School of Computing and Information Systems

COMP10002 Foundations of Algorithms
Semester 2, 2017
Assignment 2

Learning Outcomes

In this project you will demonstrate your understanding of dynamic memory and linked data structures, and extend your skills in terms of program design, testing, and debugging. You will also learn about graph labeling, and implement a simple shortest path mechanism (broadly speaking, the equivalent of bubblesort) in preparation for the more principled approaches that are introduced in comp20007 Design of Algorithms.

Path Planning

Any process that involves movement requires path planning. If we are at location A and wish to get to location B, we seek out and follow the "best" route between them, where "best" might be shortest (in terms of distance), or fastest (in terms of time taken), or cheapest (in terms of cost), or most scenic (in terms of appeal), or any weighted combination of those factors. This simple operation happens when we use Google maps, when we request quotes for airline tickets, and when we use our phone to request an Uber pickup.

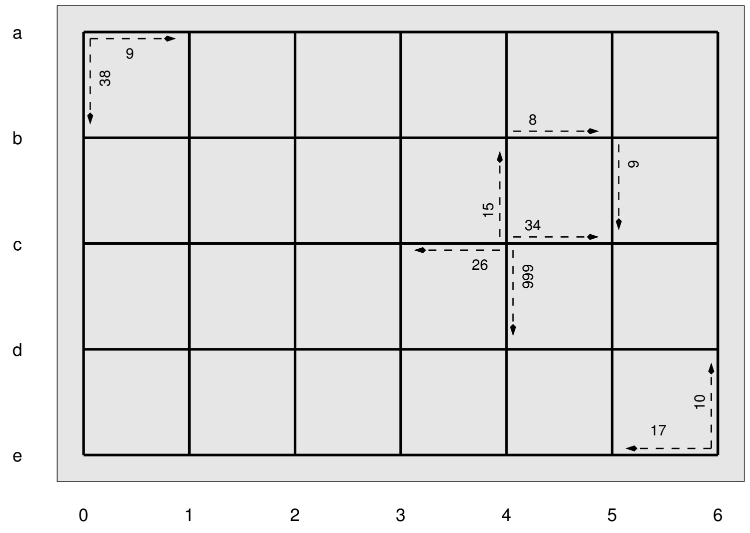
Rectangular Grids

The founders of Gridsville were a mix of mathematicians and computer scientists. So when they discovered an uninhabited rectangular island off the south coast of Antarctica that they could use for their new colony, they planned a town using a rectangular grid of streets running north-south and east-west. The diagram on the next page shows the seven north-south streets that they built, and the five east-west streets. Streets are named with digits ("0" to "6") or letters ("a" to "e"), and intersections are labeled by their locations on the grid that results, for example, "4c". (They were very boring people.)

Hundreds of years later, Gridsville is finally getting an Uber-like service, provided courtesy of the Gridsville City Council (gcc, get it?). But the Gridsville Council has remained very traditional and is insisting that pick-ups and drop-offs may only happen at street corners, that is, locations that are addressable via the grid coordinates. To help prepare for the arrival of the new service, the Council has approached a local software company, Algs-R-Fun, and has commissioned software to help manage the vehicle fleet. In particular, each time a customer (standing at an intersection in the grid) calls for a pickup, the "best available" car should be directed to go and pick them, where "best" means "the one that can get there the fastest". The Council has already collected some travel-time measurements, and has recorded for each street, and for each city block on that street, how long it takes to travel in that direction along that block. For example, looking at Figure 1, travel east from "4c" to "5c" takes 34 seconds. Some streets cannot be used in some directions because they are one-way, and in that case the Council notes them as having a cost of 999 seconds. In the example in Figure 1, it is not possible to travel south from "4c" to get to "4d".

Stage 1 – Reading the Data (marks up to 8/15)

Write a program that reads data from `stdin` in the format illustrated on the right-hand side of Figure 1 (see `test2.txt` linked from the LMS page for the full file) where the two numbers on the first line are always the grid dimensions, x and y (they will be 7 and 5 respectively for Gridsville, but might be different for other towns elsewhere in the world, and the GCC has bold ideas about being able to sell



Input `test2.txt`
39 lines in total:

```
7 5
0a   9 999 999  38
1a ... 999 ... ...
2a ... 999 ... ...
...
4c  34  15  26 999
...
4e ... ... ... 999
5e ... ... ... 999
6e 999  10  17 999
4c
5c
1e
```

Figure 1: Street layout in Gridsville. Intersections are labeled by their grid coordinates. See the FAQ page for the full `test2.txt` input file for this example.

their software to other towns), followed by $x \times y$ lines, each line describing the travel times out of one intersection; and where "`...`" in the figure represent further integer values. Those $x \times y$ lines (35 of them for Gridsville) will always be presented in row-at-a-time order of intersection name, and each intersection name will always be an integer immediately followed by a single alphabetic character starting from `a`, and is followed by exactly four more integers on that line, representing the time in seconds needed to travel (respectively) one block east, one block north, one block west, and one block south of that intersection.

Your program should build an internal representation of the specified intersections, using `structs` where ever appropriate. You may assume that all input files you will be provided with will be "correct", according to the description given above, that all of the times will be strictly positive integers, and that there won't be any tricksy-wicksy special cases introduced during the testing, nor any missing lines, nor any incorrect or wacky values in the lines; but you may *not* assume any upper limit on the number of streets involved in the city grid. Note also that the input values will be `int`, to keep it simple and avoid the rounding problems associated with floating point representations.

The grid size line and grid cost data lines are is followed by a third section of data in the input file: one or more grid references, one per line. You are also to read those grid references into a suitable data structure. Required output from this stage is the following summary:

```
mac: ass2-soln < test2.txt
S1: grid is 7 x 5, and has 35 intersections
S1: of 140 possibilities, 37 of them cannot be used
S1: total cost of remaining possibilities is 2115 seconds
S1: 3 grid locations supplied, first one is 4c, last one is 1e
```

You may use any data structure that you feel to be appropriate for the grid and cost data, including a two-dimensional array of struct, a one-dimensional array of struct, or a linked structure. For what it's worth, my sample solution uses a one-dimensional array of struct as the storage component for the city grid and the travel times. All storage used for input data should be created using `malloc()` and/or `realloc()`, and `free()`'ed at the end of the program.

Stage 2 – Path Finding (marks up to 13/15)

Suppose you are a driver currently at location “4c”. There is a minimum travel cost from there to every other location in the grid, based on the per-block travel times. To see how these costs can be computed, suppose we label “4c” with a cost of 0, and every other grid location with a cost of 999 (meaning, not yet reached). If we then use nested loops to explore each possible direction out of every possible grid location, we’ll find that some of those cost estimates were too high, and can be reduced. For example, after one cycle of reductions, it will be known that grid reference “4b” has a cost of 15, that “5c” has a cost of 34 (or less, because there is a cheaper route found later, via “4b”), and that grid location “3c” has a cost of 26 of less – all of them labeled from “4c”, using the streets out of it. The second iteration – again, over every grid location and every edge out of it – will then allow initial costings to be established for the grid locations connected to “4b”, “5c”, and “3c”. For example, “3b” will be checked via both “3c” and “4b”. Every time a path is considered, the least cost one should be retained. In the case of ties, retain the labeling that comes from the lexicographically smallest grid location.

If we continue to repeat that whole process until we have an iteration (over all grid locations and directions) in which there are no more changes made, then the costs have stabilized and reached their final minimum values. Can you see now why I compared this (relatively inefficient) algorithm to bubblesort? [If you want to try and implement a more efficient mechanism, look up “Dijkstra’s Algorithm” at the wikipedia. But remember, there are no marks in this project for courage. Get a simple version working first.] While you are working through getting this process implemented, you should use a `DEBUG` mode to generate detailed output in any form that you wish. Turn the debugging off again before your final submission.

For this stage, the required output is the path (and cost of it) from the first additional grid location in the third part of the input file to each of the other grid locations in the third part of the input file (that is, compute all of the costs, and then print out a selected subset of them). For `test2.txt`:

```
S2: start at grid 4c, cost of 0
S2:       then to 4b, cost of 15
S2:       then to 5b, cost of 23
S2:       then to 5c, cost of 32
S2: start at grid 4c, cost of 0
S2:       then to 3c, cost of 26
S2:       then to 3d, cost of 36
S2:       then to 3e, cost of 48
S2:       then to 2e, cost of 70
S2:       then to 1e, cost of 80
```

Hint: as you label each grid location with a better cost, make a note of where that labeling came from. Then unstack that path (think recursion) when you need to trace it back to the starting point. You may assume that every grid location can be entered and exited somehow, and do not need to worry about dead ends being created.

Stage 3 – Send the Closest Car (marks up to 15/15)

Ok, now for the fun part. Suppose that the grid coordinates supplied in the third part of the input file are the locations of the currently available cars. For each pickup location in the city grid, compute which car can get there the fastest from its start position. Present your solution visually, as shown in Figure 2. My program to implement all three stages is around 450 lines long, including detailed comments and some debugging output, about 100 lines longer than the solution for Assignment 1. But it is also rather more complex. *Start early if you intend to try and obtain these last two marks!*

```
S3:          0       1       2       3       4       5       6
S3:   +---+-------+-------+-------+-------+-------+-------+
S3: a |  75      75 <<<< 53 <<<< 42 <<<< 31      31      39
S3:   |   ^                               ^       ^       ^
S3:   |   ^                               ^       ^       ^
S3: b |  44      64 <<<< 46      36 <<<< 15      10 >>>> 27
S3:   |   ^               ^               ^       ^       v
S3:   |   ^               ^               ^       ^       v
S3: c |  42      43      25      26 <<<<  0       0      49
S3:   |   ^       ^       ^                       v
S3:   |   ^       ^       ^                       v
S3: d |  28      12       8 >>>> 26 >>>> 60       8      47
S3:   |   ^       ^       ^                       v       ^
S3:   |   ^       ^       ^                       v       ^
S3: e |  13 <<<<  0 >>>>  5 >>>> 26 >>>> 70      26 >>>> 37
```

Figure 2: Example of Stage 3 output. More examples are linked from the FAQ page. If there is disagreement between the examples on the FAQ page and this handout, follow the ones in the FAQ.

The boring stuff...

This project is worth 15% of your final mark. A rubric explaining the marking expectations will be provided on the FAQ page.

Submission will again be done via `dimefox` and the `submit` system. 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 and executes correctly on our test system, which has some different characteristics to the lab machines. 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://academicintegrity.unimelb.edu.au` for more information. Note also that solicitation of solutions via posts to online forums, whether or not there is payment involved, is also taken very seriously. In the past students have had their enrolment terminated for such behavior.

Deadline: Programs not submitted by **10:00am on Monday 16 October** 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 31 October.

And remember, algorithms are fun!