', 0), ('e', 1)], [('e', 1), ('d', 2)]], 64)$

Remember, heuristics are just functions that take search states and return numbers that estimate the cost to a nearest goal. More effective heuristics will return values closer to the actual costs to the goal. Do recall the definitions of admissibility and consistency from the lectures and know what is required to achieve optimality under the different conditions.

Non-Trivial Heuristics: A trivial heuristic is one that always returns zero (analogous to UCS) and this will not save us any time. Also, a heuristic that computes the true cost to goal will cause the autograder to timeout. You want a heuristic which reduces total compute time; although for this assignment the autograder will only check node counts (aside from enforcing a reasonable time limit).

Invalid Goal State / No Goal

If no goal exists in the puzzle, there is no solution. In such cases, your 4 search algorithm implementations should return an **empty list** for your list of valid moves (and **0** for your path cost for **UCS** and **AStar**).

Grading Rubrics (Total: 10 marks)

Requirements (Marks Allocated)	Total Marks
• Correct implementation of Breadth First Search Algorithm evaluated by passing all public test cases and hidden test cases (2m). • Correct implementation of Depth First Search Algorithm evaluated by passing all public test cases and hidden test cases (2m). • Correct implementation of Uniform Cost Search Algorithm evaluated by passing all public test cases and hidden test cases (3m). • Correct implementation of A* Search Algorithm evaluated by passing all public test cases and hidden test cases (3m).	10

Grading Details

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

There are 3 configuration files released to you via LumiNUS (together with the skeleton implementation files). These files serve as public testcases. There are additional private testcases that we will also run using your code (on CodePost). This is to prevent students from using brute-force

and simply returning the results without implementing the search 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 a valid list of moves) 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.

For UCS and A*, the path cost returned by your algorithms will be compared with ours to determine if your search path is the optimal path.

In summary, to pass a test case, two conditions have to be fulfilled for BFS/DFS and three conditions have to be fulfilled for UCS/A*. They are:

1. Valid list of moves from start position to goal position.
2. Time taken to run your search is within our benchmark.
3. For UCS/A*: Path cost given by your search is the optimal path cost.

Hence, even if your implementation is technically correct but if the time taken is not within the threshold or if the path cost is not the optimal path cost for UCS/A*, the autograder will fail the test case.

CodePost (Platform for Running Autograder)

We will be using CodePost as our platform to run the autograder and test your code. You should have received an email from CodePost in your NUS email stating that you are added to the course in CodePost. If you did not receive it, you can join using the link below. Do note that you can only join using your NUS email.

Logging in for the first time (Invitation link): https://codepost.io/signup/join?code=5VJAGQ7S5U

(Remember to check your spam/junk mail for the activation email. Contact the course staff if you do not receive it after 30 minutes.)

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

1. For each task (BFS/DFS/UCS/A*), click on "Upload assignment", and upload your Python file `BFS.py/DFS.py/UCS.py/Astar.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:

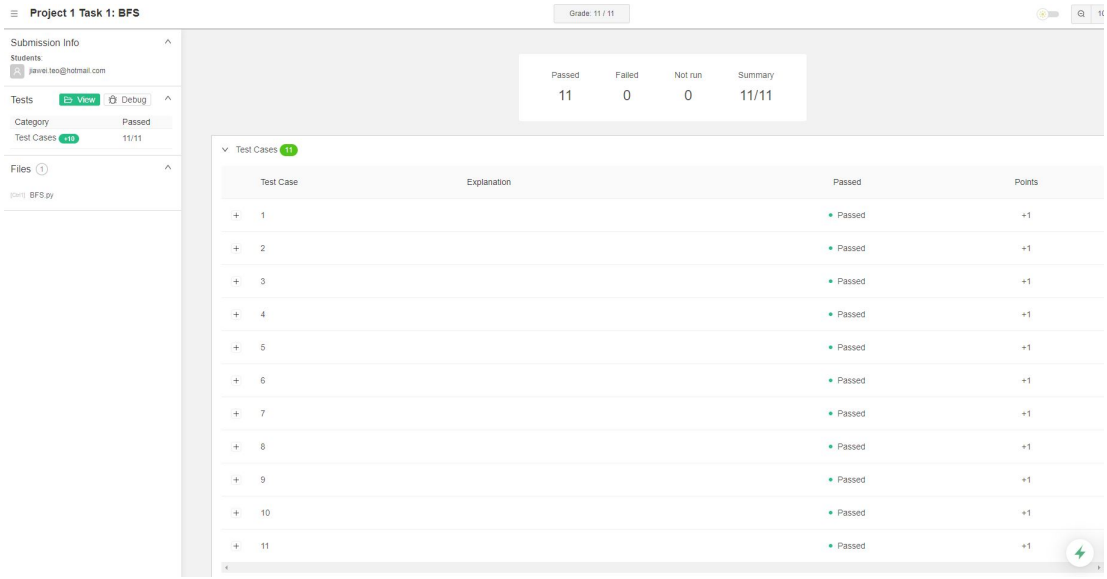

Figure 7: Codepost output

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.

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.

Other details

Allowed Libraries:

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

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

Submission Details via LumiNUS

- For this project, you will need to submit 1 zip file containing 4 Python files: `BFS.py`, `DFS.py`, `UCS.py` and `AStar.py`.
- Place all 4 files in a folder, name the folder as `studentNo` and zip it as `studentNo.zip`. An example will be: `A0123456X.zip`.
- In summary, your submission file should be a zip file and when unzipped, it will contain the 4 files in a folder: **A0123456Z.zip** $\rightarrow$ **unzip** $\rightarrow$ **A0123456Z folder** $\rightarrow$ **BFS.py, DFS.py, UCS.py, AStar.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 1 Submission Folder**