ESCI - UPF

Maze solving

Algorithms and Data Structures

Backtracking and Shortest-path approach

Mònica Torner - Xavier Crespo

Second year
22/02/2023

Contents

1 Introduction

You can find the submission files of this project at:
https://github.com/XavierUPF/ADS-SUBMISSIONS-BDBI/tree/main/submission_1

Mazes have been a source of fascination for humans for centuries, captivating people of all ages with their intricate designs and puzzling challenges. A maze is a complex network of paths or passages, often designed as a puzzle to be navigated through to reach a goal or find a way out. From ancient labyrinths etched into the floors of Greek temples to modern-day corn mazes, the art of creating and solving mazes has endured through the ages. Whether enjoyed as a recreational activity, used as a tool for problem-solving and strategy development or used as a torture game for some psychopath, mazes continue to be a good challenge for our brains.
But why use our brains filled with evolutive advantages if we can use millions of transistors that do the job?

For this, our aim in this submission is to solve a given 2D-array maze using backtracking and shortest-path approaches.
As said, the maze is given in a 2D-array where 0 indicates a wall or a non-valid move and 1 indicates a valid move.
The output of the program should be the path from start to end given in coordinates. Also, we can manage to represent it visually with *pyploy* from Matplotlib just to see the solution path easily.

2 The maze

We want to solve a "big enough" maze in order to see how good our algorithm is to solve it. For this, we will be solving Horta's Labyrinth (Laberint d'Horta).

This little labyrinth located at the north-east side of Barcelona has an available top image for us, so we can create the 2D-array to store the properties of the maze.

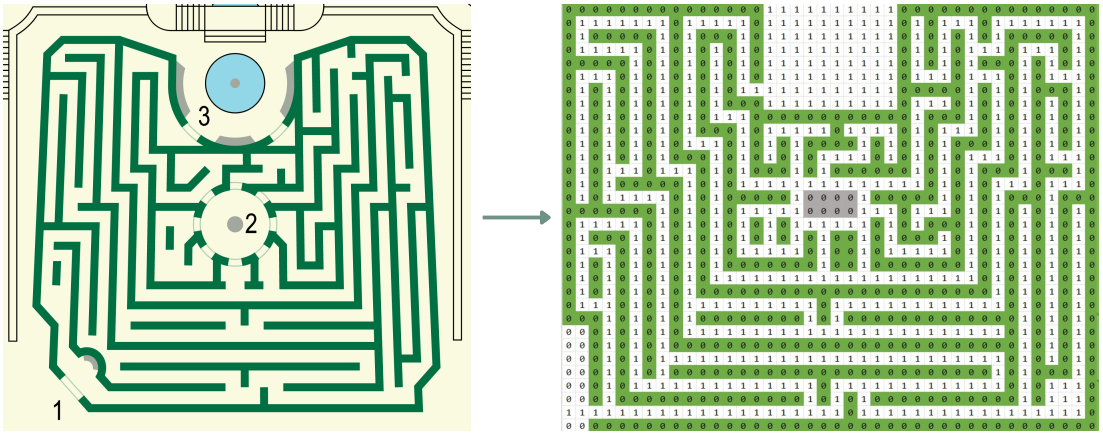

Figure 1: Horta's Labyrinth (left) and its rough matrix representation (right).
https://es.m.wikipedia.org/wiki/Archivo:Labyrinth_-_Parc_del_Laberint_d%E2%80%99Horta_-_Barcelona.svg

Taking 1 as the entrance (row 30, col 0) and 3 as the exit (row 6, col 15), the solution that comes to our mind is the following:

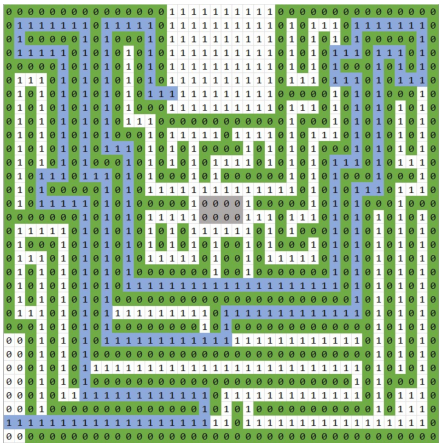

Figure 2: Our proposed path. It took us 20 seconds to solve it. Not bad for us, too slow for trained monkeys.

3 Backtracking

How our backtracking approach works:

1. `If the destination is reached, return True.`
2. `If the current cell is valid, add it to the path and mark it as visited.`
3. `Move down, right, up, and left, in that order.`
4. `If none of the above movements work, backtrack and return False.`

Thus, the solution provided from this algorithm is the following:

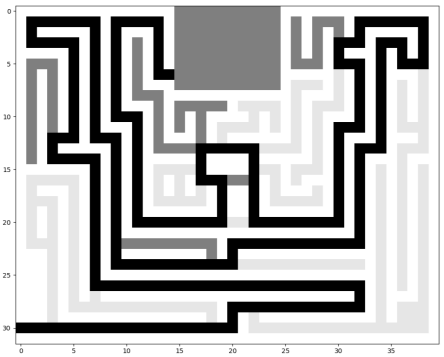

Figure 3: Light gray paths are the ones that the algorithm has tried, gray paths are the ones that has not tried and black path is the actual solution path obtained.
It took 0.0028676986694335938 seconds.

It is clearly noticeable that the path found is not the shortest one. But for the algorithm, as long as the path keeps working, it continues going forward.

4 Shortest path

How our shortest-path approach works:

```
1. Find the shortest path in the maze using breadth-first search.
2. Keep track of visited cells.
3. Create a queue of cells to visit.
4. While the queue is not empty:
    4.1. Pop the first cell.
    4.2. If we reached the destination:
        4.2.1. Add the destination to the path.
        4.2.2. Return the path.
    4.3. If we haven't visited the cell yet:
        4.3.1. Mark the cell as visited.
        4.3.2. Add the cell to the path.
        4.3.3. Move and check if it is a valid move.
5. If we can't reach the destination, return None.
```

Thus, the solution provided from this algorithm is the following:

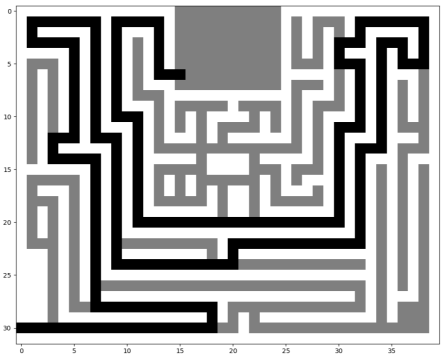

Figure 4: Black path is the actual solution path obtained.
It took 0.003805875778198242 seconds.

Breadth-first search (BFS) is a popular algorithm for traversing or searching a graph or tree data structure. It starts at a given node and explores all the nodes at the same level before moving to the next level. It is implemented using a queue data structure to keep track of the nodes to be visited.
The BFS function starts by creating an empty queue and adding the starting node to it. It also creates an empty set called visited to keep track of the nodes that have been visited.
The main loop of the function continues as long as the queue is not empty. It gets the next node from the queue using "pop(0)" method, which removes and returns the leftmost element from the queue. If the node has already been visited, it skips it and moves on to the next node.
If the node has not been visited, it marks it as visited by adding it to the visited set.
Finally, the function adds all the neighbouring nodes of the current node that have not been visited to the end of the queue using the append method. This ensures that the BFS algorithm visits all nodes at the same level before moving on to the next level.