Assignment 1: Finding the Shortest Path

1 Task Description

In this assignment, you have to guide the player agent in Minecraft through a maze to reach the goal block. For this exercise, we will assume that the agent can see the whole maze, i.e. it is completely observable. Given this information, the agent has to reach the goal in the minimum number of moves.

This assignment is based heavily on tutorial_7.py in the Malmo Python tutorials, so please follow the tutorials 1 through 5 to familiarize yourself with the API and the Malmo platform (tutorial 6 is not relevant yet).

1.1 Provided Source Code

We have provided two Python files. The most important one is assignment1.py which contains the complete code to setup the Malmo environment with different mazes, and have the agent run through it. The provided implementation is incomplete, and the agent needs your help to solve the maze most efficiently.

1.2 Setup and Running the Code

Assuming you have installed Malmo and started working your way through the tutorials, all you need to do to run this assignment is to copy the two files above to the Python_Examples folder, and after launching Minecraft, run python assignment1.py (Python 3 or higher). If everything run successfully, the agent should do nothing while the timer counts down for each of the ten missions. The output in the terminal should look like the following:

```
Size of maze:  6
Waiting for the mission 1 to start
Mission 1 running.
Output (start,end) 1 :  (None, None)
Output (path length) 1 :  0
Output (actions) 1 :  []
Error:  out of actions, but mission has not ended!
Error:  out of actions, but mission has not ended!

Mission 1 ended
Size of maze:  6
Waiting for the mission 2 to start
Mission 2 running.
Output (start,end) 2 :  (None, None)
Output (path length) 2 :  0
Output (actions) 2 :  []
Error:  out of actions, but mission has not ended!
Error:  out of actions, but mission has not ended!
```

1.3 Overview of the Code

As mentioned before, much of this assignment is based on tutorial_7.py , so please refer to the Malmo Tutorials in the Python_Examples directory. The source code creates a series of increasing difficult mazes made of diamond_block blocks as the floor, each with a start (emerald_block) and an end (redstone_block) block. The mission starts with the player at the start block, and ends when the player reaches the end block. Make sure you are at least somewhat familiar with the whole source code, however the main control code that is relevant for your implementation are in the following lines:
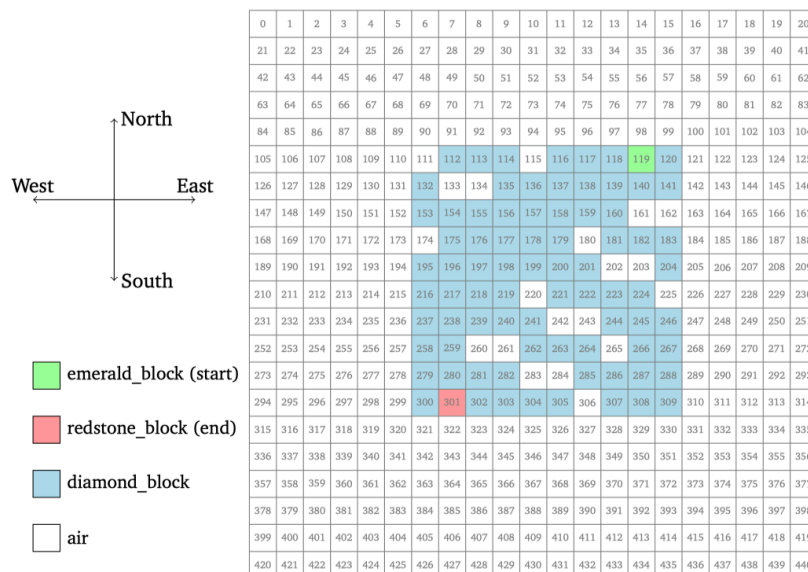

Figure 1: **Grid Layout:** Layout of the two-dimensional grid laid out as a one-dimensional array. The location of the maze, especially the start and end blocks, may be different in the assignment. The array contain string-values representing the four types of possible blocks.

```
223    grid = load_grid(world_state)
224    start, end = find_start_end(grid) # implement this
225    path = dijkstra_shortest_path(grid, start, end) # implement this
226    action_list = extract_action_list_from_path(path)
```

We first get the array of blocks that consists of the 2-dimensional maze from Malmo in load_grid . Then, we want to identify the start and the end blocks (each represented by an integer index into the grid array). Given the start and the end blocks, and the whole maze, we want to compute the shortest path of blocks from the start to the end, computed using Djikstra's algorithm. Finally, we convert these list of blocks into Malmo actions that are carried out by the agent. We provide the correct implementation of the first and the last functions, but leave the correct implementation of the second and the third functions to you (more on this later).

1.4 Grid Layout

It is important for you to understand how the grid is laid out. We have requested the 21 × 21 grid of blocks from the world that lie at the floor of the player, with the player at the center of it ((10,10), if zero-indexed). This two-dimensional is laid out as a single-dimensional array of strings (representing the type of the block) in a row-major way (where row is the east-west direction), i.e. the (i, j) coordinate is represented by the index $j \times 21 + i$, where j is the east-west coordinate, and i is the north-south coordinate. For the precise indexing, see the illustration in Figure 1.

Since the grid is represented by a single-dimensional array, you need to get the neighboring blocks of the current block, represented by an integer index i. Looking at extract_action_list_from_path should give you a hint; the blocks to the immediate left (west) and right (east) are of course represented by $i - 1$ and $i + 1$, while the blocks in the north and south require jumping one whole row, thus $i - 21$ and $i + 21$, respectively. The function extract_action_list_from_path uses this to go the other way: if block i and j are next to each other on the grid, the action to get from i to j is represented by $j - i$, i.e. if $j - i$ is +1, the move is moveeast , if −1, the move is movewest , if −21, the move is movenorth , and if +21, the move is movesouth .

Note: The player is facing south, so do not get confused if moving south is forward

2 What Do I Submit?

Here we'll describe what exactly you need to submit to the assignment

    1. Code: Finding Start and End Blocks: As a simple exercise, implement the find_start_end function, that takes the grid as an array of string describing the block types, and returns the indices of the start and the end block. This should only require, at maximum, a few lines of code.

    2. Output: Start and End Block Indices : Run the code with the above implemented, and look at the output lines that start with Output (start,end).

    3. Code: Shortest Path Implementation : Implement Dijkstra's algorithm in order to find the shortest path from the source to the destination. The set of possible actions from each block is one step in north, south, east, or west directions, i.e. taking multiple or diagnol steps is not allowed. The path you compute should be shortest in terms of number of moves (all of them cost the same), should be a list of block indices (integers), should include both the start and the end blocks, and of course, should not contain any air blocks. Submit the complete implementation of your function.

    4. Output: Length of the Shortest Paths: Run the code with the above implemented, and look at the output lines that start with Output (path length) and with Output (actions).

Eg :

```
Mission 2 running.
1
Output (start,end) 2 : (220, 326)
Output (path length) 2 : 13
Output (actions) 2 : ['movewest 1', 'movesouth 1',
 'movesouth 1', 'movewest 1', 'movewest 1', 'moves
outh 1', 'movesouth 1', 'movesouth 1', 'moveeast 1
', 'moveeast 1', 'moveeast 1', 'moveeast 1']

Mission 2 ended
Size of maze: 7
```