Exercise 1: Part 5

Maze

Python Programming Bootcamp by Dr Rohitash Chandra
UNSW, 2021

Introduction

The goal of this assignment is to help you practice   programming – with arrays and functions.

Description

A robot is asked to navigate a maze. It is placed at a certain position (the *starting* position) in the maze and is asked to try to reach another position (the *goal* position). Positions in the maze will either be open or blocked with an obstacle. Positions are identified by (x,y) coordinates.

At any given moment, the robot can only move 1 step in one of 4 directions. Valid moves are:

- Go North: (x,y) -> (x,y-1)
- Go East: (x,y) -> (x+1,y)
- Go South: (x,y) -> (x,y+1)
- Go West: (x,y) -> (x-1,y)

Note that positions are specified in zero-based coordinates (i.e., 0...size-1, where *size* is the size of the maze in the corresponding dimension).

The robot can only move to positions without obstacles and must stay within the maze.

The robot should search for a path from the starting position to the goal position (a *solution path*) until it finds one or until it exhausts all possibilities. In addition, it should mark the path it finds (if any) in the maze.

You will implement a simple algorithm for findıng a path in a maze that connects the entrance to the exit. From any given cell in the maze, the algorithm will try out all 4 possible direction to determine the path. We assume that such a path does exist.

Your program will implement a recursive backtracker. This will find a solution, but it will not necessarily find the shortest solution. The idea of this algorithm is very simple: If you are at a wall (or a cell you have already visited), return failure; otherwise, if you are at the finish, return success, else recursively try moving in the four directions. Mark a cell when you try a new direction, and erase the mark when you return failure, and a single solution will be marked out when you hit success. When backtracking, it is best to mark the space with a special visited value, so you do not visit it again from a different direction. This is method is called a depth first search. This method will always find a solution if one exists, but it will not necessarily be the shortest solution.

Programming Directives

We'll solve the problem of finding and marking a solution path using *recursion*. Remember that a recursive algorithm has at least 2 parts:

5. Base case(s) that determine when to stop.

6. Recursive part(s) that call the same algorithm (i.e., itself) to assist in solving the problem.

Because our algorithm must be *recursive*, we need to view the problem in terms of similar *subproblems*. In this case, that means we need to "find a path" in terms of "finding paths."

Let's start by assuming there is already some algorithm that finds a path from some point in a maze to the goal, call it FIND-PATH(x, y).

Also suppose that we got from the start to position x=1, y=2 in the maze (by some method):

To find a path from position x=1, y=2 to the goal, we can just ask FIND-PATH to try to find a path from the North, East, South, and West of x=1, y=2:

- FIND-PATH(x=1, y=1)
- FIND-PATH(x=2, y=2)
- FIND-PATH(x=1, y=3)
- FIND-PATH(x=0, y=2)

Generalizing this, we can call FIND-PATH recursively to move from any location in the maze to adjacent locations. In this way, we move through the maze.

It's not enough to know how to use FIND-PATH recursively to advance through the maze. We also need to determine when FIND-PATH must stop.

One such *base case* is to stop when it *reaches the goal*.

The other base cases have to do with moving to invalid positions. For example, we have mentioned how to search North of the current position, but disregarded whether that North position is legal. In order words, we must ask:

1. *Is the position in the maze* (... or did we just go outside its bounds)?
2. *Is the position open* (... or is it blocked with an obstacle)?

Now, to our base cases and recursive parts, we must add some steps to mark positions we are trying, and to unmark positions that we tried, but from which we failed to reach the goal:

```
FIND-PATH(x, y)

    1.      if (x,y outside maze) return false
    2.      if (x,y is goal) return true
    3.      if (x,y not open) return false
    4.      mark x,y as part of solution path
    5.      if (FIND-PATH(North of x,y) == true) return true
    6.      if (FIND-PATH(East of x,y) == true) return true
    7.      if (FIND-PATH(South of x,y) == true) return true
    8.      if (FIND-PATH(West of x,y) == true) return true
    9.      unmark x,y as part of solution path
    10. return false
```

All these steps together complete a basic algorithm that finds and marks a path to the goal (if any exists) and tells us whether a path was found or not (i.e., returns true or false). This is just one such algorithm--other variations are possible.

To use FIND-PATH to find and mark a path from the start to the goal with our given representation of mazes, we just need to:

1. Locate the start position (startx, starty)
2. Call FIND-PATH(startx, starty)

Your program will read the dimensions of the maze as well as the location of all obstacles from a file. Your program must check that the (x, y)-coordinates of obstacles are valid. The format of the input file will be as follows:

```
<length> <width>
<x1> <y1>
<x2> <y2>
<x3> <y3>
   .
   .
   .
```

Grading

Your program will be graded as follows:

Grading Point	**Marks**
Initialize maze	5
Implementation of recursion	30
Marking of visited cells	10
Finding the path	10
Graphical display - Animation	40
Code quality (e.g., variable names, formulation of selection statements and loops, etc)	5

Resources:

1. https://www.programiz.com/python-programming/recursion
2. https://www.laurentluce.com/posts/solving-mazes-using-python-simple-recursivity-and-a-search/

Acknowledgment

The assignment is adapted from initial exercise designed by Prof. Christian Omlin