Alpen-Adria-Universität Klagenfurt

Course of Information Search & Advanced Topics in Artificial Intelligence 1
Professor Alice Tarzariol

Report of the project: Maze Analyser

Francesca Favero, Ruggero Franz, Lorenzo Marcon
July 17, 2021

Academic year 2020/2021

Contents

1 Introduction

1.1 Problem description

We want to solve the exploration of a maze by an agent towards a fixed position. We started from a more complex problem with multi agents but then we went back to a simpler problem with a single agent for the discovery of the maze and its resolution. Then we saw what are the different solutions for this type of problem and we found three interesting ones. To solve this problem we used 3 different approaches.

1.2 Approaches

We began to solve the problem with what was the optimal solution for us, that is the approach explained during the lectures that is **Q-Learning**. We thought it was very interesting to solve it even with different approaches and then make a comparison with the Q-Learning approach. So, we also worked on two different resolution methods such as: Deep Q Network training and the top-down maze resolution approach (using the Djikstra algorithm).
The work was divided as follows:

- Ruggero worked more specifically on the approach that uses Q-Learning;
- Lorenzo worked more specifically on Deep Q Network training;
- Francesca worked more specifically on the approach that uses Djikstra.

1.2.1 Q-Learning maze resolution

In this approach we used the q-reinforcement learning where the agent is in a certain state and must decide what action to do based on the q-values that determine the actions available in that state. The q-values are calculated at each step with the formula seen in class. to balance exploration and exploitation, instead of having a fixed epsilon, we used the **GLIE** technique As for the environment we used a *5 x 5* grid, where the agent starts from the coordinates (0,0) and must arrive at the coordinates (4,4) and for testing we changed parameters, such as learning rate and gamma, to find out what is the optimal combination (understood as execution times and / or success rate) of these 2 parameters.

1.2.2 Deep Q-Network training maze resolution

The approach was carried out with the deep Q-Learning network and developed in Python using existing libraries developed for reinforcement learning such as keras. The deep learning network uses the concepts of Q-Learning when the space of states is too large. This avoids using large tables for the q-values. We don't want to use tables because they would be too heavy to handle in memory if the space is that big. On the other hand it requires a large amount of epochs to be able to calculate the Q-Values within these tables and therefore will require very long runs. The neural networks are exploited within this situation (ie a very large table) to have an approximation to the Bellman equation. Since we have a very large space, we obtain a very complex equation and therefore neural networks are right for us. Using the functions already implemented, for example in keras, we can define the agents with policies already implemented.

1.2.3 Top-down maze resolution

First we created the environment. We therefore built 10 mazes randomly and fixed the two agents. This code was then transformed into a png so you can see the results graphically. The blue agent is looking for the red agent. Once the data has been fixed, an algorithm analyzes the image of the maze and keeps in memory the starting point (blue agent) and the end point (red agent). Having found the inputs through the resolution of the Djikstra algorithm we find the fastest way in a short time. All this is then saved on the image of the maze so that you can see it graphically.

2 Plain Q-Learning

2.1 Background

This projects has the goal to teach an agent how to get out of a maze. The maze is a simple 5x5 size grid, without obstacles inside. The agent's starting position is in the lower left corner of the grid while the target position is in the upper right corner. The q-learning technique is used in order to make the agent learn how to get out of the maze. Also, for balancing the exploration and exploitation of the agent, the GLIE method is used.

2.2 Implementation

2.2.1 Enviorment

The enviorment is imported from the *Open AI Gym*'s libraries and is composed by a 5x5 size grid. Every position defines a state and some action are available in every state. There are 4 actions available in this environment: up, down, left, right. The position outside the grid are considered "*out of bounds*" and if an action leads to an out of bounds position, then the move is said *invalid* and a negative reward is provided to that pair *<state, action>*. If the action is valid, that is, it ends in a state that falls within the limits of the grid, then the agent changes position and will go to the cell indicated by the pair *<state, action>* and will receive a predefined reward.

2.2.2 Reward

For the 5x5 size grid, our target state is the cell with coordinates (4,4) (that is the opposite corner to that of the agent's starting position). Reaching the target state provides a reward of +10000 while doing an action that would take the agent out of the boundaries of the grid provides a reward of -100. In all other cases (the agent is brought into a valid state) the reward is calculated as follows: $new_target_distance = np.linalg.norm(self.state - self.target_state)$. That is, if the distance of the current state from the target state is smaller than the distance of the previous state from the target state (so the agent is getting closer to the goal), then the reward is positive +1, otherwise (so the agent is moving away) negative -1.

2.3 Learning phase

The learning of the agent is done in *episodes* and each episode has a certain length ($env.path_length$) which is measured by the number of actions done by the agent. The agent, at each step, can decide whether to make an *exploitation* or *expoloration* move. This decision is made based on the *epsilon* probability which is calculated according to the *GLIE* methodology:

```
    $epsilon = 1/env.path_length.$
IF $random.uniform(0, 1) < epsilon$
THEN
EXPLORE: select a random action
ELSE
EXPLOIT: select the action with max value (best future reward)
```

Each episode starts with $env.path_length$ equal to 1, so the first few moves will have a high chance of being exploratory (and therefore random). As the episode progresses, and therefore the value of $env.path_length$ grows, the likelihood of the agent taking an exploitation action increases.

For each action performed (that is at every step) the variable $env.path_length$ is increased by 1 and a Q-VALUE is associated to the pair $< action, state >$. In every moment, the agent is in a certain state (that is a position) and when an action (that is a move) is done, the agent changes position and goes in a new state. Then, the Q-value table is update with the following formula: $Q_table[state, action] += learning_rate * (reward + gamma * np.max(Q_table[new_state]) - Q_table[state, action])$.

The actual learning is done with some *'runs'*. Each run consists of 500 episodes, each one 200 steps long. Each run has a *Q-table* which is updated at each step of each episode. The runs differ in the values of learning rate and gamma, in order to find the optimal combination. At the end of each *run* we should obtain a Q-table containing the values that allow to reach the target state easily (not optimal, not having a reference policy).

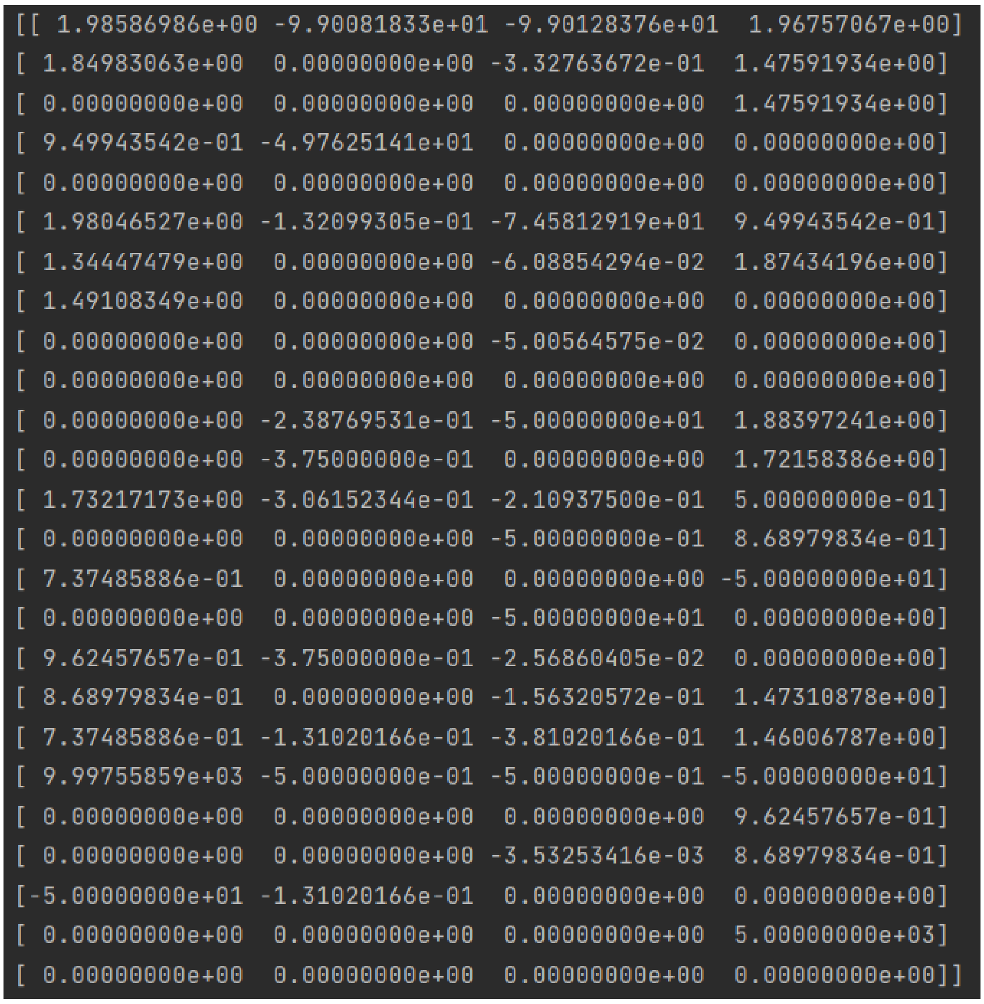
An example of Q-table output is shown in 2.1.

2.4 Testing Phase

From the *Learning phase* of the agent we have obtained a *Q-table* for each pair of *Learning Rate* and *Gamma* values which are fed into the *Testing phase*. In the *Testing phase* the agent always executes the action with the highest q-value (no exploration), thus obtaining a deterministic behavior. This allows us to run a single test for each Q-table.

The testing phase aims to find out what are the optimal values of *learning rate* and *gamma*. The strategy we used to determine which parameter to use to evaluate the $< lr, gamma >$ pair is as follows: if the agent reaches the target state, then the number of *steps* it took to reach it is taken as a parameter, the total *score* otherwise.

One thing we have noticed is that, as the learning rate increased, the generated Q-tables were unable to bring the agent to the target state. High values of *Learning Rate* make the update function for the Q-values too much sensitive to the variations. In figure **??** it is possible to see the trend of the score values as the learning rate values vary.



```
[[ 1.98586986e+00 -9.90081833e+01 -9.90128376e+01  1.96757067e+00]
 [ 1.84983063e+00  0.00000000e+00 -3.32763672e-01  1.47591934e+00]
 [ 0.00000000e+00  0.00000000e+00  0.00000000e+00  1.47591934e+00]
 [ 9.49943542e-01 -4.97625141e+01  0.00000000e+00  0.00000000e+00]
 [ 0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00]
 [ 1.98046527e+00 -1.32099305e-01 -7.45812919e+01  9.49943542e-01]
 [ 1.34447479e+00  0.00000000e+00 -6.08854294e-02  1.87434196e+00]
 [ 1.49108349e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00]
 [ 0.00000000e+00  0.00000000e+00 -5.00564575e-02  0.00000000e+00]
 [ 0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00]
 [ 0.00000000e+00 -2.38769531e-01 -5.00000000e+01  1.88397241e+00]
 [ 0.00000000e+00 -3.75000000e-01  0.00000000e+00  1.72158386e+00]
 [ 1.73217173e+00 -3.06152344e-01 -2.10937500e-01  5.00000000e-01]
 [ 0.00000000e+00  0.00000000e+00 -5.00000000e-01  8.68979834e-01]
 [ 7.37485886e-01  0.00000000e+00  0.00000000e+00 -5.00000000e+01]
 [ 0.00000000e+00  0.00000000e+00 -5.00000000e+01  0.00000000e+00]
 [ 9.62457657e-01 -3.75000000e-01 -2.56860405e-02  0.00000000e+00]
 [ 8.68979834e-01  0.00000000e+00 -1.56320572e-01  1.47310878e+00]
 [ 7.37485886e-01 -1.31020166e-01 -3.81020166e-01  1.46006787e+00]
 [ 9.99755859e+03 -5.00000000e-01 -5.00000000e-01 -5.00000000e+01]
 [ 0.00000000e+00  0.00000000e+00  0.00000000e+00  9.62457657e-01]
 [ 0.00000000e+00  0.00000000e+00 -3.53253416e-03  8.68979834e-01]
 [-5.00000000e+01 -1.31020166e-01  0.00000000e+00  0.00000000e+00]
 [ 0.00000000e+00  0.00000000e+00  0.00000000e+00  5.00000000e+03]
 [ 0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00]]
```

Figure 2.1: Q-table after some episodes: each row represent a state (a 5x5 size grid has 25 states) while the columns represent the 4 possible action in every state. Every pair $< state, action >$ has a value: if in the whole run, an action is not performed in a state then that pair will have value 0. Otherwise the value is given by the q-value update formula.

From this plot we can establish, empirically, that the best values for the *Learning Rate* lie between 0 and 0.5.

2.5 Conclusion

The goal of this task was to teach an agent how to get out of a space (very simple, without obstacles). To do this, it was decided to use the *reinforcment learning* method together with the *GLIE* technique. This last technique is in my opinion "improvable", that is, it can be implemented in different ways to obtain a better result. The problem with standard implementation is that only after a few steps does the probability of exploration drop significantly (after only 10 steps we have a 10% chance of making an exploration move) and therefore we have a long-tail distribution. An improvement would be to divide the episodes of the training phase into two parts: in the first part we use $1 - epsilon$ (so

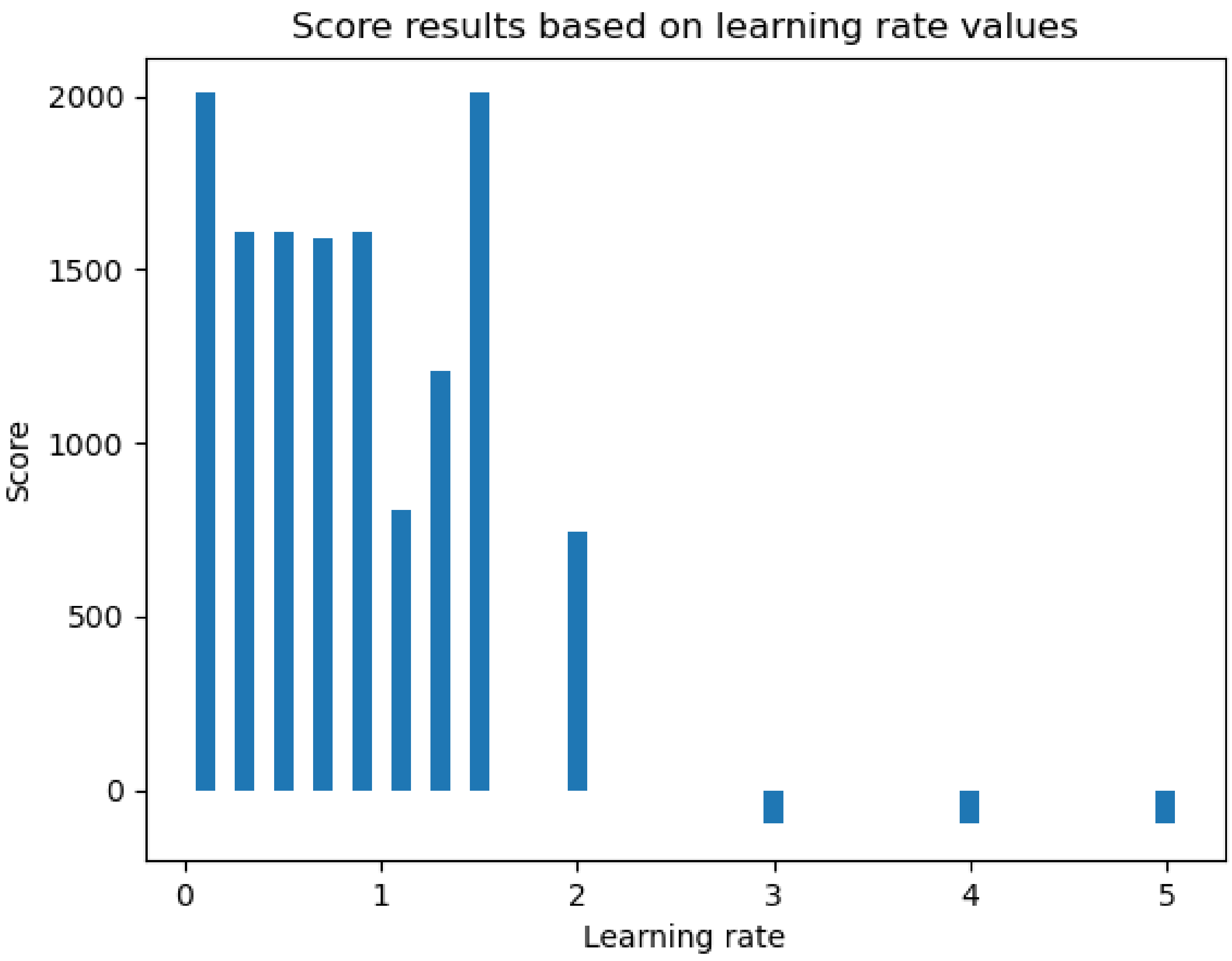


Figure 2.2: The average score of the episodes decrease as the learning rate value increases.

at the beginning of the episodes there will be a high probability of making exploratory moves) while in the second part of the episodes we only use *epsilon*. In this way there will be the possibility to make exploratory moves even towards the end of the episode.

Another interesting approach could be to make a smoothing in order to linearize the probability distribution.

3 Deep Q-Network training maze resolution

3.1 Background

The task was developed in python using available libraries for reinforcement learning such as Keras and Gym.

- **Keras:** An open source library that provides an interface for artificial intelligence and neural networks tasks. It is based on the interface of TensorFlow library.
- **OpenAI Gym:** It is a toolkit for developing and comparing reinforcement learning algorithms. Gym library is a collection of environments, without assumptions about the agent, and support the reinforcement learning libraries (Tensorflow and Theano). The environments have a shared interface to help the developers to write algorithms and custom environments.

Deep Q Learning is suitable for this task because it is a typically problem framed as Markov Decision Process (a set of states S and actions A). The transitions are performed with probability P and reward R for a discounted gamma. The Q-Network proceeds as a non linear approximation which maps states in an action value. During the training, the agent interacts (fits) with the environment and the data received during the learning of the network. The original article that start DQN can be found here https://www.cs.toronto.edu/ vmnih/docs/dqn.pdf.

The final goal is to approximate the non linear function *Q(S,A)* with the deep learning of a multi layers neural network. The DQN, like for the supervised neural network, has loss function to predict the next state (assuming state s, action a and reward r).

3.2 Implementation

The source code is divided in two files *main.py* and *nnModel.py.* In the first one are defined:

- **Grid class:** Our custom environment, defined with gym interface for environments, which required to override specific methods:
  - *init:* The class initializer requires the subset of *walls states*, maximum dimension of the grid and the available steps for the agent (*path_length*). The initializer defines attributes of the maze: action space, observation space, actual state, path lenth, target state and the walls susbset.

  - *step:* This function set the movement and the reward of the agent. It returns the state, reward, done (a boolean that check the status of the episode) and a log variable called info.
  - *render:* Visualization of the problem, not implemented for this task.
  - *reset:* At the end of the episode it resets actual state and path variables.

- **model:** In this variable is stored the status of the neural network, e.g. it is possible check the architecture using *.summary()*.

- **utilities:** Several functions are defined to plot, time logging, printing and saving weights of the network.

In the second file are implemented the deep neural network and the agent:

- **build_model:** Requires in input the dimension (or shape) of possible states and the number of actions. It is a deep sequential model, so each of the three layer is activated in sequence and the output layer has linear function that returns and array with the four probabilities of the 4 actions available. The structure is the following:

  1. **Input layer:** It receives the states and actions and start to build with 64 neurons and an activation function "ReLU" that fires to the next layer.
  2. **Hidden layer:** It reduces the features space to 32 neurons and fires with ReLU activation function to the last layer
  3. **Output layer:** It reduce the actions space with a linear function to the actual agent's actions and returns the four probabilities.

- **build_agent:** Requires in input the nn model and the number of actions. It returns a dqn agent with a policy (BoltzmannQPolicy), a memory state and the actual trained agent.

  BoltzmannQPolicy: The action distribution is divided by actions parameters. The Boltzmann policy build a spectrum between picking the action randomly and the most optimal solution. In fact, the distribution gives the probability measure that a system will be in certain state of a function with parameters (in the original work are temperature and energy).

  The agent *dqn* is returned to the main process for the fitting with model and environment.

3.3 Experiments

The environment is initialize calling *Grid* class and passing the required parameters.

For the training experiments we are considering a grid 64x64 with 11 obstacles and a path length of 500 steps.

In the picture below are reported some results of the network training in 130 epochs. It is possible see the time interval for each epochs, between 0.5 second and 4 sec. The

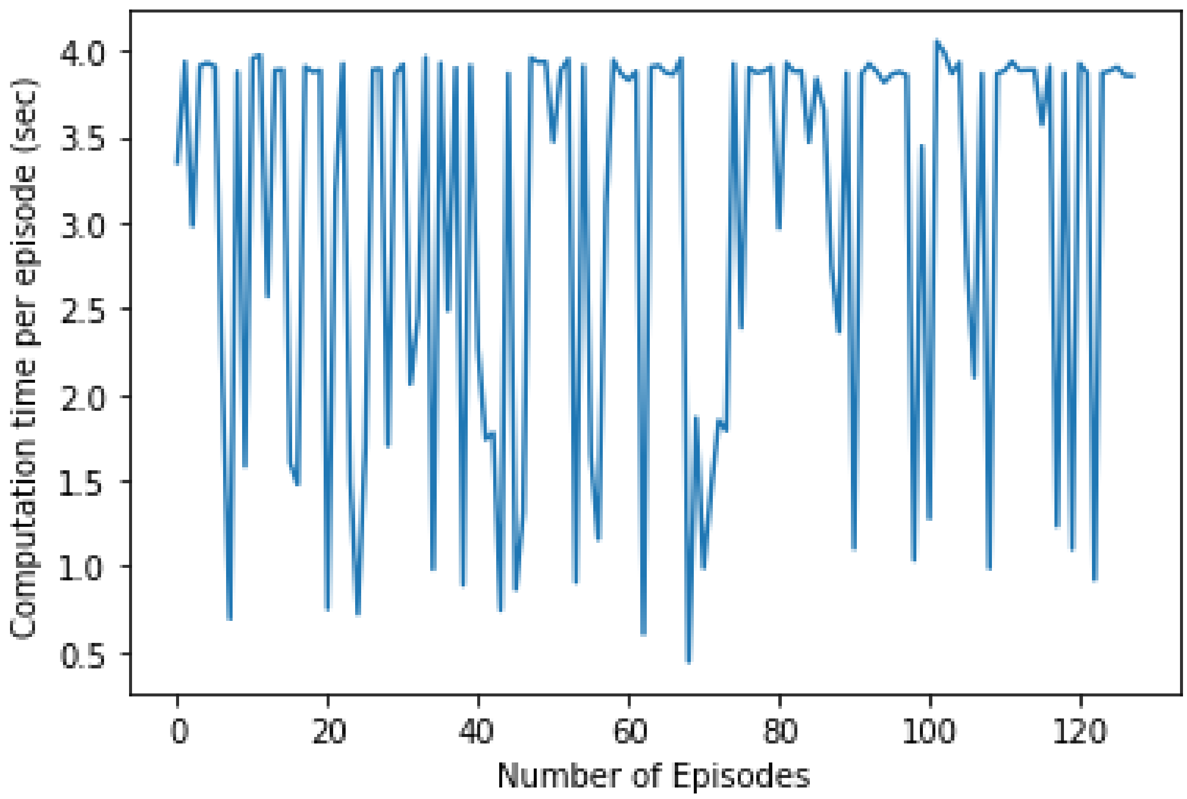


Figure 3.1: Time to resolution per epoch

difference between the epochs it is to be brought to the machine time to do 500 steps and the achivement from the agent of the target state.

The expected mean time distribution per episode is a monotonic descendent function. We are expecting an inversion of the following histogram, where the frequencies with high frequencies on short computational time and less number of longer runs will be exchange.

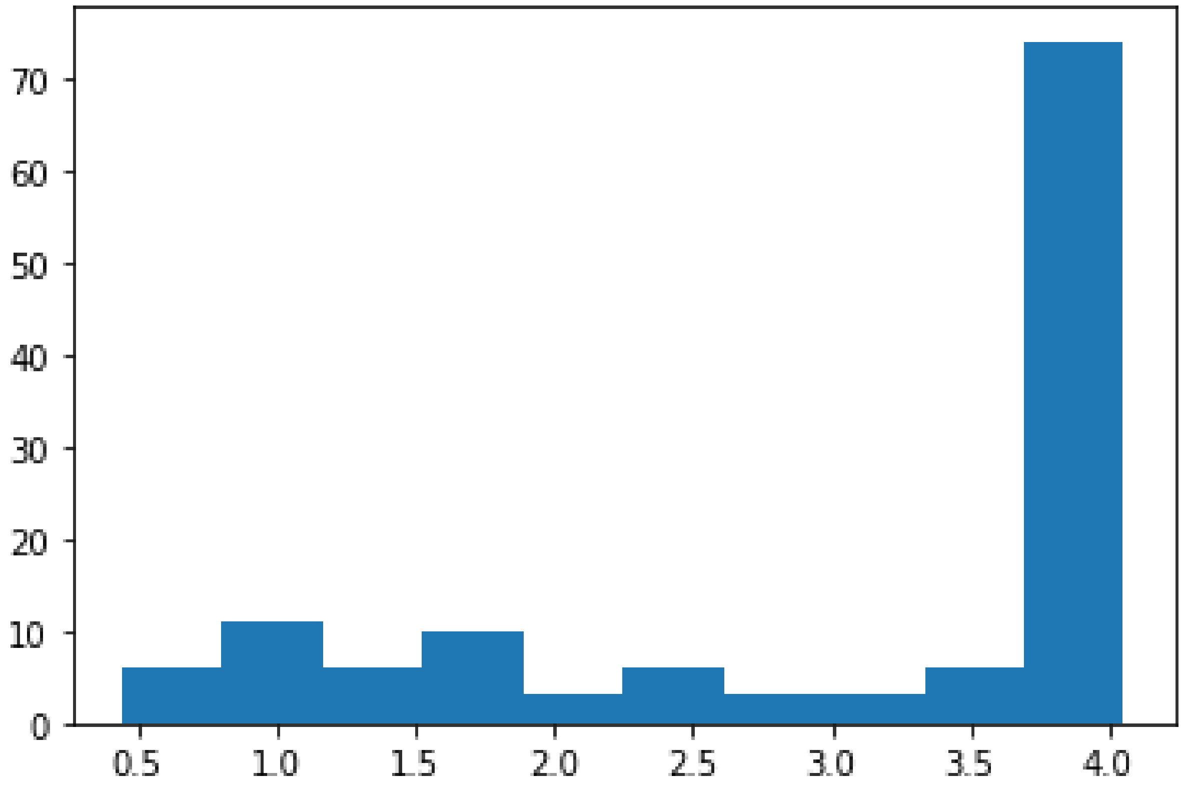


Figure 3.2: Time distribution of the epochs

4 Top-down maze resolution approach

4.1 Background

This task was developed in python. The imports used are listed below.

- The following libraries were used in maze_generator:
  - `from queue import LifoQueue` : it is used to create a Last In First Out stack where adjacent nodes are inserted.
  - `from PIL import Image`: it is used to create images where mazes are saved graphically.
  - `import numpy as np`: creates cells and fills them with zero. It is used to initialize the maze.
  - `from random import choice`: it is used to randomly choose a neighbor so that the mazes are always all different.
  - `import agents`: sets up agents in the maze
  - `import argparse`: makes it easy to write intuitive command line interfaces. The argparse module is useful because it also automatically generates help and usage messages and generates errors when users provide invalid arguments to the program.
  - `import os`: it is used to create the folder containing the mazes.
  - `import sys`: it is used to do multiple runs of the same code.
- The following libraries were used in agents:
  - `import cv2` : it is used to read the maze image and add agents to it.
  - `import random`: it is used to place the red agent in a random position in path.
  - `import numpy as np`: it is used to find all valid positions in the maze.
- The following libraries were used in maze_analyser:
  - `from typing import List` : generates a list of nodes for a maze for navigation so that they are linked up width weighted edges based on the distance from one another in the maze.
- The following libraries were used in dijkstra:

- `import functools`: serves for higher-order functions: functions that act or return other functions.
- `from queue import PriorityQueue`: it is used to keep in memory the stack of nodes that make up the path between the start node and the finish node
- `from typing import List`: it is used to return the list of nodes that form the shortest path to reach the agent.
- `import numpy as np`: used to create color fades between blue and red (only used to improve the readability of the solution).
- `from PIL import Image`: used to open the maze image (input image) and to load its pixels. Finally used to save the new output image.
- `import time`: used to calculate the execution time of the Dijkstra algorithm
- `import os`: used to create the results folder and save the images that solve the mazes in it.
- `from maze_analyser import Node, nodes_from_maze`: used as an input to the algorithm. To know how the maze is made and then subsequently be able to solve it.
- `import argparse`: Used in the same way as maze_analyser.

4.2 Implementation

For this type of task we need 4 files. To show how it works, run *maze_generator.py* and then *dijkstra.py*: 2 folders will appear, one with the generated mazes and one with the resolution of these.

- **maze_generator**: creates the mazes with different sizes. With the settings currently set, it creates 10 mazes with 3 different sizes. The mazes consist of a white path, black walls and two agents. The blue agent is always positioned at the entrance to the maze in the coordinates (1,0) while the red agent is positioned randomly. The output is saved in an image
- **agents** : creates the agents and place them in the maze.
- **maze_analyser**: analyzes the image of the maze, keeps in memory the position of the blue agent (start position), all the nodes on which it could move (the path) and the position of the red agent (the arrival position). The maze must be made up of black and white pixels, the start point must be either on the left and top side,the end point must be on the red agent.
- **dijkstra**: Using the input values provided by maze _analyser it searches for the shortest path from the initial node (blue agent) to the final node (red agent). Dijkstra's algorithm is used to find the shortest path. Once the set of nodes forming the shortest path is found, this path is saved to an image and colored in a blue to red gradient. Finally, the blue agent is placed in place of the red agent to simulate its displacement.

Dijkstra's algorithm is an algorithm for finding the shortest paths between nodes in a graph. It is uses labels that are positive integers numbers, which are totally ordered. Dijkstra's algorithm uses a data structure for storing and querying partial solutions sorted by distance from the start. Let the node at which we are starting be called the initial node (in our case the blue agent). Let the distance of node Y be the distance from the initial node to Y. The final node will be the red agent who is randomly placed in the maze (in the white path). Dijkstra's algorithm will assign some initial distance values and will try to improve them step by step.

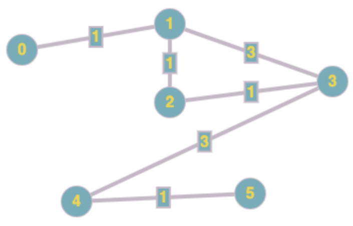


Figure 4.1: Dijkstra example

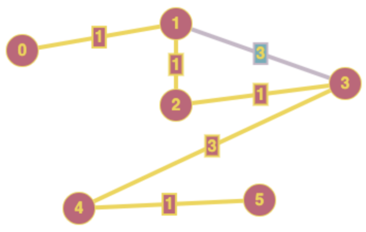
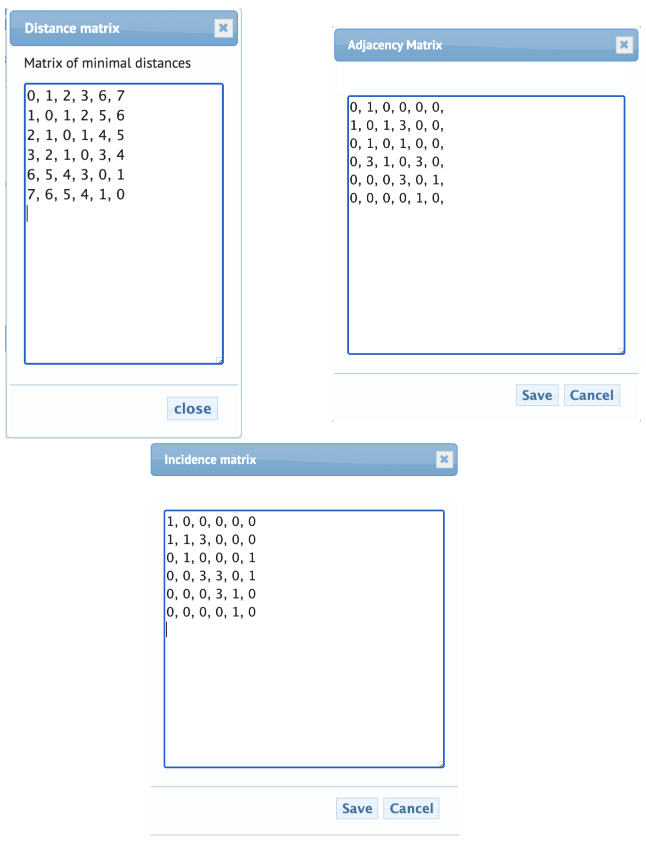


Figure 4.2: Dijkstra example solved



Matrix of minimal distances

```
0, 1, 2, 3, 6, 7
1, 0, 1, 2, 5, 6
2, 1, 0, 1, 4, 5
3, 2, 1, 0, 3, 4
6, 5, 4, 3, 0, 1
7, 6, 5, 4, 1, 0
```

```
0, 1, 0, 0, 0, 0,
1, 0, 1, 3, 0, 0,
0, 1, 0, 1, 0, 0,
0, 3, 1, 0, 3, 0,
0, 0, 0, 3, 0, 1,
0, 0, 0, 0, 1, 0,
```

```
1, 0, 0, 0, 0, 0
1, 1, 3, 0, 0, 0
0, 1, 0, 0, 0, 1
0, 0, 3, 3, 0, 1
0, 0, 0, 3, 1, 0
0, 0, 0, 0, 1, 0
```

Figure 4.3: Distance Matrix, Adjacency Matrix and Incidence Matrix

4.2.1 Output

The resulting output will be the fastest path from agent blue to agent red. For example, from the situation on the left (Figure 4.4) generated with maze_generator, Dijkstra's algorithm will produce the solution on the right (Figure 4.5). This maze is one of the simplest generated but the difficulty can be increased by setting the file maze_generator. All mazes have a solution and the red agent does not move since Dijkstra's algorithm does not learn so it would never be able to reach the red agent if it moved.

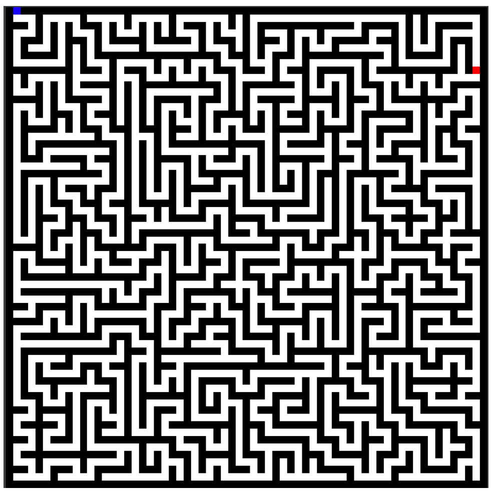


Figure 4.4: Maze

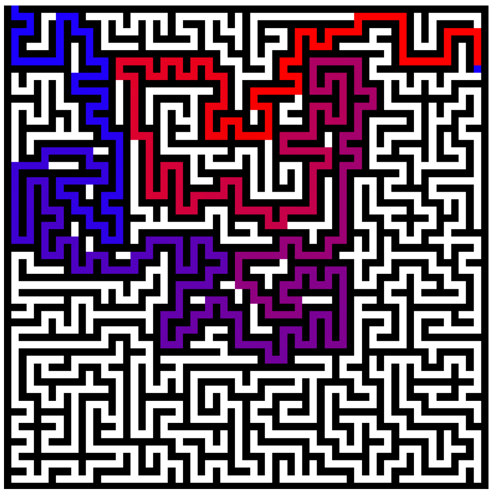
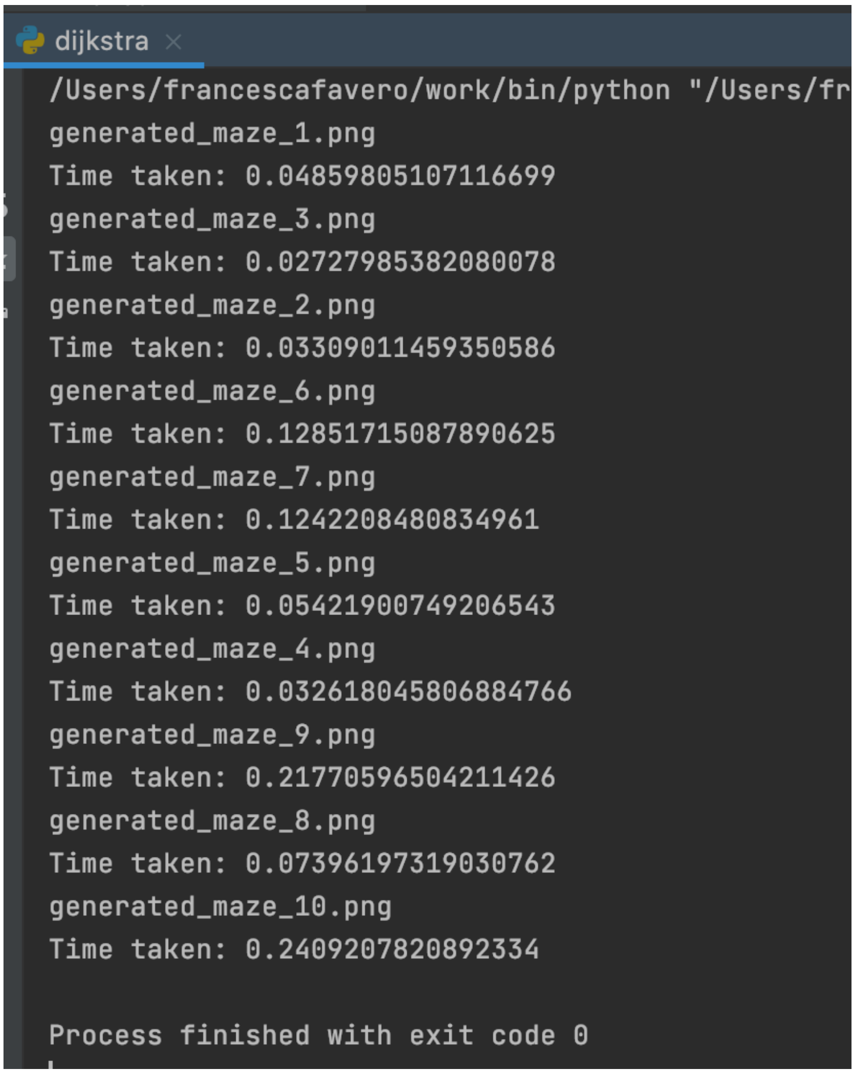


Figure 4.5: Maze solved

4.3 Experiments

Using Dijkstra's algorithm the experiments were to measure the running time by changing the size of the maze. The difficulty between the first maze and the last is tripled in fact you can see how (Figure 4.6), even if the resolution times are very short, there is a difference between the first and the last resolution time. Obviously the red agent is always placed at random so it is difficult to make fair comparisons but it can be noted that the execution time has an increase.



```
/Users/francescafavero/work/bin/python "/Users/fr
generated_maze_1.png
Time taken: 0.04859805107116699
generated_maze_3.png
Time taken: 0.02727985382080078
generated_maze_2.png
Time taken: 0.03309011459350586
generated_maze_6.png
Time taken: 0.12851715087890625
generated_maze_7.png
Time taken: 0.1242208480834961
generated_maze_5.png
Time taken: 0.05421900749206543
generated_maze_4.png
Time taken: 0.032618045806884766
generated_maze_9.png
Time taken: 0.21770596504211426
generated_maze_8.png
Time taken: 0.07396197319030762
generated_maze_10.png
Time taken: 0.2409207820892334

Process finished with exit code 0
```

Figure 4.6: Time Taken by Dijkstra Algorithm

Conclusions

4.4 Three different approaches for a maze resolution

These three approaches cover some current methodologies such as *deep q-learning* and *Dijkstra* and methods seen in class such as the simple *q-learning* algorithm. Overall, the application of q-learning approach seemed suitable for this type of problem.

The biggest differences between the 3 different approaches we found were in the training phase: in the dijkstra approach the step is not necessary, being a deterministic algorithm. In the other 2 cases the q-tables proved to be quite effective. Più specificamente nel caso del *simple q-learning* sono bastati 50-70 episodi (di lunghezza 200 passi l'uno) per avere una q-table che conducesse l'agente al *Target State*. On the other hand, in the case of neural *networks*, training takes much longer because the number of parameters to be trained is high. Furthermore, the results obtained from the training phase were not as excellent as initially thought. On a positive note, this approach scales well: as the size of the grid increases, it has proved more efficient than the population done in the case of *simple q-learning*.

Since *neural networks* need to be formalized (find optimal values), trained and tested, this appears to be a much greater effort than the *simple q-learning* approach. Effort not justified by the results that were not satisfactory compared to expectations. In this sense it seems convenient to use the *simple q-learning* approach.