The Amazing Maze Runner

October 2023

Figure 1: From: https://www.slashfilm.com/

Introduction

You find yourself in the center of a maze. Can you find your way out? In the final exercise of this course the challenge is to write and implement an algorithm that leads you out of the maze.

You should solve the maze with no prior knowledge of the maze. This means you only learn about the maze by interacting with it as you would do in real life. Algorithms that are based on "seeing" the whole maze at once are not allowed. This is further enforced by the provided maze environment (see next section).

Interacting with the maze

You are provided (see Canvas) with a python script `maze_env.py` that allows you to create, visualize and interact with the maze. A very basic use is described here.

First you need to import the function `make()`:

```
from maze_env import make
```

When you call the function `make()` it returns an environment that simulates the maze:

```
env = make(size=7, seed=12345, perfect=True)
```

You specify the size (should be uneven and larger or equal to 7). In the example the maze has a size of 7×7. If you do not provide a value for the optional argument `seed` you will get a random maze each time you call the function. If you want to get the same maze each time you can specify a value for `seed`. The value should be integer. Each time you call the function with the same value you get the same maze. This can be useful in the testing phase. Finally you can specify the optional argument `perfect` (default value is `True`). If set to `True` the maze will be a so-called *perfect* maze which has a single solution[1]. If you set `perfect=False` the maze is no longer perfect and there will likely be multiple unique paths out of the maze.

[1] See assignment 2 of this document for a more detailed explanation

You can then reset the environment to get your initial position in the maze[2].

```
loc, done = env.reset()
```

The function returns the state of the environment which is composed of the current location `loc` (tuple of two integers indicating row number and column number in the maze) and a boolean `done` indicating if you reached the exit or not.

To check how the maze looks like you can render it:

```
env.render()
```

In this case this looks like:

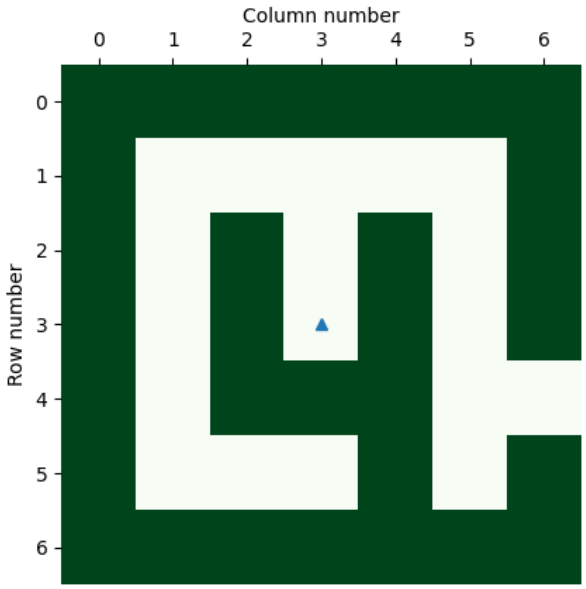

Figure 2: A maze of size 7×7

As you see, the maze has 7 rows and 7 columns. The walls are indicated with green. The blue triangle indicates your current position (row 3, column 3) in the maze. The blue triangle points upwards, which means that you are facing up/north. The exit in this maze is located at row number 4 and column number 6.

To move your position in the maze you can execute an action. An action is an integer value that specifies the direction in which you move, see Table for an overview.

Table 1: Possible actions to manoeuvre inside the maze. Directions are relative to the maze!

Action	Meaning
0	move up
1	move right
2	move down
3	move left

Let's move upward:

```
action = 0
loc, done = env.step(action)
```

When you execute the `env.step()` function, the environment tries to carry out the action (in this case `action=0`, so it tries to move upward). If this is possible (there is no wall) the environment moves you one

[2]Each time you call `env.reset()` the environment moves you back to the starting position.

position upward. The function returns the new location (tuple of two integers indicating row number and column number) and a Boolean indicating if you reached the exit or not.

```
print(loc, done)
```

yields:

```
(2, 3) False
```

So the new position is row number 2 and column number 3 and you are not yet at the exit of the maze.
The environment also has a function that returns a list of all possible actions given your current position:

```
actions = env.action_space()
print(actions)  # gives [0, 2]
```

Executing this code gives `[0, 2]`. This means that from your current position in the maze (2, 3) you can only use actions 0 (upward) or 2 (downward). Going left or right is not possible because there are walls.
Finally, if your script figured out the path out of the maze you can plot the path (defined as a 2D ndarray) in the maze:

```
path = np.array([[3, 3], [2, 3], [1, 3], [1, 4], [1, 5],
                 [2, 5], [3, 5], [4, 5], [4, 6]])
env.render(path=path)
```

Gives as output the maze with the path:

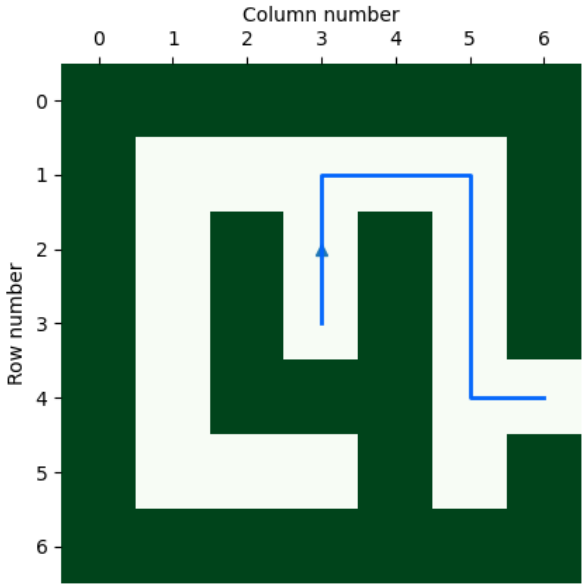

Figure 3: Rendered maze with a path

All available functions and methods are listed below:

Table 2: Overview of the available functions and methods of the maze environment.

Function/method	Explanation
`env = make(size, seed=None, perfect=True)`	Creates and returns a maze environment.
`loc, done = env.reset()`	Resets the environment to the initial state and returns the state
`env.render(path=)`	Renders the maze in a matplotlib figure
`actions = env.action_space()`	Returns a list of possible actions
`loc, done = env.step(action)`	Executes the specified action and returns the state

Assignment 1: Get Started

In this assignment you make yourself familiar with the provided python function to create and interact with the maze environment.

a) Download the required python file `maze_env.py` to your working directory.

b) Write code that creates a maze environment with a maze of size 7. Try using a value for the argument `seed` and also try making an enviroment without the use of `seed`. Try all functions that are available (see Table above) and make sure you understand how they work and what their output is.

c) Solve the maze manually and define a variable (with name `path`) that contains the path as a 2D numpy array that contains the consecutive positions in the maze that leads you out of the maze. Render the maze but now with the path included.

d) You are now going to implement the worst algorithm possible to solve the maze: random actions. Define a function `way_out(env)` that continues to play random actions until you are at the exit of the maze. The function should return the path. TIP: try this function only for very small mazes or it takes forever to solve.

Assignment 2: The Algorithm

Here you are challenged to come up with an algorithm/strategy that gets you out of the maze. Obviously the strategy/algorithm depends on the properties of the maze. Therefore these properties are listed below:

- You start at a location in or near the center of the maze.
- There exists only one exit in either one of the four sides.
- There is only one possible solution out of the maze (this also means that there are no "islands" in the maze, or in other words you can not walk in a circle and end up at the same location again). This is sometimes referred to as a *perfect maze*.

Furthermore, as explained before, your algorithm is NOT allowed to rely on knowledge of the whole maze. Normally when you are inside a maze you do no have that knowledge so your algorithm cannot make use of this.

The assignment is to describe an algorithm that gets you out of the maze. Obviously the algorithm should consist of a set of explicit rules such that if someone, who is completely ignorant, would follow these rules he/she will escape from the maze.

Assignment 3: Implementation

Before actual implementation it is probably a good idea to write **pseudo-code** for your algorithm. Make sure that the pseudo-code is clearly readable, and uses for-loops etc when appropriate.

Implement the actual code for your algorithm using your pseudo-code as a guideline. Call this function `way_out(env)`. Include code to test your function and plot the maze including the path. Test the robustness of your algorithm by executing it for many different mazes and at least up to `size=101`.

Example output:

```
path = way_out(env)

# gives as output something like (depending on the maze ofcourse)
array([[3, 3],
       [2, 3],
       [1, 3],
       [1, 4],
       [1, 5],
       [2, 5],
       [3, 5],
       [4, 5],
       [4, 6]])
```

TIPS:

- Start testing with a relatively small maze size!
- `env.make(size)` creates random mazes of the specified size. For testing this may not be practically. Therefore you can use the `seed` argument to get the same maze. Obviously when the script works test it with many different mazes!
- If you find it difficult to get your algorithm working, consider plotting the maze and the path when you are looping through the code with `env.render()`.

IMPORTANT! Correct solutions are continuous paths that do not branch. For some algorithms you will obtain paths that sometimes crosses itself (i.e. comes back at a previous position but now from a different direction). In this case the path in between is redundant. If your algorithm shows this behaviour, add code that cleans up the path such that sections of the path that are redundant are removed.

Assignment 4: Characterization

Here you will measure how much time it takes for your algorithm to solve the maze. More specific, you investigate how the time depends on the size of the maze.

a) Figure out how you can time functions in Python. TIP: check out the `timeit` module.

b) Time your function `way_out()` for a range of maze sizes. Try to get the range of sizes as large as possible. Per size you probably need to average over quite some mazes as the execution time can vary between mazes.

c) Make a plot of the average execution time as function of the maze size.

d) Investigate the scaling behaviour. How does the execution time scale with the maze size n? Is it linear ($t \propto n$), exponential ($t \propto e^{c \times n}$) or does it follow a power law ($t \propto n^c$)? How do you test this?If applicable, determine the constant c.

e) Based on your analysis, how long would it take for your computer to solve a maze with `size = 1000000`?

Assignment 5: Speeding it up

Finally you are challenged to speed up your code such that it provides the solution to the maze in as less time as possible[3]. Time your code and compare your earlier results obtained. Document what changes you implemented to make the code faster. If you cannot find any possible improvements, explain what you tried and why you think it can not further be improved.

[3] It is also allowed to come up with an implementation of an alternative (faster) algorithm if you like.