Assignment 1

Part 1 Pathfinding

Pathfinding is the problem of finding a path between two points on a plane. It is a fundamental task in robotics and AI. Perhaps the most obvious usage of pathfinding is in computer games, when an object is instructed to move from its current position to a goal position, while avoiding obstacles (e.g., walls, enemy fire) along the way.

Pathfinding in commercial games is frequently accomplished using search algorithms[1]. We consider a simplified version in this assignment. The following shows a 2D map drawn using ASCII characters:

```
1 1 1 1 1 1 4 7 8 X
1 1 1 1 1 1 1 5 8 8
1 1 1 1 1 1 1 4 6 7
1 1 1 1 1 X 1 1 3 6
1 1 1 1 1 X 1 1 1 1
1 1 1 1 1 1 1 1 1 1
6 1 1 1 1 X 1 1 1 1
7 7 1 X X X 1 1 1 1
8 8 1 1 1 1 1 1 1 1
X 8 7 1 1 1 1 1 1 1
```

Given a start position and an end position on the map, our aim is to find a path from the start position to the end position. The character 'X' denotes an obstacle that cannot be traversed by a path, while the digits represent the elevation at the respective positions.

Any position is indicated by the coordinates (i, j), where i is the row number (ordered top to bottom) and j is the column number (ordered left to right). For example, the top left position is (1, 1), the bottom right is (10, 10), while the position with elevation 3 is (4, 9). Given start position (1, 1) and end position (10, 10), a possible path is:

```
* * * 1 1 1 4 7 8 X
1 1 * 1 1 1 1 5 8 8
1 1 * * * * * * * 7
1 1 1 1 1 X 1 1 * 6
1 1 1 1 1 X 1 * * 1
1 1 1 1 1 1 1 * 1 1
6 1 1 1 1 X 1 * * *
7 7 1 X X X 1 1 1 *
8 8 1 1 1 1 1 1 1 *
X 8 7 1 1 1 1 1 1 *
```

Note that we use **4-connectedness** for paths, which means any two points on the path are only connected if they are vertically or horizontally (but <u>not</u> diagonally) adjacent.

Problem Formulation

Following the lecture notes, we formulate the problem as follows:

- **States:** Any obstacle-free position (i, j) on the map.
- **Initial States:** A position (i_0, j_0) given by the user.
- **Actions:** Since we consider 4-connectedness, only four actions are available: Up, down, left and right (**your program must expand each node in this order**). Available actions for positions adjacent to the map boundary or obstacles are reduced accordingly.
- **Transition Model:** Moving left moves the current position to the left, etc.
- **Goal Test:** Check if the current state is the end position (i_*, j_*) given by the user.
- **Path Cost:** Given a map M and a Path $P\{(i_0, j_0), (i_1, j_1), ... (i_N, j_N)\}$, the cost of the path is calculated as:

$$g\left(P\right) = \sum_{k=1}^{N} c\left(i_{k-1}, j_{k-1}, i_k, j_k, M\right),$$

where

$$c\left(a, b, c, d, M\right) = \begin{cases} 1 + M\left(c, d\right) - M\left(a, b\right) & if M(c,d) - M(a,b) > 0 \\ 1 & otherwise \end{cases}$$

and M(a, b) is the elevation at the position (a, b). In words, the cost of a path is the sum of the costs between adjacent points in the path, and the cost between adjacent points is 1 plus the difference between the elevation of the two points if we climb "uphill" or simply 1 if we stay "level" or slide "downhill".

This means shorter paths which avoid climbing cost less. As an example, the cost in the path in the previous page is 25. What is the optimal (i.e., cheapest) path?

Your Tasks

Solve this pathfinding task using three different methods:

- Breadth First Search (BFS)
- Uniform Cost Search (UCS)
- A* Search (A*)

You should base your program on the pseudocode GRAPH-SEARCH see below, and carefully think about the appropriate data structures to use.

```
function GRAPH-SEARCH (problem, fringe) returns a solution, or fail
ure
    closed <- an empty set
    fringe <- INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
    loop do
        if fringe is empty then return failure
        node <- REMOVE-FRONT(fringe)
```

```
        if GOAL-TEST(problem, STATE[node]) then return node
        if STATE[node] is not in closed then
            add STATE[node] to closed
            fringe <- INSERTALL(EXPAND(node, problem), fringe)
    end
```

A* search requires a **heuristic**. In this assignment, you must implement two such **heuristics**.

- The Euclidean distance between current position and end position.
- The Manhattan distance between the current position and end position.

For the map above with start position (1, 1) and end position (10, 10), your program should help you answer these questions:

1. Are the paths returned by the three methods different?
2. What about the optimality of the returned paths?
3. Which method is the most computationally and memory efficient?
4. Do the two heuristics for A* Search provide different solutions?
5. Does checking for repeated states matter in this problem?

Deliverables

Write your pathfinding program in Python 3.6.9 in a file called **pathfinder.py**. Your program must be able to run as follows:

>>> python pathfinder.py [map] [algorithm] [heuristic]

The inputs to the program are as follow:

- [map] specifies the path to the map, which is a text file formatted according to this example:

  ```
  10 10
  1 1
  10 10
  1 1 1 1 1 1 4 7 8 X
  1 1 1 1 1 1 1 5 8 8
  1 1 1 1 1 1 1 4 6 7
  1 1 1 1 1 X 1 1 3 6
  1 1 1 1 1 X 1 1 1 1
  1 1 1 1 1 1 1 1 1 1
  6 1 1 1 1 X 1 1 1 1
  7 7 1 X X X 1 1 1 1
  8 8 1 1 1 1 1 1 1 1
  X 8 7 1 1 1 1 1 1 1
  ```

  The first line indicates the size of the map (rows, columns), while the second and third line represent the start and end positions respectively. The map then follows where all elevation values are integers from 0 to 9 inclusive.
- [algorithm] specifies the search algorithm to use, with the possible values of **bfs**, **ucs**, and **astar**.

- [heuristic] specifies the heuristic to use for A* search, with the possible values of **euclidean** and **manhattan**. This input is ignored for BFS and UCS.

Your program must then **print to standard output** the path returned by the search algorithm in the following format:

```
* * * 1 1 1 4 7 8 X
1 1 * 1 1 1 1 5 8 8
1 1 * * * * * * * 7
1 1 1 1 1 X 1 1 * 6
1 1 1 1 1 X 1 * * 1
1 1 1 1 1 1 1 * 1 1
6 1 1 1 1 X 1 * * *
7 7 1 X X X 1 1 1 *
8 8 1 1 1 1 1 1 1 *
X 8 7 1 1 1 1 1 1 *
```

where the path is indicated by asterisks *, superimposed on the original map beginning from the start position and leading to the end position. **Do not include extraneous spaces or other characters in the output**.

If the given map or problem does not have a **feasible path**, your program must print:

null

Again, do not include extraneous spaces or other characters in the output.

Python Libraries

You are allowed to use Python standard libraries to write your pathfinding program (see **https://docs.python.org/3/library** 🔗 **(https://docs.python.org/3/library)**) for the components that make up the Python v3.6.9 standard libraries). In addition to the standard libraries, you are allowed to use NumPy. Note that the marking program will not be able to run your program to completion if other third party libraries are used.

Submission

You must submit your program files on Gradescope. Instructions on accessing Gradescope and submitting assignments are provided at **https://helpgradescope.com/article/5d3ifaeqi4-student-canvas** 🔗 **(https://help.gradescope.com/article/ccbpppziu9-student-submit-work)** . Please use the course code **6ZWNXV** to enrol in the course.

WARNING

There are two submission locations, one for **undergraduates** and one for **graduates**.

For **undergraduates**, please submit your pathfinding program (pathfinder.py) to Assignment 1 - UG

Assessment

The code will be compiled and run on several test problems. If it passes all tests, you will receive 15% (undergraduates), or 12% (postgraduates) of the overall course mark. There are <u>no</u> manual marks for style or commenting.

Using other Source Code

You may not use other source code for this assignment. You should personally and carefully implement the search algorithms to fully understand the concept.

Due Date

This assessment's Due Date is specified on MyUni.

Part 2: Pathfinding by direct optimisation (Post-Graduates Only)

For postgraduate students, completing this section successfully will give you the remaining 3% of the marks.

Here we shall attempt to directly optimise the path instead of step-by-step searching. We consider the simulated annealing algorithm shown below. For more background on simulated annealing, see Section 4.1 of Russell and Norvig (3rd ed.)

```
Algorithm 1

 1: input Initial path P⁰, initial temperature T_ini, final temperatu
 re T_fin,
                          cooling rate α, segment length d.
 2: output Optimised path P
 3: Intitialise T ← T_ini, P ← P⁰
 4: while T > T_fin do
 5:    Pʰ ← rand-local-adjust(P, d)
 6:    Δg ← g(P) - g(Pʰ)
 7:    if Δg > 0 then
 8:       P ← Pʰ
 9:    else
10:       With probability e^(Δg/t), P ← Pʰ
11:    end if  /* Record T and g(P) here for bookkeeping.*/
12:    T ← αT
13: end while
14: return P
```