Department of Computing and Information Systems

COMP20005 Engineering Computation
Semester 1, 2016
Assignment 2

Learning Outcomes

In this project you will demonstrate your understanding of structures and arrays of structures, and will develop a computational solution for a non-trivial problem. You are also expected to make extensive use of functions; and to demonstrate that you have adopted a clear and elegant programming style. You will find it difficult to create a working solution unless you plan your program carefully in advance, and develop it incrementally.

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

- a function for printing a maze out via a suitable pointer-to-`struct` argument, with each cell's character doubled to make the maze easier to view on the screen.

The required output for this stage for `test0.txt` is:

```
Stage 1
=======
maze has 5 rows and 9 columns
##..##############
##..............##
########..########
##........##....##
##..########..####
```

In this stage your elemental `struct` for each cell might only contain one variable, the type of that cell.

Stage 2 – Determining Reachable Regions (marks up to 12/20)

Each legal move of the robot takes it either one step vertically or one step horizontally, from one viable cell to an immediately adjacent viable cell. A viable cell is *reachable* if the robot can reach it starting at any of the gaps in the top row of the maze, and then following (any number of) legal moves.

Develop an algorithm for determining (and storing with your structure) a flag that records the reachability of every viable cell in the maze, including any exit gaps in the last row. The output of this stage is again a map of the maze, but with reachable cells shown as doubled `'+'` characters, and non-reachable cells as doubled `'-'` characters. If any of the exit gaps is reachable, then the maze as a whole can be reported as having a solution. The required output from this stage for `test0.txt` is

```
Stage 2
=======
maze has a solution
##++##############
##++++++++++++++##
########++########
##++++++++##----##
##++########--####
```

Note the region in this maze that is non-reachable, including one of the exit gaps. (The alternative message to be used, if all of the exit gaps are non-reachable, is `"maze does not have a solution"`.) Further examples showing the full output that is required are provided on the LMS, and you should study them carefully so that you understand the details of what is required in this stage.

Note that the output from this stage is *in addition* to the output of Stage 1.

Stage 3 – Calculating Costs (marks up to 16/20)

Now add a further variable to the `struct` that represents each cell of the maze, and for each reachable cell, compute into that variable the minimum cost of any path from any entry gap in the top row through until that cell, counting one unit of cost for each cell that is traveled through, and with a cost of zero assigned at each top-row gap. The cost of every second reachable cell should be printed using two digits; other cells should be printed as before. If the cost of a reachable cell is greater than 99, then only the last two digits of the number should be printed. The required output from this stage for `test0.txt` is

```
Stage 3
=======
maze has a solution with cost 10
##00##############
##++02++04++06++##
########++########
##++08++06##----##
##10########--####
```

Further examples showing the full output that is required are provided on the LMS, and you should study them carefully so that you understand the details of what is required in this stage. Note that the output from this stage is *in addition* to the output of Stages 1 and 2.

Stage 4 – Plotting a Path (marks up to 20/20)

Now add further state information to the struct for each cell so that the exact path implied by one solution is drawn, and none of the other cell costs are shown. In this stage, reachable cells not on the final path should be shown as doubled blanks. The required output from this stage for `test0.txt` is

```
Stage 4
=======
maze solution
##00##############
##..02..04      ##
########..########
##..08..06##----##
##10########--####
```

In cases where there are two or more exit gaps that have the same minimum distance from a start gap, a path to the leftmost of them should be plotted. Note that the output from this stage is *in addition* to the output of Stages 1, 2 and 3, assuming that a solution exists. In cases where there is no solution nothing should be printed except for the Stage 1, 2 and 3 output. Further examples showing the full output that is required are provided on the LMS, and you should study them carefully so that you understand the details of what is required in this stage.

A Note on Algorithms

You are free to adopt any approach that you wish to labeling cells and computing path costs, but you do need to be systematic, and develop a mechanism that computes the correct answers. Be sure to provide comments in your programs to help the markers understand the particular mechanism you have used.

One possible approach is to cycle through the maze, examining every cell in order. Then, if that cell has been labeled with a path cost, use that cell to try and also label its neighbors with a path cost that is one greater. Path costs of labeled cells should only ever decrease, once a cell is first labeled. If a complete run through of every cell results in no changes to the path cost of reaching any cell in the maze, then a final set of path costs must have emerged. On the other hand, if any cell got its cost reduced in the last run through, start another pass through and allow that change to propagate further if it needs to. Begin by assigning a path cost of zero to the gaps in the top row of the maze.

This isn't a very *efficient* algorithm, but it will be fast enough for the scale of maze being considered here. There are – of course! – more efficient algorithms than this that can be applied when there are millions or even billions of cells involved (for example, when the maze is three-dimensional). Come back and enrol in comp20003 in second semester if you want to know more.

The boring stuff...

This project is worth 20% of your final mark. A rubric explaining the marking expectations will be provided on the LMS.

You need to submit your program for assessment; detailed instructions on how to do that will be posted on the LMS once submissions are opened. Submission will *not* be done via the LMS; instead you will need to log in to a Unix server and submit your files to a software system known as `submit`. You can (and should) use `submit` **both early and often** – to get used to the way it works, and also to check that your program compiles correctly on our test system, which has some different characteristics to the lab machines. *Failure to follow this simple advice is highly likely to result in tears.* Only the last submission that you make before the deadline will be marked.

You may discuss your work during your workshop, and with others in the class, but what gets typed into your program must be individual work, not copied from anyone else. So, do **not** give hard copy or soft copy of your work to anyone else; do **not** "lend" your "Uni backup" memory stick to others for any reason at all; and do **not** ask others to give you their programs "just so that I can take a look and get some ideas, I won't copy, honest". The best way to help your friends in this regard is to say a very firm "**no**" when they ask for a copy of, or to see, your program, pointing out that your "**no**", and their acceptance of that decision, is the only thing that will preserve your friendship. *A sophisticated program that undertakes deep structural analysis of C code identifying regions of similarity will be run over all submissions in "compare every pair" mode. Students whose programs are so identified will be referred to the Student Center for possible disciplinary action without further warning. This message* ***is*** *the warning.* See `https://academichonesty.unimelb.edu.au` for more information.

Deadline: Programs not submitted by **10:00am on Monday 23 May** will lose penalty marks at the rate of two marks per day or part day late. Students seeking extensions for medical or other "outside my control" reasons should email `ammoffat@unimelb.edu.au` as soon as possible after those circumstances arise. If you attend a GP or other health care professional as a result of illness, be sure to take a Health Professional Report form with you (get it from the Special Consideration section of the Student Portal), you will need this form to be filled out if your illness develops in to something that later requires a Special Consideration application to be lodged. You should scan the HPR form and send it in connection with any non-Special Consideration assignment extension requests.

Marks and a sample solution will be available on the LMS before Tuesday 7 June.

And remember, programming is fun!