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 16, 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 that uses **Q-Learning**. We talked about it a lot in class. But then 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-reinforcment 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) then follows data analysis, changing 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 tables. 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 which is used for the resolution of the Q-Values. 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, you can define the agents with their policies already implemented and the actions that are then carried out by them based on the table found by the network and so you can go to select those that are the converging values and to solve the problem

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 Parte di Ruggero

2.1 Background

2.2 Implementation

2.3 Experiments

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 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

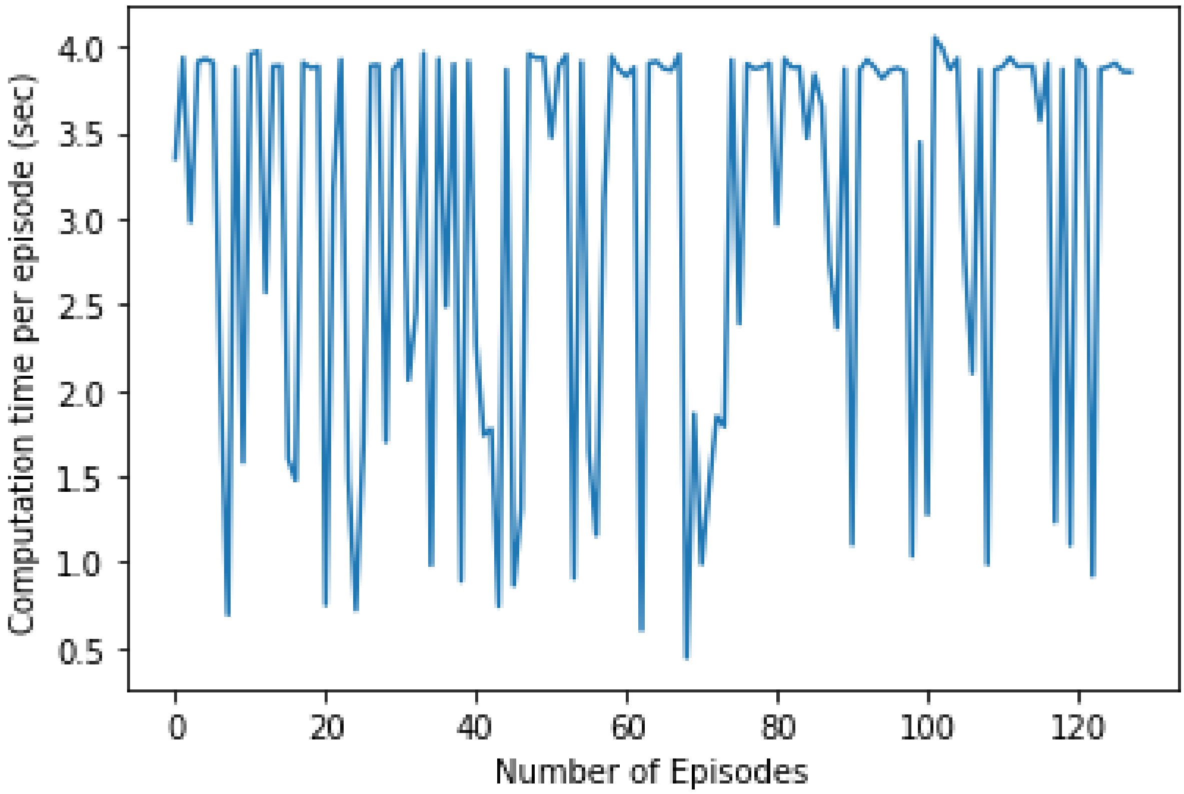

Figure 3.1: Time to resolution per epoch

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.

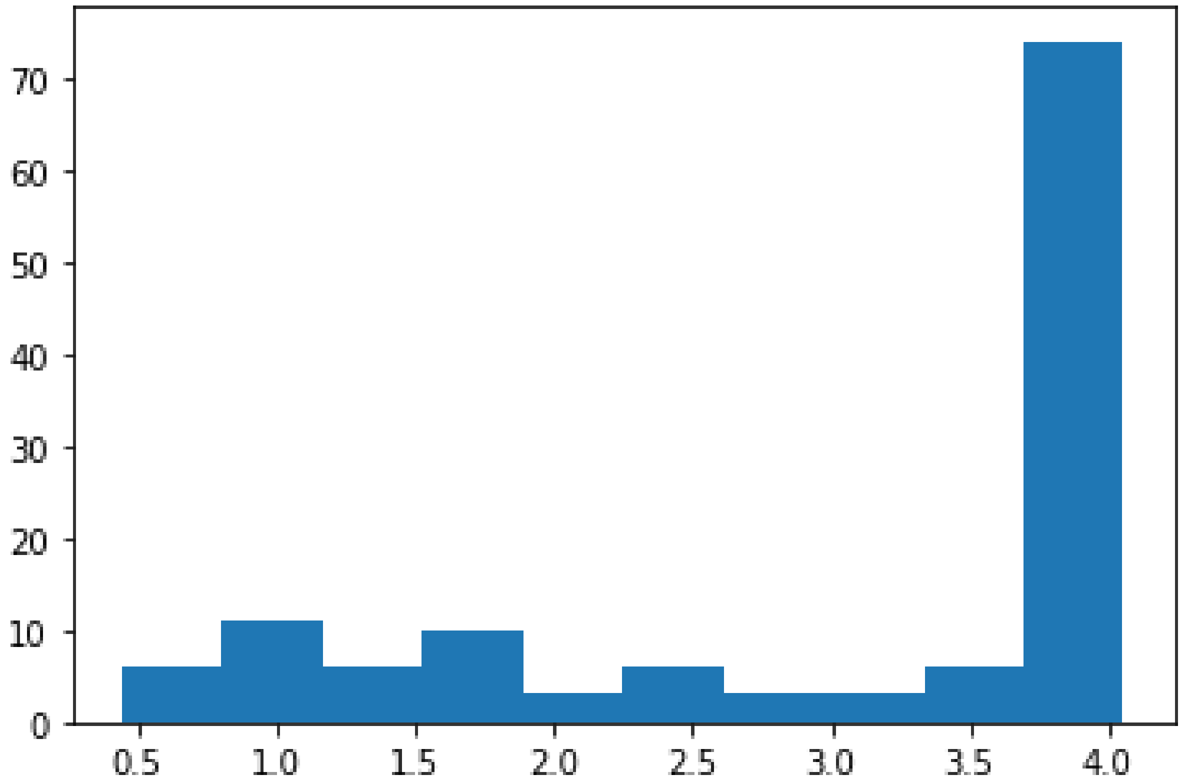

Figure 3.2: Time distribution of the epochs

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.

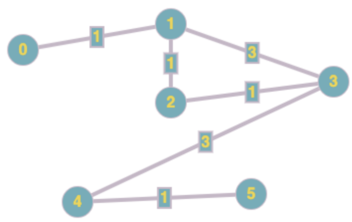

Figure 4.1: Dijkstra example

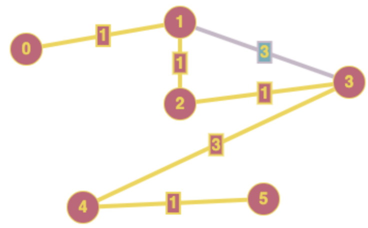

Figure 4.2: Dijkstra example solved

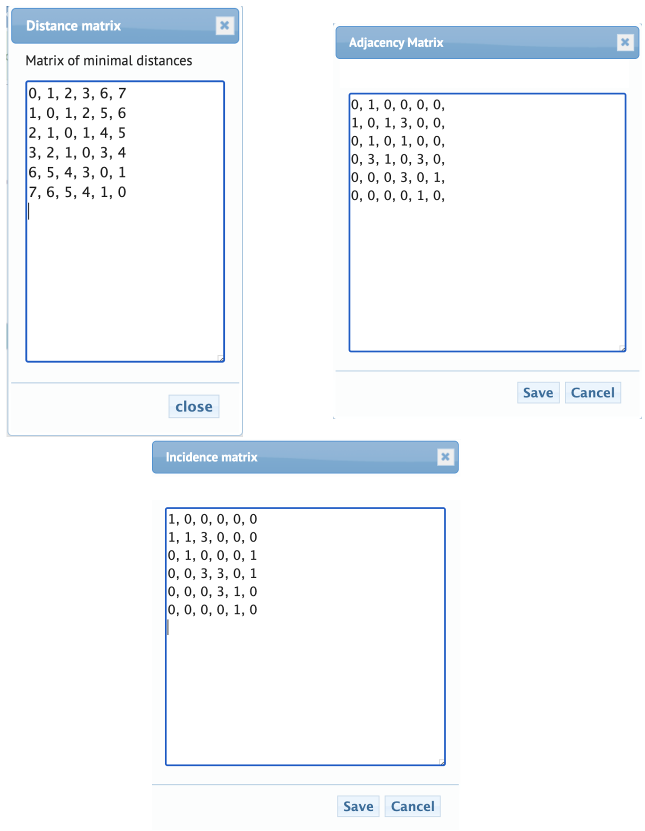

Figure 4.3: Distance Matrix, Adjacency Matrix and Incidence Matrix

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.

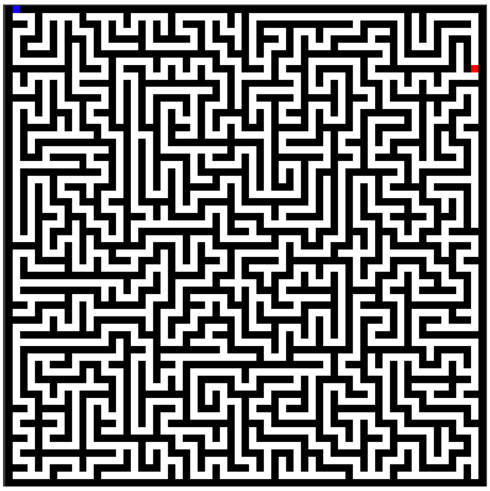

Figure 4.4: Maze

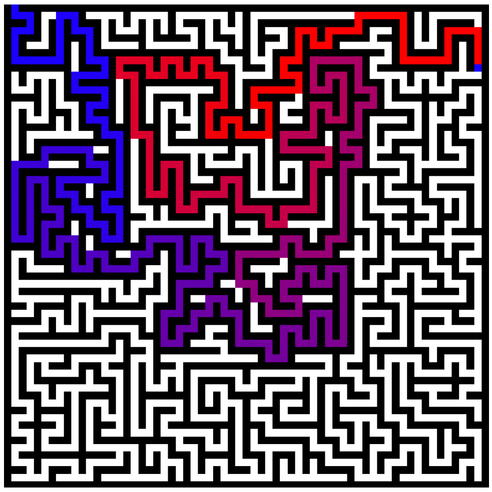

Figure 4.5: Maze solved

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.

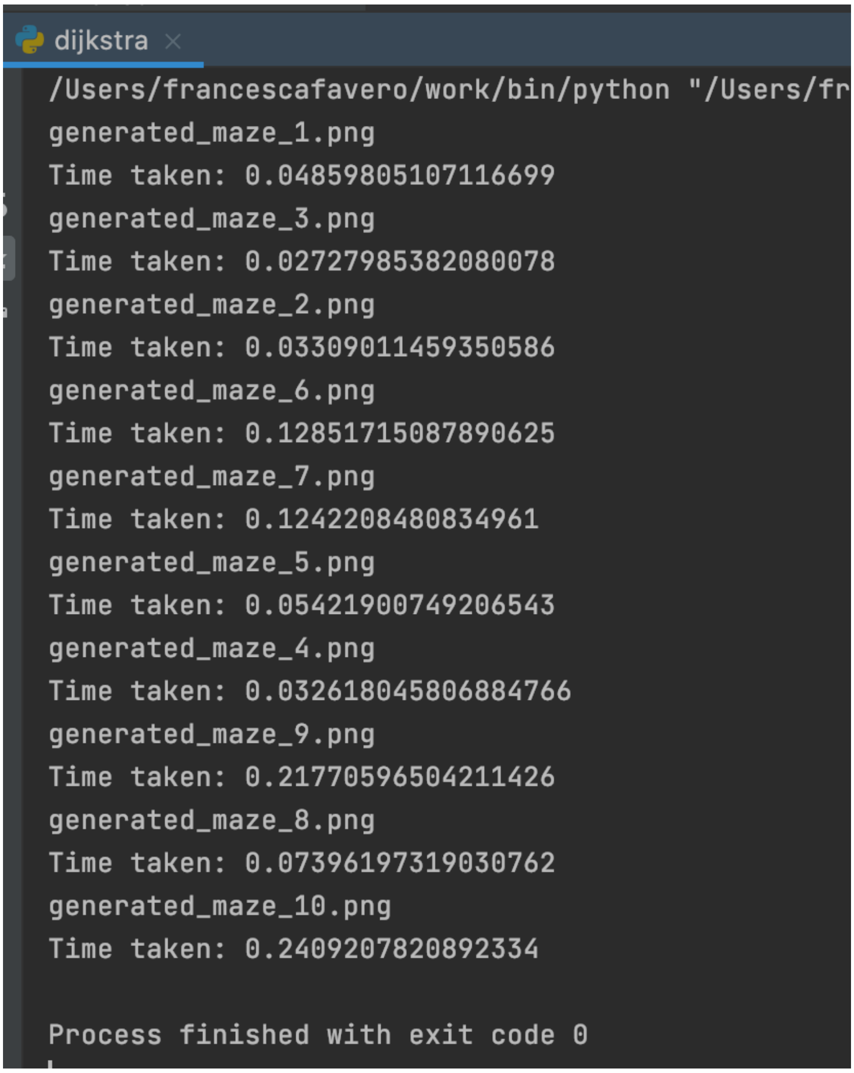

Figure 4.6: Time Taken by Dijkstra Algorithm

Conclusions

4.4 Maze Resolution with Q learning

The Q Learning results are reported in the following plots. The idea is to do some confrontation between grids of different dimensions.