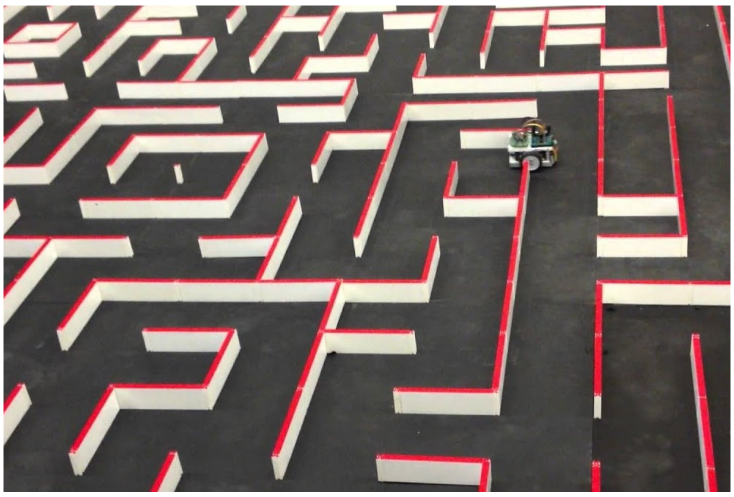
Computing Techniques
Escola Politècnica Superior, University of Lleida
Fall 2020
Jesús Ojeda

Assignment 1 – Solving a maze

1 Introduction

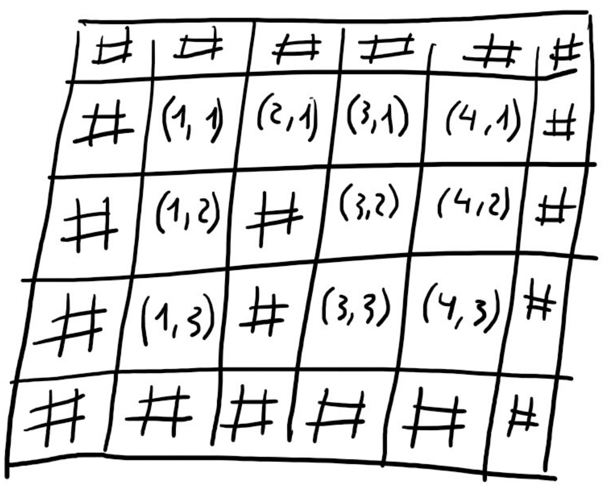
One of the primordial areas in artificial intelligence is defined in terms of search. We can establish any problem as a set of states, which are interconnected through the actions we can take at any moment. Then, starting from one such state, we can end at some other state.
One of the more direct applications of this frame of mind is that of path-finding. Picture a robot within a maze; beginning at a starting point within the maze, we want the robot to achieve the exit of the maze in a finite number of steps.



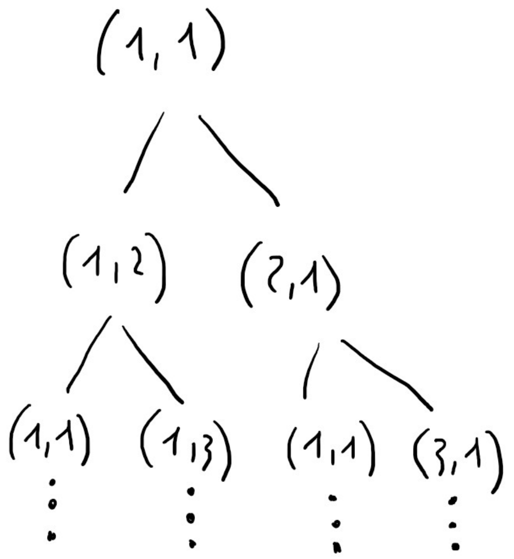
Two of the most commonly taught search algorithms are the graph-based **Depth-First Search (DFS)** and **Breadth-First Search (BFS)** (and you will see them in the Discrete Mathematics course). Those algorithms are precisely what we will be implementing in this assignment to evaluate recursive and iterative algorithms.

2 Problem

We want to compare two approaches, one recursive and one iterative, to solving the same problem: path-finding. As such, we will use a grid in which each position is a cell connected to its neighbors.



This defines a graph structure, in which each cell will be a vertex and will be connected with an edge to the cells it has direct contact with. To simplify this labyrinth world, we will consider that walls (cells with the # sign) are not valid neighbors cells, so from the point of view of a cell, its children/neighbors are just the empty cells available at north, south, east and west directions (if any).



Lets summarize how BFS and DFS work:
Breadth-First Search At any given point, we are in a position A in the graph. We want to visit all the neighboring positions $N_A = N_{A_1}, N_{A_2}, N_{A_3}...$ in direct contact with A. Then we will want to visit the direct neighbors of N_{A_1}, and then the direct neighbors of N_{A_2} and so on. You can think about the order in which we have to visit the neighbors as a queue.
In the image above, lets assume we start at $(1, 1)$, then we will visit $(1, 2)$ and then $(2, 1)$. Then we would visit the neighbors of $(1, 2)$ which are $(1, 1)$ and $(1, 3)$, and will continue visiting the neighbors of $(2, 1)$, so $(1, 1)$ and $(3, 1)$ ... and so on.

One optimization to be done is to store the positions we have already visited in order to avoid enqueuing them again.

Depth-First Search Once again, we are in a position A in the graph. If we define a branch for each neighbor N_{A_i}, we want to visit all possible descendants from branch N_{A_1} before getting to branch N_{A_2}, and then all its descendants before going to N_{A_3} and so on. You can think about visiting always left, until you reach the end of the tree, the start getting back and visit right, starting from the the lowest level.

In the example above, we start at $1, 1$. We will first visit the first neighbor $(1, 2)$. At this point we want to visit again the first neighbor, which is again $(1, 1)$. So you see, we have entered one infinite loop, here we will require the same optimization as before to avoid positions we have already visited.

Lets assume we have this optimization and just visit novel positions. We start at $(1, 1)$, we visit its first (novel) neighbor $(1, 2)$, then its first (novel) neighbor $(1, 3)$. $(1, 3)$ has no more novel neighbors, so we get up through the tree to $(1, 2)$, which also does not have more novel neighbors, up to $(1, 1)$, and then down to $(2, 1)$, then $(3, 1)$... and so on.

2.1 Code starting point

The provided code for this assignment is composed by two Python files and a txt one:

- **labyrinth.py** – Maze code. Defines the classes `Cell` and `Labyrinth`. A Labyrinth is composed of Cells, has a Start Cell and an End Cell and can be loaded from file. A Cell represents a position within the labyrinth and knows its neighboring Cells (through the method `getChildren`).

- **search.py** – Main code. This is the file to be called from the terminal. It defines two empty functions `DFS` and `BFS`. **You will have to write their code here**.

- **test_maze.txt** – A test maze in ASCII just with the characters {`#, _, S, E`}. A `#` character represents a wall at that position in the maze. A `_` character represents empty space. The `S` character represents the starting point for the maze and the `E` character represents the goal or exit of the maze. The file contains the following maze:

```
##############
#____________#
#____##__###_#
#__#_#S#__##_#
#__#____#__#E#
##############
```

3 Scores per task

- **3pt** Correctly implement the DFS recursive algorithm.
- **3pt** Correctly implement the BFS iterative algorithm.
- **1pt** Comment the code (a doc-string per function plus special comments trough the algorithms). Comments of the style:

  ```
  #we equal the variable 'a' to 4
  a = 4
  ```

  are irrelevant. I want to know **why** you write that code, **not what** the code does.
- **1pt** Test both algorithms on different mazes. Try different sizes (at least 10x10, 15x15 and 20x20) with different wall compositions and start and end nodes. Write a short report with these results and explain why you think one algorithm beats the other in terms of finding a shorter solution. **Hint:** BFS provides shorter solutions.
- **1pt** Both algorithms search a path in a maze. Which are the pre-condition and post-condition of them? Are they the same?
- **1pt** In terms of the following values:
  - **b** → branching factor: maximum number of children a node can have
  - **d** → depth: length of the shortest path between the start and end nodes
  - **m** → maximum length: length of the maximum path that can be achieved

  What is the $\mathcal{O}$ cost in terms of visited nodes of finding the solution for the BFS algorithm? What is the $\mathcal{O}$ cost for the DFS algorithm? (Answer these questions and argue about them in the report from the previous task)

4 Delivery

When? We will work through this assignment in the PRALAB sessions and the delivery date is due Monday, 9th of November of 2020, at 23:55. You have this info in the Virtual Campus.

What?

- The **search.py** Python file, thoroughly commented on your implementation. It is important that you maintain the `BFS` and `DFS` function signatures as they are (although you can add whatever additional code you need in that file). It will be tested against **Python 3.7 or greater**, be sure to use an appropriate version!
- The report (write it in whatever but **export it to PDF**) with the results of the algorithms on different mazes and the answer to pre-/post-conditions and the $\mathcal{O}$ questions.

Additionally: This assignment can done individually or in pairs. It accounts for 25% of the final score of the course. Be extra careful **including the names of the participants as a comment in the first line of the uploaded file, as well as the PDF report**.