Path Planning

Path planning is required in many situations, including satnav software and autonomous robot control. For example, the item-pickers employed in Amazon's warehouses follow instructions that specify their routing through the warehouse as they assemble each order, with the route determined in advance so as to minimize their walking time (or riding time, the warehouses are big). Your task in this project is to develop a program that computes a shortest routing through a set of obstacles structured as a maze.

Stage 1 – Reading and Printing (marks up to 6/20)

The input to your program will consist of a character-based description of a maze, to be read from `stdin` using input redirection, in the same way as was used in Assignment 1. (Please do *not* make use of `fopen()` and `fread()` from Chapter 11.) Mazes are composed of a rectangular array of two characters, `'#'` to indicate no-go cells that may not be used by the robot (the maze walls), and `'.'` to indicate passable cells that the robot may move into. For example, the description in `test0.txt` (available on the LMS) is

```
#.#######
#.......#
####.####
#....#..#
#.####.##
```

and describes a maze that has one gap in the top row (for entry to the maze), two gaps in the bottom row (for exiting the maze), and is configured as a grid of five rows and nine columns, with each row of the input corresponding to a one row maze cells. The robot is assumed to always enter the maze at one of the available gaps in the top row, and must always exit via one of the available gaps in the bottom row. If there is no path from any entry gap to any of the exit gaps then the maze has no solution.

You should assume throughout that input files will be "correct", and will contain nothing but `'#'`, `'.'`, and `'\n'` characters laid out correctly in a rectangular grid that will never be bigger than 100×100 cells; will always have at least one entry gap in the top row, and will always have at least one exit gap in the final row. You may *not* assume that there will be a legal path from any particular entry gap to any particular exit gap, and must correctly handle cases in which the maze has no valid solution.

In this first stage you should develop a program that has these elements:

- a type definition for a suitable `struct` for recording a maze as a two dimensional array, plus some auxiliary variables, and in which each cell in the maze is represented as a lower-level `struct` that contains the state variables associated with each cell;
- a function that reads the input maze, via a suitable pointer-to-`struct` argument; and