School of Computing and Information Systems

comp20005 Intro. to Numeric Computation in C
Semester 1, 2022
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 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.

Robots!

The world is increasingly automated. Robot vehicles undertake deliveries around factories, airports, and shopping centers; robot vacuum cleaners clean our floors; and robotic agents greet guests arriving in hotels and convention centers.

To be useful, any self-propelled robot must always know its location, and be able to sense the locations of obstacles in its environment. Once it has that information it must also be able to undertake *path planning*, to navigate the environment and arrive at its next required destination. In this programming project we will assume that the initial location of a robot, and of a set of obstacles, is given; and that your program must plan a path for the robot to a destination point.

To keep things simple we will work within a synthetic "*grid world*" completely defined by integer coordinates, with robots that can move left, right, up, down, or diagonally one cell at a time. The location of the robot at any given moment is specified by its (x, y) location as two integer coordinates. For example, in Figure 1 the robot is at location $(8, 14)$. Obstacles are defined as rectangular regions specified as four integer values

$$x_{min}, x_{max}, y_{min}, y_{max} .$$

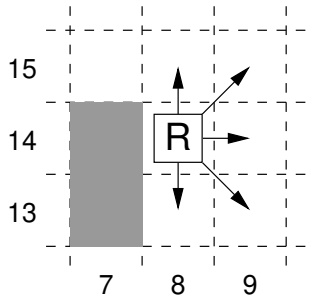

Figure 1: Robot and obstacle.

In Figure 1 the shaded region represents an obstacle described as $(7, 7, 13, 14)$. The robot can move to any of five neighboring cells, but cannot enter the space blocked by the obstacle (the obstacle might be an actual wall or table, or it might just be a region reserved for humans, such as a queue or walkway), and nor can it "clip" the corner of the obstacle to get to location $(7, 15)$. Two moves are required to get to $(7, 15)$, described by $(8, 14) \rightarrow (8, 15) \rightarrow (7, 15)$. Obstacles can touch each other and can overlap – that's how more complex shapes can be assembled.

Stage 1 – Control of Reading and Printing (marks up to 8/20)

Your first program should read an input file from `stdin` describing the robot's world. The input values will all be integers, two on the first line, and then four on every line after that. The two integers describe the rectangular "size" of the world, and then the second block of lines describe the obstacles. For example, the input file `test0.txt`

```
7 3
4 4 0 1
5 5 1 2
3 3 2 2
```

describes a world that is 7×3 cells wide$\times$high with three obstacles in it – this is the world that is partly shown in Figure 2. You may assume that robot worlds will always be at most 100 cells wide and at most 100 cells high, with cells described by coordinate pairs $[0 \ldots 99] \times [0 \ldots 99]$. (Yes, we are programmers, we will count from zero.) The first input line with the world dimensions will *always* appear, and then there will be between zero and 100 lines each describing an obstacle. In this stage your program should read all of the available data, and then print it out again. For the file `test0.txt` the required output is:

```
S1, world is  7 cells wide x  3 cells high
S1, world contains 3 obstacles
S1, obstacle  0 covers [ 4, 4] x [ 0, 1]
S1, obstacle  1 covers [ 5, 5] x [ 1, 2]
S1, obstacle  2 covers [ 3, 3] x [ 2, 2]
```

Other examples of the required output are linked from the Assignment 2 page on the LMS. To obtain full marks you need to *exactly* reproduce the required output lines. You may assume that the input provided to your program will always be sensible and correct, and you do not need to perform any data validation.

You can base your program on your Assignment 1 solution, if you wish. But note, for this assignment you are required to define and use `structs` to manage the input and computed data.

Stage 2 – Finding The Zones (marks up to 16/20)

The robot has a *home base* location at which it plugs itself in and recharges its battery. For simplicity, let's take that to be the location $(0, 0)$, as shown in Figure 2. You may assume that no obstacle will ever be placed at $(0, 0)$. Relative to the home base, every cell in the grid then falls into one of three categories: those that contain obstacles, shown in grey in Figure 2; those that are *reachable* from the home base, because the robot can get to them from $(0, 0)$ by moving through other reachable cells, shown as blanks in Figure 2; and those that are *unreachable* because they are (for example) surrounded by obstacles, marked by "*a*" and "*b*" in Figure 2. There might be several zones of unreachable cells, none of which can reach each other because of the obstacles. Or, it might be that every non-obstacle cell is reachable from the home base.

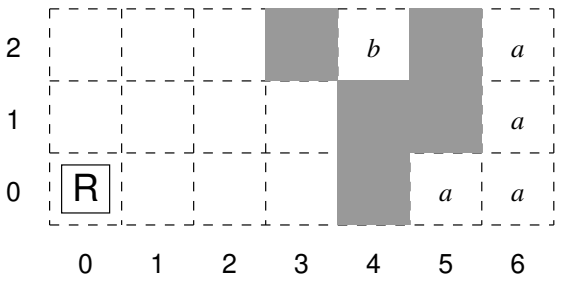

Figure 2: Reachability zones for `test0.txt`.

In this stage your program must compute, for every cell in the robot world, whether it is reachable or not; and, if not, which non-reachability zone it falls into. For the file `test0.txt` (Figure 2):

```
S2,   11 of   21 cells are reachable
S2,    4 of   21 cells are in unreachable zone a
S2,    1 of   21 cells are in unreachable zone b
S2,    5 of   21 cells are obstacles
```

The zones (for example, zone a in Figure 2) are groups of cells within which a robot could get from any cell to any other cell if it was already in that zone, but to which the robot *cannot* get to at all when starting from the home base. The labeling of the zones ("a", "b", and so on) is based on the *first* cell for that zone that you encounter when moving left to right across each row, processing rows from the bottom of the grid to the top (that is, $(5, 1)$ is encountered before $(4, 2)$). You may assume that there will never be more than 26 different reachability zones, and that labeling them with lower-case letters will always be possible. See the LMS page for further examples of input and required output.

Hint: start with $(0, 0)$ as a *seed* cell that is marked as "reachable". Then iterate over the whole grid, all rows and columns, and for each cell that is currently marked as reachable, mark all of its neighbors the same way, keeping track of any changes made. Once you get to the end of the whole grid, if you did make changes, go back to the beginning of the grid and go right through again. Eventually, when you

go all the way through all the cells without making any more changes, all the reachable cells will have been located. Then look for any cells have still not been labeled, and if you find any, use the "lowest, leftest" such cell as a new seed from which to label zone a using exactly the same process. Keep going until every non-obstacle cell is either in an obstacle, or is labeled as reachable, or is labeled with a zone. There will be lots of loops, so make sure you use functions carefully so that the loop nesting depth stays under control.

Stage 3 – Deciding Routes (marks up to 20/20)

Every cell that is reachable from the home base has a corresponding *distance*, defined by assuming that rectilinear moves (left, right, up, down) have a cost of 2, and that diagonal moves have a cost of 3 (approximately $2\sqrt{2}$). Each cell's cost is given by the *best* way of getting to that cell from the home base. For example, in Figure 3 cell $(2, 2)$ has a cost of 6 because the cheapest way of getting there is to use two diagonal moves. And a reminder: diagonal moves are not possible across the corners of obstacles, which is why Figure 3 shows cell $(5, 1)$ with a cost of 14 – the path to it must include $(3, 2) \rightarrow (4, 2) \rightarrow (5, 2) \rightarrow (5, 1)$.

To compute path costs you need to carry out a similar nested-loop computation as in Stage 2, first labeling the home base with a "cost" of 0 and every other cell with a very high cost, and then sweeping through every cell checking each its neighbors to see if their current "best cost so far" can be reduced. After each sweep right through all of the grid, if any changes were made (that is, cells that had their cost decreased), go back to the beginning and do it all over again. Keep on doing that until no more changes of cost take place, at which point you will have reached a stable configuration.

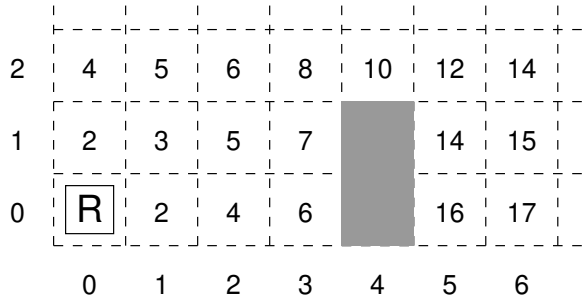

Figure 3: Path cost calculation.

The output of this stage is a map of the robot world, with each cell in the even numbered rows plotted as a single character, and the odd-numbered rows not plotted at all. (Typical computer fonts have each letter approximately twice as tall as it is wide, and we want a square robot world to plot as an approximately square map on the screen.) Each cell in the even numbered rows in the robot world is plotted as a character according to its cell type:

- with `'R'` if it is the home base;
- with `'|'` if it is an obstacle (the "vertical pipe" character);
- with `'a'`, `'b'` and so on if it is unreachable, with the letter noting its zone number (see Figure 2);
- with a single digit derived from its path cost if it is reachable from the home base.

The rule for the reachable cells involves the last two digits of its decimal value. If the last digit is between 0 and 3 inclusive, then the second to last digit is printed; and if the last digit is between 4 and 9 inclusive, then a `'.'` is printed. For example, $31 \rightarrow$ `'3'`, and $54 \rightarrow$ `'.'`, and $89 \rightarrow$ `'.'` and $172 \rightarrow$ `'7'` and $240 \rightarrow$ `'4'`. This rule will lead to output with bands of approximately equal cost becoming visible. Here are *all* of the map lines for `test0.txt`, you can compare this with Figure 2.

```
S3,    | ...|b|a
S3,    | 00..||a          <-- your program does not output this line!
S3,  0 + R0..|aa
```

Your final program should actually only print the first and last of these three lines; and the middle line is included here purely to help you understand the structure of the map. The y-axis numbers should be printed for cell indexes that are multiples of ten. The LMS Assignment 2 page gives several more examples that show in more detail what you are to generate, and once you look at them will appreciate why we are thinking of these as "maps".

Modifications to the Specification

There are bound to be areas where this specification needs clarification or correction, and you should refer to the "Assignment 2" LMS Discussion regularly and check for possible updates to these instructions.

The Boring Stuff...

This project is worth **20% of your final mark**, and is due at **6:00pm on Friday 20 May**.

Submissions that are made after that deadline will incur 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 service as a result of illness, be sure to take a Health Professional Report (HPR) 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 with any non-Special Consideration assignment extension requests.

Submission: Your `.c` file must be uploaded **via the LMS "Assignment" page**. *Don't forget to include, sign, and date the Authorship Declaration that is required at the top of your program.*

Testing: You can also carry out pre-submission testing via a system known as `submit`, available at `http://submit-web.eng.unimelb.edu.au`. The `submit` system uses the same computer/compiler that will be employed for post-submission testing (a Unix computer called `dimefox` which is very unforgiving of uninitialized variables and out-of-bounds array accesses). Note that `submit` may become congested and non-responsive in the final day or hours before the deadline. *Even though it is called "submit", this testing service does* **not** *register your assignment.* **You must upload your final program to the LMS via the "Assignment" page to actually submit it**.

Marking Rubric: A rubric explaining the marking expectations is linked from the assignment's LMS page, and you should study that rubric very closely. Feedback, marks, and a sample solution will be made available approximately two weeks after submissions close.

Academic Honesty: 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, and not developed jointly with 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**" if they ask to see your program, pointing out that your "**no**", and their acceptance of that decision, are the only way to preserve your friendship. See `https://academicintegrity.unimelb.edu.au` for more information. Note also that solicitation of solutions via posts to "tutoring" sites or online forums, whether or not there is payment involved, and whether or or not you actually employ any solutions that may result, is also serious misconduct. In the past students have had their enrolment terminated for such behavior.

The LMS page links to a program skeleton that includes an Authorship Declaration that you must "sign" and date and include at the top of your submitted program. Marks will be deducted (see the rubric linked from the LMS page) if you do not include the declaration, or do not sign it, or do not comply with its expectations. A sophisticated program that undertakes deep structural analysis of C code identifying regions of similarity will be run over all submissions. Students whose programs are identified as containing significant overlaps will have substantial mark penalties applied, or be referred to the Student Center for possible disciplinary action, without further warning.

And remember, programming is fun!