ESC190H1S Winter 2022: Lab 5

Graphs

TA evaluations in practicals: April 5th/6th, 2022
Code submission due: April 7th, 2022 at 11:59pm

Overview

In this lab, we will practice working with graphs, taking inspiration from our friendly neighbourhood Toronto Transit Commission.

Lab Submission and Grading

It is highly recommended that you read through the entire document and starter code before starting this lab. The following files have been provided for this lab (only submit `lab5.c` for grading):

- `lab5.c`, which will contain your code for submission
- `test5.c`, which will contain some sample code for testing
- `extras.c`, which will contain some extra functions that you are welcome to use (these will be added automatically to Gradescope)
- `lab5.h`, which will contain some `struct` and function declarations

Violation of any of these constraints may result in a grade of zero from the autograder.

- You **MUST NOT** modify `lab5.h` or `extras.c`.
- Your code **MUST** compile on the autograder to receive any credit.
- You **MUST NOT** modify the function signatures.

Ensure your filenames and function names are identical to those specified in this lab handout in order for your work to be marked. There are also some short answer questions throughout the lab to complete and show your TA during the lab session. Feel free to use any code posted on the course Quercus, making sure to add any additional `struct` definitions and helper functions to `lab5.c`. Some testing code has been provided for you in `test5.c`, but you should do additional testing to convince yourself that your code works and runs clean with Valgrind. Add your lab partner as a teammate on Gradescope.

Part 1. Dijkstra's for the Delivery?

Sandy and Cima have decided to expand their restaurant's offerings to include food delivery service. Sandy suggests adopting all the cats at their local cat rescue shelter and training them to handle the deliveries in exchange for treats. Cima counter-proposes that they invest in a fleet of ultra-light, full-carbon Bianchi Specialissimas to get the job done themselves. Either way, they will need to carefully plan their routes in order to get the deliveries done efficiently. Inspired by the ESC190 Week 10 lectures, Sandy and Cima decide to use a graph to plan their delivery routes. They opt to build a graph based on the city's public transit stations so that they can switch to public transit mid-route if the weather takes a turn for the worst.

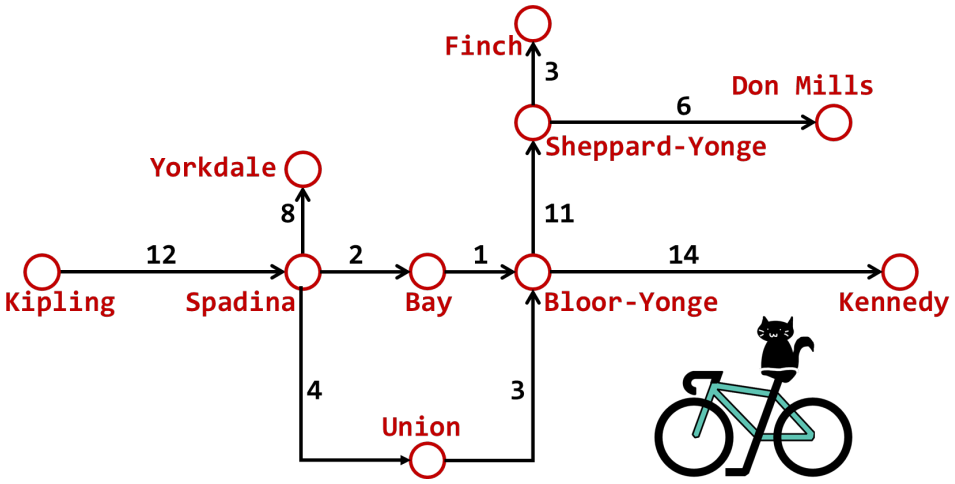

Figure 1. Sample graph of public transit stations. Vertices/transit stations are indicated in `merlot` and edges/service routes (with weight values) are indicated in `black`. Image not to scale.

Question: Refer to the graph in **Figure 1**. What is the shortest path from `Kipling` station to `Don Mills` station? Calculate the costs associated with each vertex in the graph after using Dijkstra's Algorithm to find the shortest path between these vertices.

Task: Complete the following function in `lab5.c`:

```
char **plan_route(Graph *gr, char *start, char *dest);
```

This function generates the shortest path from two vertices - we recommend using Dijkstra's Algorithm! You can assume the edge weights in the graph will be integers with values between 1 and 5000, inclusive.

Inputs: `<gr>` is a pointer to a graph, implemented with an adjacency list. `<start>` and `<dest>` are pointers to strings containing vertex names.

Output:

- If a path from `<start>` to `<dest>` exists in the graph, return a pointer to an array of strings, representing the sequence of vertices in the shortest path, tracing backwards from `<dest>` to `<start>`. The first element of the array should be a pointer to the same string as `<dest>` and the last element of the array should be a pointer to the same string as `<start>`.
- If no path exists between the `<start>` and `<dest>`, return a null pointer.

Part 2. Service Advisories for D(el)ays

In order for Sandy and Cima to plan their routes accurately, they need to be able to update the graph when there are changes to public transit service. Potential modifications to consider include:

- Adding a vertex when a new station is built.
- Updating the value of an edge when service between two stations is altered.
- Removing a vertex (and all the edges connected to it) when service at a station is disrupted.

Question: Draw what the graph in **Figure 1** would look like after making the following modifications:

- Add a vertex called `Sheppard West`.
- Add an edge from `Yorkdale` to `Sheppard West` with weight `5`.
- Add an edge from `Sheppard West` to `Sheppard-Yonge` with weight `4`.
- Remove the `Bay` station vertex (and all the edges connected to it).

What is the shortest path from `Kipling` to `Don Mills` after making these modifications? Calculate the costs associated with each vertex in the graph after using Dijkstra's Algorithm to find the shortest path between these vertices.

Task: Complete the following functions in `lab5.c`:

```
void add(Graph *gr, char *station);
```

This function attempts to add a new vertex to a graph.

Inputs: `<gr>` is a pointer to a graph to be modified, implemented with an adjacency list. `<station>` is a pointer to a string containing a vertex name.

Output:

- If a vertex named `<station>` already exists in the graph, do nothing.
- Otherwise, modify the graph to contain a new vertex named `<station>`. The new vertex should be initialized to have no neighbours.

```
void update(Graph *gr, char *start, char *dest, int weight);
```

This function attempts to change a directed edge within a graph.

Inputs: `<gr>` is a pointer to a graph to be modified, implemented with an adjacency list. `<start>` and `<dest>` are pointers to strings containing vertex names. `<weight>` is an integer between 0 and 5000, inclusive.

Output:

- If `<weight>` is 0, the edge from `<start>` to vertex `<dest>` should be removed from the graph, if it exists. If it doesn't exist, do nothing.
- If `<weight>` is not 0, the edge from `<start>` to vertex `<dest>` should be modified to have a weight of `<weight>`. If the edge, `<start>` vertex, or `<dest>` vertex, do not already exist in the graph, add them to the graph.

```
void disrupt(Graph *gr, char *station);
```

This function attempts to remove a vertex from a graph. All edges connected to the removed vertex should also be removed from the graph.

Inputs: `<gr>` is a pointer to a graph to be modified, implemented with an adjacency list. `<station>` is a pointer to a string containing a vertex name.

Output:

- If the graph contains a vertex named `<station>`, remove it and any edges that are connected to it in either direction.
- If there are no vertices named `<station>` in the graph, do nothing.

This is the last lab for the semester, so remember to thank your TA for a great semester!