National University of Singapore
School of Computing
CS3243 Introduction to AI

Project 1.2: Introduction to Informed Search

Issued: 25 August 2025 Due: 21 September 2025, 2359hrs

1. Overview

In this project, you will **implement 1 search algorithm** to find a valid sequence of actions to take in a 2-dimensional grid maze.

1. **A* search** (**A***).

This project is worth 7% 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 or later**
- Deadline: **21 September 2025, 2359hrs**
- 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 (i.e., that is taken from another source) should not be used directly. You should implement 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 more than 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 Dungeon!

2.1 Functionality

Your agent is looking for Runes in a dungeon (i.e., just like the maze in **Project 1.1**) filled with many enemy creeps. Each move the agent makes, and each creep it encounters will cause it to lose movement points (MP). Your agent ***must find (exactly) one rune while minimising the loss of MP***.

At each timestep, the agent can do exactly one of the following.

- Take an **action**.
- Use a **skill**.

An **action** is a movement (UP, DOWN, LEFT, RIGHT) in a 2-dimensional dungeon (recall the same actions and environment from **Project 1.1**). The exact rules and costs of an action are described in **Section 2.2**. A **skill** *modifies the next action that is taken or the state of the dungeon*. The rules for skills are described in **Section 2.3**.

Your task is as follows.

Implement the A* search algorithm to find a sequence of actions that would lead the agent from the initial position in the dungeon to a position containing a rune using a minimal amount of MP.

Note: all coordinates below will be given as **matrix coordinates**. This means that a coordinate `(x, y)` refers to the position at row `x` and column `y` in the dungeon. Refer to **Project 1.1** file for more details.

2.2 Actions

There are 4 actions that your agent can take: UP, DOWN, LEFT, and RIGHT. Such an action will bring the agent one cell in that direction - e.g., if the agent starts from `(1, 0)`, then moving UP will bring your agent to `(0, 0)`. You *cannot move onto a cell blocked by an obstacle or outside the bounds of the dungeon*. Each action costs 4 MP.

Apart from the cost due to taking an action, there are also *costs due to encountering a creep*. Each creep that is encountered will cost 1 MP.

Example: suppose you start at position `(1, 0)` with 10 creeps, and you use UP to move to position `(0, 0)` with 20 creeps. Then, the total MP lost from taking an UP action is 4 + 20 = 24 MP.

2.3 Skills

You have two skills in your arsenal: a standard skill, FLASH, and an ultimate skill, NUKE.

2.3.1 Flash

When cast, FLASH changes the next move's rule from "move by 1 cell" to "move until an obstacle or dungeon boundary is reached". Each cell traversed in this manner incurs a cost of 2 MP (note that the usual cost from move from one cell to the next is 4 MP). An invocation of FLASH costs 10 MP and **does not change the position of the agent**. This skill can only be cast at most `i` times, where `i` is provided as part of the input. It affects only the next action taken, and no other actions can be taken at the same time as its use.

Examples:

1. Suppose you are at cell `(0, 0)` in a dungeon with 1 row and 4 columns.
   - Without FLASH, you need 3 actions to reach `(0, 3)`, namely RIGHT → RIGHT → RIGHT. Total MP cost is (4 + B) + (4 + C) + (4 + D), where B, C, D corresponds to the number of creeps at `(0, 1)`, `(0, 2)`, and `(0, 3)` respectively.

- With FLASH, you only need 1 FLASH invocation and 1 action, namely FLASH → RIGHT. Total MP cost is 10 + (2 + 2 + 2) + B + C + D, where B, C, D corresponds to the number of creeps at `(0, 1)`, `(0, 2)`, and `(0, 3)` respectively.
- Note the following differences in cost computations when FLASH is used. – There is an additional cost of 10 MP. This corresponds to the invocation cost. – All the 4s become 2s. This is because FLASH modifies action cost from 4 to 2.

2. Suppose you are at cell `(0, 0)` in a dungeon of width 8. Suppose cell `(0, 4)` is an obstacle. Then, the following holds.
   - FLASH → RIGHT will bring you to `(0, 3)`, since FLASH does not allow you to pass through obstacles.
   - Note that from `(0, 0)`, FLASH → RIGHT cannot bring you to `(0, 2)`, since this violates the rule that FLASH causes movements to "move until an obstacle ... is reached".
   - FLASH → FLASH → RIGHT is strictly worse than FLASH → RIGHT, since the first FLASH incurs an extra 10 MP lost, decreases the remaining number of FLASH that can be cast left, and does not modify any action.

2.3.2 NUKE

When cast, NUKE allows you to remove creeps **within** a radius of 10 Manhattan Distance steps from where the agent casts the NUKE skill. An invocation of NUKE costs 50 MP and **does not change the position of the agent**. This skill can only be cast at most `j` times, where `j` is provided as part of the input. The effect of a NUKE is **permanent**.

Example: In a dungeon with 100 rows and 100 columns, suppose the agent is at position `(50, 50)`. If the agent then uses the NUKE skill, then all non-obstacle cells within a radius of (Manhattan) distance 10 will have its creeps removed - e.g., if cell `(59, 51)` has 81 creeps originally, after NUKE this cell would have 0 creeps. Cells outside of this radius, e.g., `(60, 51)`, are unaffected.

2.4 Input Constraints

In the following, 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 in the dungeon. Type is `int`.
- `"rows"`: The number of rows in the dungeon. Type is `int`.
- `"obstacles"`: The list of obstacle positions in the dungeon. Type is `list[Position]`.
- `"creeps"`: A list of positions in the dungeon containing creeps. Type is `list[list[int]]`, where the inner list corresponds to `[x, y, num_creeps]` – e.g., `[3, 0, 10]` corresponds to the cell at `(3, 0)` containing 10 creeps. Do note that all positions that are not obstacles, and that are not listed in `creeps`, contain 0 creeps.
- `"start"`: The starting position. Type is `Position`.
- `"goals"`: The position(s) of runes within the dungeon. Type is `list[Position]`.
- `"num_flash_left"`: The maximum number of times FLASH can be invoked. Type is `int`.
- `"num_nuke_left"`: The maximum number of times NUKE can be invoked. Type is `int`.

2.5 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. More 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)]`.

You are to implement a function called `search` that **takes in** a dictionary, which is described in **Section 2.4**, and **returns *a valid sequence of actions*** (i.e., sequence actions (**Section 2.2**) and skills (**Section 2.3**)) that will bring the agent to position in the dungeon that contains a rune. More specifically, you should return a `list[int]` representing the sequence of actions taken.

Use the following encoding to represent your actions as integers:

```
class Action(Enum):
    UP = 0
    DOWN = 1
    LEFT = 2
    RIGHT = 3
    FLASH = 4
    NUKE = 5
```

For example, if the agent starts at position `(0, 0)`, then moves RIGHT to `(0, 1)`, and finally moves DOWN to `(1, 1)`, this sequence of actions, i.e., the output, should be encoded as `[3, 1]`.

The following are some **requirements** on your output:

1. No action must leave the agent outside the bounds of the dungeon or in a position containing an obstacle.
2. If there are no legal actions that the agent may take, then return an empty list.
3. The returned sequence of actions must **cause the least MP loss**.
4. The heuristic function **may not be** the zero heuristic.

Recall that for A*, there is a requirement of having to output a path with the lowest cost.

You are **required** to implement the algorithm(s) specified in **Section 1**. For example, when submitting for A*, you may not submit Breadth First Search (BFS) or any other search algorithms in its place. This requirement will be **enforced** on your final submissions on Coursemology.

3. Grading

3.1 Grading Rubrics (Total: 7 marks)

Requirements (Marks Allocated)	Total Marks
• Correct implementation of A* Search Algorithm (5m). • Efficient implementation of A* Search Algorithm (2m).	7

3.2 Grading Details

Each test case in the same category has 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 42 out of 45 correctness test cases correct and 4 out of 6 efficiency test cases correct for A*. Then, their mark for A* is:

$$\text{final mark} = \frac{42}{45} \times 5 + \frac{4}{6} \times 2 = 6 \tag{2}$$

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

4. Submission

4.1 Details of Submission 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 A*

- Correctness: 30 public test cases and 15 private test cases.
- Efficiency: 3 public test cases and 3 private test cases.