



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 8, 2021

ACADEMIC YEAR 2020/2021

Indice

1	Introduction	1
1.1	Problem description	1
1.2	Approaches	1
1.2.1	Q-Learning maze resolution	1
1.2.2	Deep Q-Network training maze resolution	2
1.2.3	Top-down maze resolution	2
2	Parte di Ruggero	3
2.1	Background	3
2.2	Implementation	3
2.3	Experiments	3
3	Parte di Lorenzo	4
3.1	Background	4
3.2	Implementation	4
3.3	Experiments	4
4	Top-down maze resolution approach	5
4.1	Background	5
4.2	Implementation	6
4.2.1	Output	8
4.3	Experiments	8
	Conclusions	10

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-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 5×5 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. Approaches

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 Dijkstra 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 Parte di Lorenzo

3.1 Background

3.2 Implementation

3.3 Experiments

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 with weighted edges based on the distance from one another in the maze.
- The following libraries were used in `dijkstra`:

4.2. Implementation

- `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.

4.2. Implementation

Dijkstra's algorithm is an algorithm for finding the shortest paths between nodes in a graph. It 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.

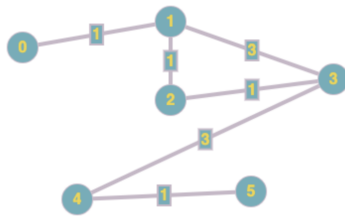


Figura 4.1: Dijkstra example

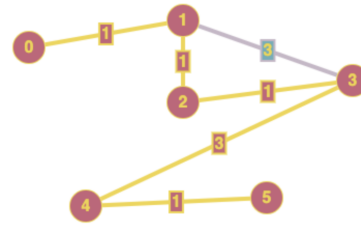


Figura 4.2: Dijkstra example solved

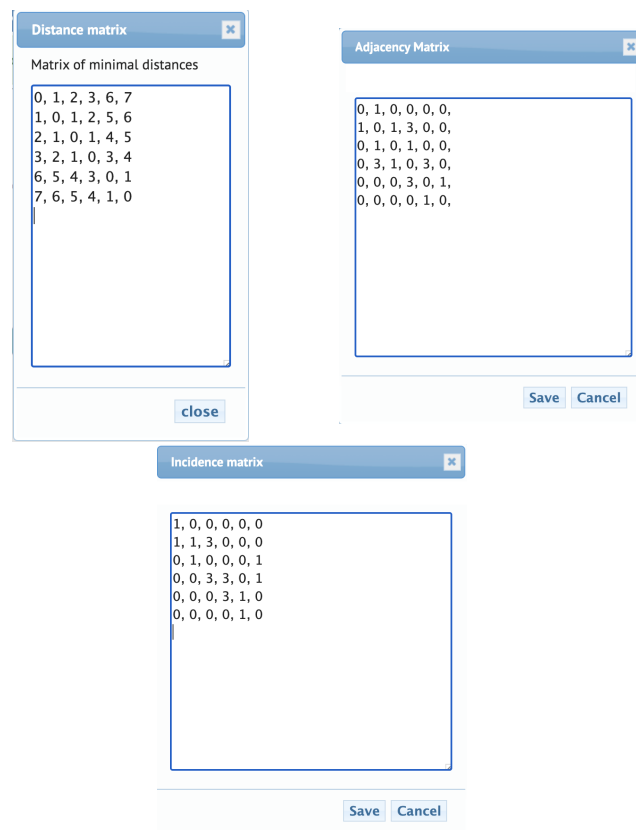


Figura 4.3: Distance Matrix, Adjacency Matrix and Incidence Matrix

4.3. Experiments

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.

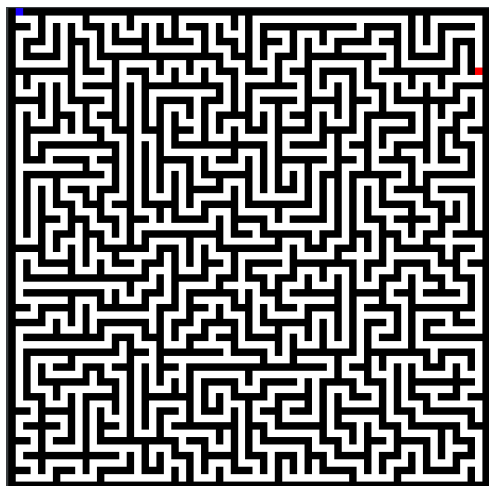


Figura 4.4: Maze

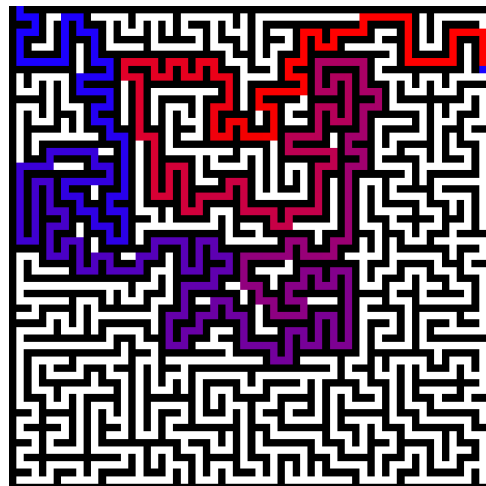
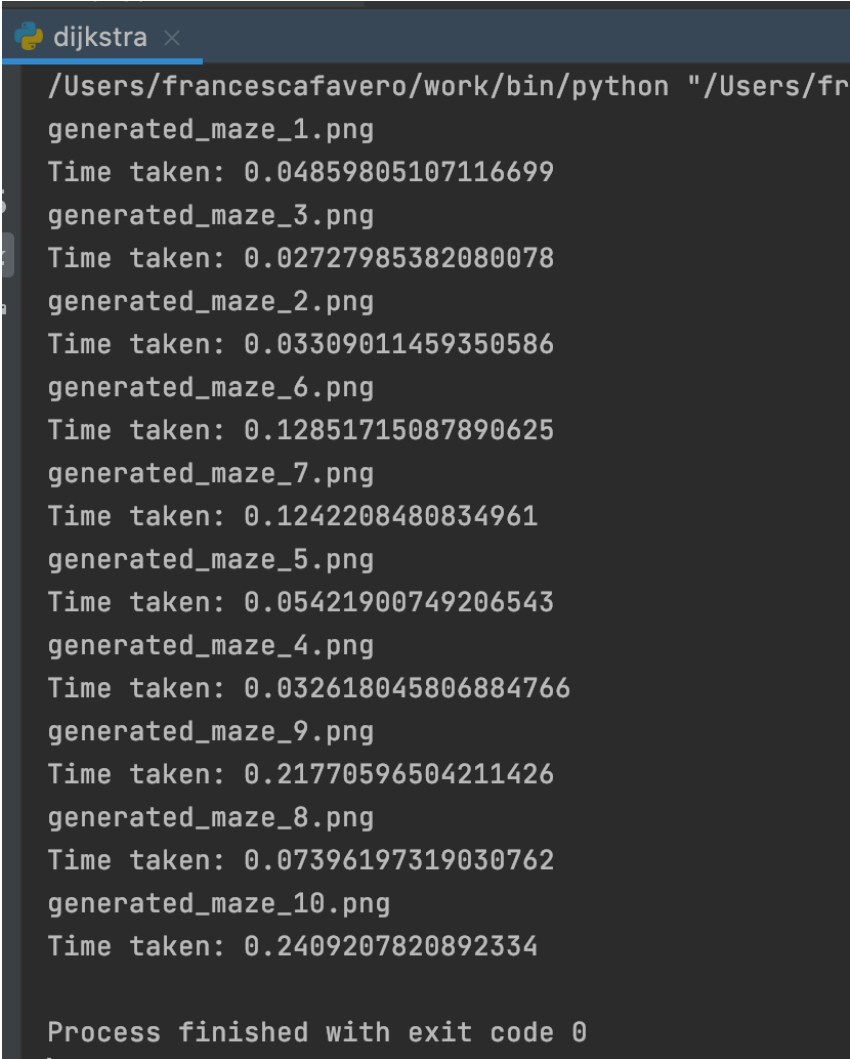


Figura 4.5: Maze solved

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.

4.3. Experiments



```
dijkstra x
/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
```

Figura 4.6: Time Taken by Dijkstra Algorithm

Conclusions

SUCAA