School of Computing and Information Systems

comp10002 Foundations of Algorithms
Semester 2, 2019
Assignment 2

Learning Outcomes

In this project, you will demonstrate your understanding of dynamic memory and linked data structures (Chapter 10), and extend your skills in terms of program design, testing, and debugging. You will also learn about route planning, and implement a simple route re-planning mechanism in preparation for the more principled approaches that are introduced in comp20007 Design of Algorithms.

Grid-Based Route (Re-)Planning

Route planning is used in the navigation of autonomous agents, e.g., self-driving vehicles, robots, and humans (think of mapping services). Grid-based route planning studies the problem of constructing a route from an initial cell to a destination cell in a two-dimensional space subdivided into grid cells that are either blocked or empty. Figure 1a shows an example grid of 10 rows and 10 columns, the initial cell at row 0 and column 0 denoted by `I` (yes, we count from zero) and the destination (goal) cell at row 9 and column 9 denoted by `G`. The green arrows show the planned moves to get from `I` to `G` that omit the blocks shown as orange squares.

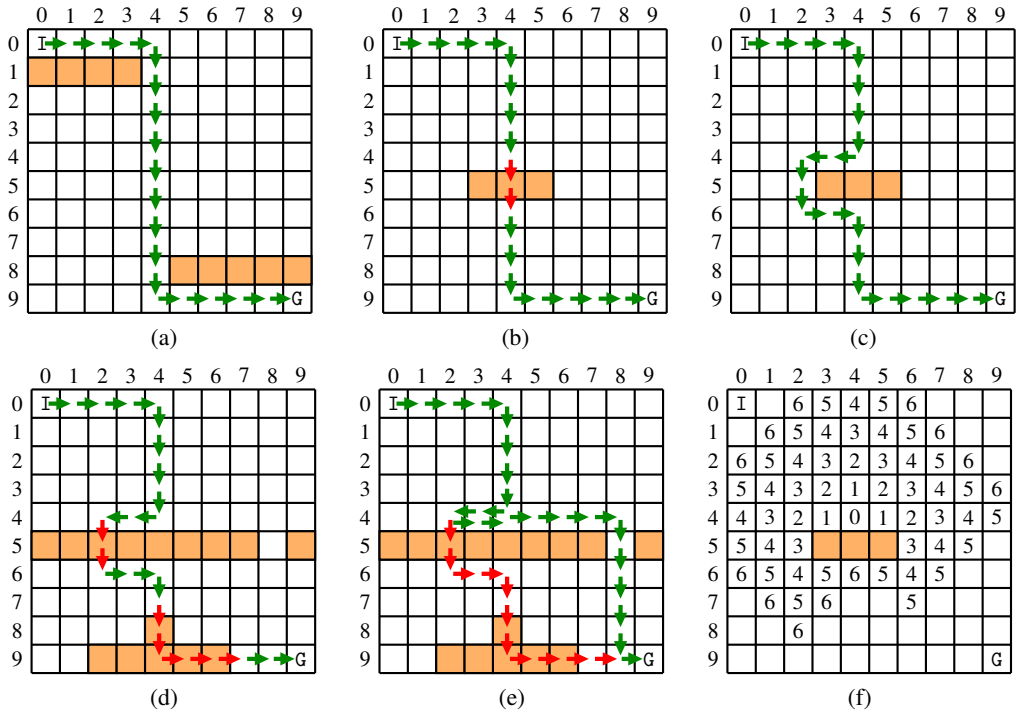

Figure 1: Grids with blocks and routes.

As the environment changes, i.e., blocks get removed or placed at some new cells, the planned route may become invalid. Figure 1b shows the grid from Figure 1a with a new configuration of blocks. The original route from Figure 1a is invalid in Figure 1b, as it visits blocked cells, see the red arrows in the figure. If an agent, e.g., a robot, takes the planned route from Figure 1a to get from cell `I` to cell `G` in the grid in Figure 1b, it will get blocked. However, at that point, it can request the new locations of all blocks from the external sensors and re-plan to move around the obstacles and get back on the original route, as shown in Figure 1c. Figure 1d shows a subsequent configuration of blocks that breaks the re-planned route from Figure 1c in two places. An agent that attempts to follow that route will get blocked for the first time at the cell at row 4 and column 2. Using the available data from the sensors, it may then decide to re-plan the route to take the green arrows in Figure 1e. Note that the agent may need to revisit some cells. The red path in Figure 1e is now useless and should be forgotten.

Input Data

Your program should read input from `stdin`. The first line of the input encodes the dimensions of the grid, for example `10x20` specifies that the grid has 10 rows and 20 columns. The second line of the input encodes the initial cell of the agent in the grid, while the third line encodes the goal cell. A cell is encoded using the format `[r,c]`, where `r` and `c` are numbers that stand for, respectively, the row and column of the cell. Subsequent input lines specify positions of blocks in the grid, one block cell per line. An input line with a single character `$` denotes the end of block lines. The input lines that follow encode a route of the agent in the grid. A route is encoded by alternating cells and `->`, for example `[0,0]->[0,1]->[0,2]` encodes the route that starts at cell `[0,0]`, and then proceeds via cell `[0,1]` to cell `[0,2]`; no blanks or tabs are used in encodings of routes, while a single newline character may follow `->`. For example, the following file `test0.txt` encodes the grid and route from Figure 1a (the numbers in italics show line numbers and are not part of the input file).

```
1  10x10
2  [0,0]
3  [9,9]
4  [1,0]
5  [1,1]
6  [1,2]
7  [1,3]
8  [8,9]
9  [8,8]
10 [8,7]
11 [8,6]
12 [8,5]
13 $
14 [0,0]->[0,1]->
15 [0,2]->[0,3]->
16 [0,4]->[1,4]->
17 [2,4]->[3,4]->
18 [4,4]->[5,4]->
19 [6,4]->[7,4]->
20 [8,4]->[9,4]->
21 [9,5]->[9,6]->
22 [9,7]->[9,8]->
23 [9,9]
24
```

Stage 0 – Reading and Analyzing Input Data (8/15 marks)

The first version of your program should read the input from `stdin` and print out some basic information so that you can be sure you have read the inputs correctly. Your program should also ensure that the supplied route is valid, i.e., it starts in the initial cell, ends in the goal cell, consists of legal moves (one cell moves and no diagonal moves) and does not visit a cell with a block. The required output from this stage is the following summary (generated for the `test0.txt` example input file shown above):

```
mac: ass2-soln < test0.txt
==STAGE 0======================================
The grid has 10 rows and 10 columns.
The grid has 9 block(s).
The initial cell in the grid is [0,0].
The goal cell in the grid is [9,9].
The proposed route in the grid is:
[0,0]->[0,1]->[0,2]->[0,3]->[0,4]->
[1,4]->[2,4]->[3,4]->[4,4]->[5,4]->
[6,4]->[7,4]->[8,4]->[9,4]->[9,5]->
[9,6]->[9,7]->[9,8]->[9,9].
The route is valid!
```

The last line in the above summary reports the *status* of the route that can take one of these five values:

Status 1: `Initial cell in the route is wrong!`
Status 2: `Goal cell in the route is wrong!`
Status 3: `There is an illegal move in this route!`
Status 4: `There is a block on this route!`
Status 5: `The route is valid!`

Status 1 is printed if the first cell in the route is different from the initial cell supplied at line 2 of the input. Status 2 is printed if the last cell in the route is different from the goal cell given at line 3 of the input. Status 3 is printed if the route contains a move that traverses more than one cell. Status 4 reports the presence of a block at one of the cells visited in the route. Otherwise, Status 5 should be printed. The status checks should be done in the order shown, and if a certain status holds, it should be reported and no further checks carried out. When printing a route, print no more than five cells per line separated by `->`. Do not use blanks or tabs when printing a route. A newline character may follow `->`. Refer to sample output files for further examples of printed routes. Note that the outputs generated by your program should be exactly the same as the sample outputs for the corresponding inputs. You should not assume the maximal allowed sizes for the input grid and route and, thus, use `malloc` to store the grid in a two dimensional (2D) array, and *linked lists* to store routes.

Stage 1 – Drawing and Replanning (13/15 marks)

Extend your program to visualize the input grid and route, attempt repair of a broken route, and visualize the repaired route. All of the Stage 0 output should be retained, then your Stage 1 output should start with this line:

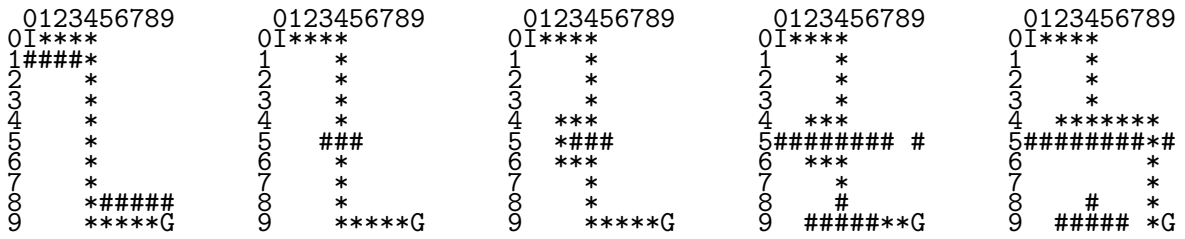

Figure 2: Examples of grid visualizations using ASCII characters.

```
==STAGE 1====================================
```

It should be followed by a visualization of the input route in the grid using ASCII characters, as shown in Figure 2; note that each of the five subfigures in Figure 2 is encoded using eleven rows, where each row has eleven characters with blanks encoded using the character with ASCII code 32. For example, Figure 2a shows the intended ASCII encoding of the grid and route from the `test0.txt` input file (also shown in Figure 1a).

In an ASCII visualization, the initial cell is encoded as character `I`, the goal cell as `G`, a cell with a block as `#`, and a visited (at least once) cell on the route as `*`. If a visited cell is also the initial cell, goal cell, or contains a block, then `I`, `G`, or `#`, respectively, should be printed, not `*`. If the *status* of the input route is not equal to 4, refer to the description of Stage 0, the output of Stage 1 terminates. Otherwise, your program should attempt to repair ***only the first*** **(closest to the initial cell)** broken segment of the route, and print (i) the ASCII visualization of the repaired route in the grid and (ii) all the steps and status of the repaired route. For example, Figure 2b shows the ASCII visualization of the grid and route from the `test1.txt` input file (also shown in Figure 1b), while Figure 2c shows the ASCII visualization of the corresponding repaired route (also shown in Figure 1c). The visualizations of the input grid and route, the visualization of the repaired route in the grid, and the steps and status of the repaired route should be separated by the line below.

```
---------------------------------------------
```

The output of the repaired walk should follow the instructions given in Stage 0. File `test1-out-mac.txt` contains the exact result of the execution: "`mac: ass2-soln < test1.txt > test1-out-mac.txt`".

We exemplify the procedure for repairing a broken route segment using the route in Figure 1b; also shown using ASCII visualization in Figure 2b. This route has one broken segment that starts at cell `[4,4]` and ends at cell `[6,4]`. To repair this segment, we construct a queue of pairs, where each pair is composed of coordinates of a grid cell and a counter value. We then initialize the queue with the pair composed from the cell where the broken segment starts and counter value of zero, the pair `([4,4],0)` in the running example. Starting from the first pair, we then traverse the queue. When traversing a pair in the queue, for each cell in the grid that is adjacent to the cell in the traversed pair and is not blocked, we add a fresh pair to the end of the queue composed of the adjacent cell and a counter value that is greater than the counter value in the currently traversed pair by one. When visiting the adjacent cells in order to add fresh pairs to the queue, we first visit the cell above, then below, then to the left, and finally to the right from the current cell. Hence, after visiting the pair `([4,4],0)`, the queue contains`([4,4],0),([3,4],1),([4,3],1),([4,5],1)`. The traversal then continues at pair `([3,4],1)`. The traversal of the queue terminates once the last pair in the queue is processed or a pair that contains a cell in the route that follows the start of the broken segment is added to the queue. In the latter case, we have discovered a repair of the broken segment!

In the running example, when the traversal of the queue terminates, the queue contains pairs with all the cells marked with numbers in Figure 1f; the numbers indicate the counter values associated with the corresponding cells, while the last pair in the queue is `([6,4],6)`. Next, we construct a route fragment between cell s at which the broken segment starts and cell t from the last pair in the queue by walking from cell t towards cell s (going backward) by progressing, at each cell, towards an adjacent cell with the smallest counter value; if multiple adjacent cells have the same counter value, the preference is given to the one that comes earlier in this list: above, below, left, right of the current cell. Thus, in the running example, the fragment of the route `[4,4]->[5,4]->[6,4]` gets replaced with `[4,4]->[4,3]->[4,2]->[5,2]->[6,2]->[6,3]->[6,4]`.

Note that the described procedure, as a side effect, can repair multiple broken segments, as it happens with the broken route in Figure 1d and the result of repairing its broken segment that starts at cell `[4,2]` shown in Figure 1e (ASCII visualizations are shown in Figure 2d and Figure 2e, respectively).

Stage 2 – Full Repair and (15/15 marks)

Extend your program from Stage 1 to repair all broken segments in a route and handle multiple changes of block positions in the grid. Each new configuration of blocks will be provided from `stdin` following after input line with a single character `$`. Once a new configuration of blocks is loaded (given in the format described under Stage 0), reassess the status of the most recently processed route, either the route resulting from Stage 1 or the last repair in Stage 2. If this route has the status of 4, iteratively apply the repair procedure from Stage 1, each time on the first broken segment in the route, either until no more broken segments remain or the next segment cannot be repaired due to the configuration of blocks. If the route was broken and successfully repaired, print the grid with the route before the repair, after the repair, and the repaired route and its status. Otherwise, only print the grid with the valid route. If the route was broken and could not be repaired, print the grid with the broken route and this message: `The route cannot be repaired!`. For the exact format of output for Stage 2 see output files `test2-out-mac.txt` and `test3-out-mac.txt` generated for `test2.txt` and `test3.txt`.

Beyond the Scope of the Project

Use your creativity and extended ASCII and/or UTF characters to print the grids in a fun, artistic, or some other cool way, and share with us your masterpiece grids. Note that this part of the project should **not** be submitted.

The boring stuff...

This project is worth 15% of your final mark. A rubric explaining the marking expectations is provided on the FAQ page. You need to submit one program for assessment; detailed instructions on how to do that will be posted on the FAQ page. 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. Marks and a sample solution will be available on the LMS before **Tuesday 5 November**.

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. 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 either lose marks through the marking rubric, or will be referred to the Student Center for possible disciplinary action without further warning. This message **is** the warning.* See `https://academicintegrity.unimelb.edu.au` for more information. Note that solicitation of solutions via posts to online forums or marketplaces, whether or not payment is involved, is also Academic Misconduct. In the past students have had their enrollment terminated for such behavior. **The FAQ page contains wording for an Authorship Declaration that you **must** include as a comment at the top of your submitted program. A significant fraction of the marks will be deducted if you do not do so.**

Deadline: Queries about this specification should be sent to `artem.polyvyanyy@unimelb.edu.au`. 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. Programs not submitted by **10:00am on Monday 21 October** will incur penalty marks at the rate of two marks per day or part day late. 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.

Remember, algorithms are fun! And yes, this document was produced using LaTeX ;)