National University of Singapore
School of Computing
CS3243 Introduction to AI

Project 1.1: Introduction to Search

Issued: 26 August 2024 Due: 15 September 2024, 2359hrs

1 Overview

In this project, you will **implement 3 search algorithms** to find valid paths in a maze.

1. **Breadth-first search (BFS)** algorithm
2. **Depth-first search (DFS)** algorithm
3. **Uniform-cost search (UCS)** algorithm

This project is worth 3% of your course grade.

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: **15 September 2024**, **2359 hrs**
- 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 to 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 1.1: Escape the Maze!

2.1 Functionality

You will be given a static maze that has a variable initial size. Given a starting position, the **objective** is to find a path from the starting position to a designated goal square **without** passing through any obstacles.

As such, the following are some constraints on what you can or cannot do.

- You can only move in 4 directions: Up, Down, Left, or Right.
- You cannot move onto a cell blocked by an obstacle.
- You cannot move outside the bounds of the maze.

2.2 Board

In this project, the initial maze size is a **parameter**, with the maximum number of columns being 1101 and the maximum number of rows being 1101.

2.2.1 Coordinate System - Matrix Coordinates

The coordinate system used on a maze is the **matrix coordinate system** i.e. `(row, col)`. For example, in a 5×10 grid, there are 5 rows and 10 columns. The indices of the rows are 0, 1, 2, 3, 4 from **top to bottom**, and the index of the columns are 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 from **left to right**. These indices are used to represent the squares on the maze, e.g., the square on row 3 and column 0 is represented as $(3, 0)$.

All inputs and outputs relating to positions will be given in matrix coordinates! Note that the matrix coordinate system *is not* the Cartesian coordinate system.

2.3 Input Constraints

In the following section, a `Position` type is a `List[int]` of length exactly 2.

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

- `cols`: The number of columns the maze has. Type is `int`.
- `rows`: The number of rows the maze has. Type is `int`.
- `obstacles`: The list of positions on the maze occupied by obstacles. Type is `List[Position]`.
- `start`: The starting position. Type is `Position`
- `goals`: The goal positions. Type is `List[Position]`

The action cost to get to any position is 1.

2.4 Requirements

You are to implement a function called `search` that **takes in** a dictionary, which is described in Section 2.3 and **returns** a valid path to a given goal. Specifically, you must return a `List[Tuple[int, int]]` representing the path that your agent will take.

For example, if you start at $(0, 0)$, then move to $(0, 1)$, then move to $(1, 1)$, you should output `[(0, 0), (0, 1), (1, 1)]`.

The following are some **general requirements** on your output.

1. All positions along the produced path must be free of obstacles
2. All positions along the produced path must be reachable from the previous position by a single movement in the 4 directions: UP, DOWN, LEFT, RIGHT, subject to maze boundaries and obstacles.
3. Paths of non-zero length must terminate at a given goal position
4. Paths of non-zero length must begin from the given start position
5. If there are no legal paths, return an empty list i.e. a path of zero length.

For BFS and UCS, there is an additional requirement of having to output a path with the lowest cost.

You are **required** to implement the algorithms asked for. For example, when submitting for Uniform Cost Search (UCS), you may not pass in Breadth-First Search (BFS) or any other search algorithms in its place. This requirement will be **enforced** on the final submissions.

3 Grading

3.1 Grading Rubrics (Total: 3 marks)

Requirements (Marks Allocated)	Total Marks
• Correct implementation of Breadth First Search Algorithm (0.7m). • Efficient implementation of Breadth First Search Algorithm (0.3m). • Correct implementation of Depth First Search Algorithm (0.7m). • Efficient implementation of Depth First Search Algorithm (0.3m). • Correct implementation of Uniform Cost Search Algorithm (0.7m). • Efficient implementation of Uniform Cost Search Algorithm (0.3m).	3

3.2 Grading Details

All test cases in the same category have the same weightage. The final mark is obtained by using the following formula:

$$\text{final mark} = \sum_{c} \frac{\text{\# of test cases with AC in } c}{\text{\# of test cases in } c} \times \text{weight of } c \tag{1}$$

For example, suppose a student gets 124 out of 138 correctness test cases correct and 12 out of 18 efficiency test cases correct for DFS. Then, their mark for DFS is:

$$\text{final mark} = \frac{124}{138} \times 0.7 + \frac{12}{18} \times 0.3 = 0.795 \tag{2}$$

For the number of test cases in each category for each algorithm, refer to Section 5.2.

4 Submission

4.1 Submission Details 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

For other libraries, **please seek permission before use!**

5.2 Test Case Information

5.2.1 DFS

1. Correctness: 78 public test cases and 60 private test cases.
2. Efficiency: 9 public test cases and 9 private test cases.

5.2.2 BFS

1. Correctness: 78 public test cases and 60 private test cases.
2. Efficiency: 9 public test cases and 9 private test cases.

5.2.3 UCS

1. Correctness: 78 public test cases and 60 private test cases.
2. Efficiency: 8 public test cases and 9 private test cases.