School of Computing and Information Systems

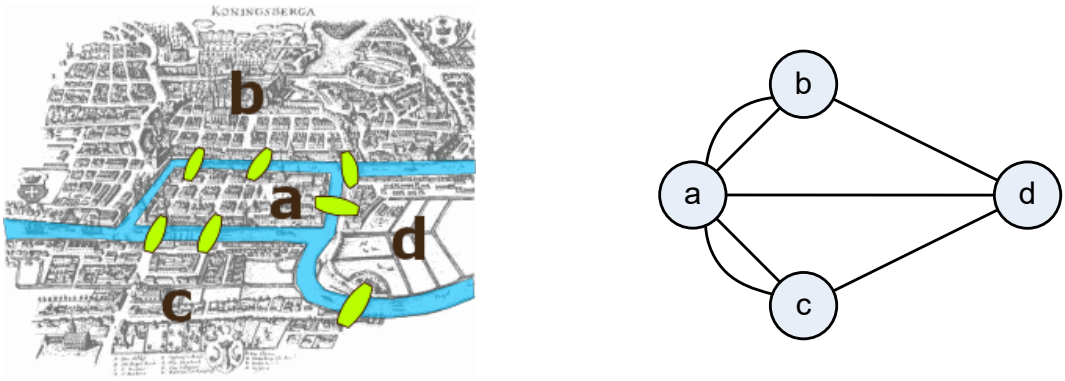
comp10002 Foundations of Algorithms
Semester 2, 2018
Assignment 2

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 algorithms, and implement a simple path discovery mechanism in preparation for the more principled approaches that are introduced in comp20007 Design of Algorithms.

Background — The Königsberg Bridge Problem

In the 18th century, in the town of Königsberg, there were seven bridges that crossed the river Pregel. These bridges connected two islands in the river with each other and with opposite banks. The city council of Königsberg and its citizens for a long time considered this problem: *Is it possible to have a continuous walk to cross all the seven bridges without recrossing any of them?* This problem is known as the Königsberg bridge problem and belongs to the general area of graph problems. Figure 1a shows a map of Königsberg from the 18th century[1], while Figure 1b gives a *multigraph* representation of the Königsberg problem.



(a) (b)

Figure 1: (a) A city map of Königsberg from the 18th century with two islands in the river and seven bridges, and (b) the Königsberg bridge problem represented as a multigraph.

In Figure 1b, a, b, c, and d are *vertices* that represent corresponding land areas from Figure 1a. Every two vertices in the multigraph are joined by a number of *edges* equal to the number of bridges joining the corresponding land areas. Given the multigraph representation of the problem from Figure 1b, the Königsberg bridge problem can be translated into the problem of determining whether this multigraph has a *trail* or a *circuit* that contains all its edges; note that such a trail or a circuit can start/end at any vertex of the multigraph.

A multigraph can have several edges with the same end vertices. For example, in Figure 1b, there are two edges with end vertices a and b, and two edges with end vertices a and c. Given two vertices u and v in a multigraph, a u-v *walk* is an alternating sequence of its vertices and edges that starts at u and ends at v, such that every edge in the sequence joins the vertices immediately preceding and following it. For example, these are three example walks in the multigraph shown in Figure 1b:

[1] Figure 1a is an adapted version of the figure at `https://goo.gl/fBfCkK`.

1. $a \to d \to a$
2. $a \to b \to d \to c$
3. $a \to b \to a \to c \to a$

A walk in a multigraph is a *trail* if it does not contain repeated edges, while a trail that ends at its start is a *circuit*. Hence, a-a walk 1 shown above is not a trail as it traverses the same edge twice, a-c walk 2 is an a-c trail but is not a circuit and, finally, a-a walk 3 is an a-a circuit of the multigraph in Figure 1b, considering that no two edges in walk 3 are the same.

A naïve strategy to solve the Königsberg bridge problem is to construct all possible sequences of all the edges of the multigraph, such that in each sequence each edge appears exactly once, and check if any such sequence is a trail or a circuit. Three such example sequences of edges are listed below:

1. $b-a, a-c, c-a, b+d, d+a, a+b, c+d$
2. $d-c, c-a, a-c, d+a, a+b, b+a, b+d$
3. $a-b, b-a, a-c, c-a, a-d, d-b, c+d$

None of the proposed sequences can be used to construct a walk of seven edges, see the '$+$' edges starting at the places where walks break. This naïve algorithmic strategy is computationally complex, as given a multigraph with n edges, it constructs and checks $n!$ edge sequences. For example, to solve the Königsberg bridge problem this strategy must construct and check 7! = 5040 edge sequences. Though 7! may appear to be a small number, if one runs the same algorithmic strategy on a multigraph composed of 60 edges, the number of edge sequences that need to be examined will roughly equal to the believed number of atoms in the observable universe.

The *degree* of a vertex in a multigraph is the number of edges *incident* to the vertex, with loops counted twice (where an edge and a vertex on that edge are called incident). For example, the degree of vertex a in Figure 1b is 5, while the degree of vertex d in Figure 2 is 6. In 1736, Leonhard Euler, a Swiss mathematician, demonstrated the negative resolution of the Königsberg bridge problem. He also described two special classes of multigraphs. An *Eulerian trail* of a multigraph is a trail that includes all the edges of the multigraph, while an *Eulerian circuit* is a circuit that includes all its edges. According to Euler, a multigraph with an Eulerian trail is called *traversable*, whereas a multigraph with an Eulerian circuit is called *Eulerian*. Euler demonstrated that a multigraph is traversable if and only if it is *connected*, that is there exists a walk between any two vertices of the multigraph, and has *exactly two vertices of odd degree*. Finally, a multigraph is Eulerian if and only if it is connected and *all its vertices are of even degree*. As all the vertices of the multigraph in Figure 1b are of odd degree, it is neither traversable nor Eulerian and, hence, there exists no continuous walk that crosses all the seven bridges without recrossing any of them.

More Background — The A&A Travel Agency

Artem and Alistair have decided to start a walking city tours company called A&A; yes, why not! To prepare for the launch, they need to come up with scenic walking routes in various cities. To accomplish this task, they designed several algorithms. As input, these algorithms take a street map encoded as a multigraph, and a starting point for the route, and as output produce a walking route of a high scenic value. A vertex in such an input multigraph encodes a crossroad in a city map and an edge represents a street that joins two crossroads. Each constructed walking route should start and end at the same crossroad/vertex and, to prevent boredom,

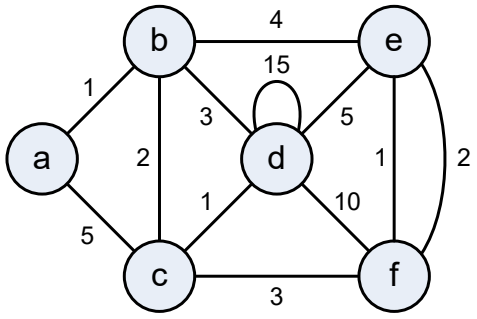


Figure 2: A multigraph representation of a street map annotated with scenic values. The numbers next to the edges encode the scenic values of the corresponding streets.

should never follow the same street/edge twice. Hence, the desired walking routes are Eulerian circuits! Artem and Alistair also plan to annotate each edge in every multigraph with its scenic value, a natural number between 1 and 100 (including 1 and 100) that reflects the importance of the corresponding street based on the perception of its touristic attractiveness, and intend to construct routes of high overall scenic values, where the scenic value of a route that takes a sequence of streets with individual scenic values of $val_1, val_2, \ldots, val_k$, is computed as $\sum_{1 \le i \le k} i \times val_i$. Such a cost model gives preference to routes with high scenic streets at the end. For example, this is a valid walking route for the input multigraph in Figure 2:

```
a-1->b-4->e-1->f-3->c-1->d-10->f-2->e-5->d-15->d-3->b-2->c-5->a
```

This route has the overall scenic value of:

$$1 \times 1 + 2 \times 4 + 3 \times 1 + 4 \times 3 + 5 \times 1 + 6 \times 10 + 7 \times 2 + 8 \times 5 + 9 \times 15 + 10 \times 3 + 11 \times 2 + 12 \times 5 = 390.$$

Input Data

Your program should read input from the command line and `stdin`. From the command line, it should read one alphabetic character, either lower or upper case. Each line of the input from `stdin` will (always) consist of two alphabetic characters followed by a natural number; the characters and the number will be separated by a single space character. Each input line will encode one distinct edge of the input multigraph, where first two characters in the line represent two vertices joined by the edge, and the natural number encodes the scenic value of the corresponding street. Note that each input multigraph will be connected and, thus, there is no need to implement this check. For example, the following file `test1.txt` encodes the multigraph in Figure 2:

```
d d 15
a c 5
b d 3
a b 1
b e 4
c d 1
e f 2
c f 3
d e 5
d f 10
b c 2
e f 1
```

As vertices are encoded as alphabetic characters, an input multigraph will never contain more than 52 vertices. A graph with m vertices can contain at most $n(n-1)/2$ edges; yes, this is yet another application of the triangular numbers ;) However, as city maps can contain multiple streets that connect two crossroads, there is no limit on the maximal number of edges in the input multigraphs.

You may use any data structure that you feel to be appropriate to encode street maps. For what it's worth, my sample solution uses a one-dimensional array of 52 linked lists, where each list stores edges incident to the vertex represented by the corresponding position in the array. All storage used for input data should be created using `malloc()` and/or `realloc()`, and `free()`'ed at the end of the program. My sample solution uses the linked list implementation from the `listops.c` file as a starting point. Thus, this implementation is included into the proposed `ass2-skel.c` file (available via the FAQ page). You may decide to use or not to use this implementation in your program.

Each of the following stages requires a function that carries out the required processing, in each case constructing a scenic route of a given street map and computing its overall scenic value.

Stage 0 – Reading Input Data (5/15 marks)

Before tackling the more interesting parts of the project, your very first program should access the starting point for a route from the command line and a weighted multigraph from `stdin`, and then 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 street map is indeed encoded as an Eulerian multigraph, i.e., it contains a walking route of interest. Required output from this stage is the following summary (generated for the `test1.txt` example input file proposed above):

```
mac: ass2-soln b < test1.txt
Stage 0 Output
--------------
S0: Map is composed of 6 vertices and 12 edges
S0: Min. edge value: 1
S0: Max. edge value: 15
S0: Total value of edges: 52
S0: Route starts at "b"
S0: Number of vertices with odd degree: 0
S0: Number of vertices with even degree: 6
S0: Multigraph is Eulerian
```

If the input multigraph has exactly two vertices of even degree, the last line of the output should be:

```
S0: Multigraph is traversable
```

If the multigraph is neither traversable nor Eulerian, the output of Stage 0 must skip the last line. Your program should proceed to Stage 1 only if the input multigraph is Eulerian; otherwise, your program should terminate by returning `EXIT_FAILURE`.

Stage 1 – Simple Scenic Route (10/15 marks)

In 1873, Hierholzer proposed a simple yet effective algorithm for finding Eulerian circuits. Here we propose a modified version of the Hierholzer's algorithm which proceeds as follows:

- Starting from a given vertex v of an input multigraph, follow a trail of edges until you return to v. The fact that this procedure is performed on an Eulerian multigraph with all its vertices of even degree (otherwise your program should not reach this stage) ensures that the trail does not get stuck at any vertex before reaching vertex v again. The trail obtained in this way is a circuit. However, this circuit may not cover all the edges of the input multigraph.
- If there exists a vertex u that is visited by the currently constructed circuit but has an incident edge that is not part of this circuit, construct another trail that starts and ends at vertex u using only so far unvisited edges, and join the constructed trail that starts and ends at u with the previous circuit. The resulting sequence of edges is again a circuit.
- Repeat the previous step until all the edges of the multigraph are visited.

As the above algorithm ignores edge weights, you are asked to implement its enhanced version that postpones visits of edges with high scenic values:

- Starting from a given vertex v, follow a trail of edges until you return to v. At each stage of the trail construction procedure, out of unvisited candidate edges, select the next edge to be an edge with the lowest scenic value. If several candidate edges have the same scenic value, select an edge that leads to a vertex represented by a character with the smallest ASCII code.
- If there exists a vertex that is visited by the currently constructed circuit but has an incident edge that is not part of the circuit, select the first such vertex u in the current circuit and construct another trail that start and ends at vertex u using only so far unvisited edges and prioritizing the selection of next edges in the trail in the same way as described in the previous step. Once constructed, join the circuit that starts and ends at u with the previous circuit.

- Repeat the previous step until all the edges of the multigraph are visited.

Required output from this stage is the following summary (for `test1.txt` example input file):

```
Stage 1 Output
--------------
S1: b-1->a-5->c-1->d-3->b
S1: b-2->c-3->f-1->e-2->f-10->d-5->e-4->b-1->a-5->c-1->d-3->b
S1: b-2->c-3->f-1->e-2->f-10->d-15->d-5->e-4->b-1->a-5->c-1->d-3->b
S1: Scenic route value is 332
```

Starting from vertex b in the multigraph in Figure 2, by repeatedly selecting the next edge with the lowest scenic value one constructs circuit `b-1->a-5->c-1->d-3->b`, refer to the first line of summary generated for Stage 1. In this circuit, vertices b, c, and d have incident edges that are not included in the circuit. As vertex b is visited first in the current circuit, another circuit that is composed of the so far unvisited edges and starts and ends at vertex b gets constructed (this is circuit `b-2->c-3->f-1->e-2->f-10->d-5->e-4->b`) and *inserted* in the current circuit at the first occurrence of b, refer to the second line of the summary. Finally, circuit `d-15->d` gets inserted into the resulting circuit at the place of the first occurrence of vertex d (which is the only vertex on the current circuit with an incident unvisited edge) to result in the Eulerian circuit shown in the third line of the summary. Your program should print to `stdout` the first ten, every fifth, and the final constructed circuit, each circuit gets printed once in the chronological order of construction. Finally, the last line of the summary for Stage 1 reports on the overall scenic value of the constructed Eulerian circuit.

If you need to print a circuit of more than twelve edges, print the first and last six edges joined by the "..." string, refer to the sample output files at the FAQ page for the exact formatting.

Stage 2 – Greedy Scenic Route (15/15 marks)

Modify the algorithm implemented in Stage 1 to attempt all the possible extensions of the currently constructed circuit. The first circuit constructed in Stage 1, that is `b-1->a-5->c-1->d-3->b`, can be extended at four places (the first and last occurrence of vertex b and vertices c and d). At each iteration, test all the possible extensions and apply the one that leads to a circuit with the highest overall scenic value. For example, the best next extension of `b-1->a-5->c-1->d-3->b` is the extension at the second occurrence of vertex b with the circuit `b-2->c-3->f-1->e-2->f-10->d-5->e-4->b`, as it results in a scenic route of the overall value of 261. Note that every constructed extension of the current circuit and the output of your program should obey all the rules described in Stage 1.

Hence, required output from this stage is the following summary (for `test1.txt` input file):

```
Stage 2 Output
--------------
S2: b-1->a-5->c-1->d-3->b
S2: b-1->a-5->c-1->d-3->b-2->c-3->f-1->e-2->f-10->d-5->e-4->b
S2: b-1->a-5->c-1->d-3->b-2->c-3->f-1->e-2->f-10->d-15->d-5->e-4->b
S2: Scenic route value is 420
```

If at some iteration your program constructs several circuits with the same overall value, select the one that was obtained via an extension of a vertex at the lowest position in the current circuit.

Beyond the Scope of the Project

If you feel like some fun, then try to improve your program further (but please don't submit any such programs for assessment, even if you get them working prior to the submission deadline). Modify the algorithm implemented in Stage 2 to attempt all possible sequences of the proposed circuit extensions to select an Eulerian circuit with the best overall scenic value. It turns out that it is better to extend the

initial circuit `b-1->a-5->c-1->d-3->b` with `d-5->e-1->f-2->e-4->b-2->c-3->f-10->d` and then with `d-15->d` at the second occurrence of vertex d in the previously constructed circuit. The resulting route is shown below and has the overall scenic value of 423:

`b-1->a-5->c-1->d-5->e-1->f-2->e-4->b-2->c-3->f-10->d-15->d-3->b.`

General tips...

You will probably find it helpful to include a `DEBUG` mode in your program that prints out intermediate data and variable values. Use `#if (DEBUG)` and `#endif` around such blocks of code, and then `#define DEBUG 1` or `#define DEBUG 0` at the top. Turn off the debug mode when making your final submission, but leave the debug code in place. The FAQ page has more information about this.

The sequence of stages described in this handout is deliberate – it represents a sensible path though to the final program. You can, of course, ignore the advice and try and write final program in a single effort, without developing it incrementally and testing it in phases. You might even get away with it, this time and at this somewhat limited scale, and develop a program that works. But in general, one of the key things that makes some people better at programming than others is the ability to see a design path through simple programs, to more comprehensive programs, to final programs, that keeps the complexity under control at all times. That is one of the skills this is subject is intended to teach you.

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 your program for assessment; detailed instructions on how to do that will be posted on the FAQ page 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. Marks and a sample solution will be available on the LMS by Monday 29 October.

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 also that solicitation of solutions via posts to online forums or marketplaces, whether or not there is payment 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. Marks will be deducted if you do not do so.*

Deadline: Programs not submitted by **10:00am on Monday 15 October** will incur penalty marks at the rate of two marks per day or part day late. Queries about the project specification should be directed to `artem.polyvyanyy@unimelb.edu.au`, while 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.

And remember, algorithms are fun!