COMP2521: Assignment 2
Social Network Analysis

A notice on the class web page will be posted after each major revision. Please check the class notice board and this assignment page frequently. The specification may change.

Change log:

- [08/04 16:50] Removed some incorrect comments and preprocessor instructions from the provided `.c` files.
- [13/04 14:40] Added testing files and a part-3 example.
- [13/04 16:20] Fixed some issues with the testing files and added header guards to `.h` files.
- [14/04 16:40] Fixed a mistake with the expected files for Part 3.
- [16/04 12:30] Added a note for Part 3.

Objectives

- to implement graph based data analysis functions to mine a given social network
- to give you further practice with C and data structures

Admin

Marks	20 marks
Individual Assignment	This assignment is an individual assignment.
Due	11:59:59pm Friday 23 April 2021 (Friday of Week 10)
Late Penalty	2 marks per day off the ceiling. The latest you can submit the assignment (with late penalty) is 8pm Monday 26 April 2021.
Submit	Read instructions in the Submission section below.

Aim

In this assignment, your task is to implement graph-based data analysis functions to mine a given social network. You should start by reading the Wikipedia entries on these topics:

- Floyd-Warshall algorithm
- Edge betweenness and community structure

The main focus of this assignment is to calculate measures that could identify "influencers", "followers", etc., and also discover possible "communities" in a given social network.

Dos and Don'ts!

Please note that:

- For this assignment you can use source code that is available as part of the course material (lectures, exercises, tutes and labs). However, you must properly acknowledge it in your solution.
- All the required code for each part must be in the respective `*.c` file. You may not create additional C files.
- You may implement additional helper functions.
- Helper functions should be declared as static.
- After implementing the functions in `FloydWarshall.h` and `CentralityMeasures.h`, you may use these functions in other tasks.
- You must not modify any `.h` files.
- None of your submitted files should contain a "main" function.
- If you have not implemented any part, you must still submit an empty file with the corresponding file name.

Provided Files

We have provided an implementation of a directed graph ADT that you should use. However, you must not modify `Graph.h` or `Graph.c`. Further, you should only interact with a graph via the functions and data structures in `Graph.h`.

Also note:

- you may assume that all edge weights will be greater than zero.
- you may assume that graphs will not contain parallel edges or self-loops.

Download files:

- ass2.zip - after you have downloaded the files, you should remove `_template` from the name of all `.c` files (the `_template` was added as a measure to prevent you from overwriting your work if you redownload the files). If you are connected to CSE, you can download the files from the command-line using the command:

  ```
  cp /web/cs2521/21T1/assigns/ass2/downloads/ass2.zip .
  ```

- testing.zip - this provides a `Makefile` to compile your code, driver programs to produce test output as well as testing scripts that you can run with commands such as `sh testFloydWarshall.sh 1`. If you are confused by why the expected output is a certain way for a test, you should first read the driver program for the corresponding test (e.g., `testFloydWarshall.c`) before asking on the forum. If you are connected to CSE, you can download the files from the command-line using the command:

  ```
  cp /web/cs2521/21T1/assigns/ass2/downloads/testing.zip .
  ```

  Note: The zip file contains copies of the files provided in `testing.zip`. If you modified any of these files or didn't rename the `_template` files as was advised above, then you should create a backup of these files first.

Part 1: Floyd-Warshall algorithm

In order to discover "influencers", we need to **repeatedly** find shortest paths between **all pairs** of nodes. In this section, you need to implement the **Floyd-Warshall algorithm**, which is a shortest-path algorithm for graphs. Unlike Dijkstra's algorithm, which is a single-source, shortest-path algorithm, the Floyd-Warshall algorithm computes the shortest distances between every pair of vertices in a graph.

The algorithm begins with the following observation: If the shortest path from i to j passes through k, then the partial paths from i to k and k to j must be minimal as well. A similar idea is adopted in the Warshall algorithm (*in Week 7 lecture*) for constructing the transitive closure matrix of a graph.

Consider a graph G with vertices numbered 1 through N (**note:** in the provided files, vertices are numbered from 0 to $N-1$). In the $(k-1)$-th step, let $shortestPath(i, j, k-1)$ be a function that returns the shortest path from i to j that only uses vertices from the set $\{1, 2, \dots, k-1\}$ as intermediate vertices along the way in the graph. In the k-th step, the algorithm will then have to find the shortest paths between all pairs (i, j) using only the vertices from $\{1, 2, \dots, k\}$.

For all pairs of vertices it holds that the shortest path must either only contain vertices in the set $\{1, \dots, k-1\}$, or otherwise must be a path that goes from i to j via vertex k. This implies that in the k-th step, the shortest path from i to j either remains $shortestPath(i, j, k-1)$ or is being improved to $shortestPath(i, k, k-1) + shortestPath(k, j, k-1)$, depending on which of these paths is shorter. Therefore, each shortest path remains the same, or contains the node k whenever it is improved. In each iteration, all pairs of nodes are assigned the cost for the shortest path found so far:

$$shortestPath(i, j, k) = \\ \min[shortestPath(i, j, k-1), shortestPath(i, k, k-1) + shortestPath(k, j, k-1)]$$

This formula is the heart of the Floyd-Warshall algorithm. The algorithm works by first computing $shortestPath(i, j, k)$ for all (i, j) pairs for $k = 1$, then $k = 2$, and so on. This process continues until $k = N$, and we have found the shortest path for all (i, j) pairs using any intermediate vertices.

The Floyd-Warshall algorithm typically only provides the lengths of the shortest paths between all pairs of vertices. We can **construct the shortest path** between any two endpoint vertices i and j by recording the "*next vertex*" of i along the shortest path to j. The idea of recording the "*next vertex*" is similar to the idea of recording the "*predecessor vertex*" in **Dijkstra's Algorithm**. Note that *multiple shortest paths* may exist between two vertices. For simplicity, in this assignment, we will only consider the first found shortest path between two vertices. In the lectures, we will discuss how to construct all of the shortest paths between two vertices in Dijkstra's Algorithm.

Your task: In this section, you need to implement the following file:

- `FloydWarshall.c`

See `FloydWarshall.h` for details on the data structure that needs to be returned. **Note:** if there is no path from vertex v to vertex w, the distance from v to w should be set to infinity, defined in `FloydWarshall.h`, and the *next vertex* of vertex v along the (non-existant) path to w should be set to -1. For all vertices v, the *next vertex* along the path from v to itself should be set to -1.

Part 2: Edge Betweenness Centrality

Centrality measures play a very important role in analysing a social network. For example, vertices with higher "betweenness" measures often correspond to "influencers" in the given social network. In this part you will implement a well known centrality measure, **edge betweenness**, for a given graph.

Edge betweenness is an indicator of highly central edges in networks. Mathematically, betweenness centrality of an edge e is the sum of the fraction of all-pairs shortest paths that pass through e:

$$c_B(e) = \sum_{s,t \in V,\ t \text{ is reachable from } s} \frac{\delta(s, t|e)}{\delta(s, t)}$$

where V is the set of nodes, $\delta(s, t)$ is the number of shortest paths from s to t, and $\delta(s, t|e)$ is the number of those paths passing through edge e.

In this assignment, as we have made an assumption that there is at most one shortest path between each pair of vertices, the formula can be simplified to the following:

$$c_B(e) = \sum_{s,t \in V,\ t \text{ is reachable from } s} \delta(s,t|e)$$

Note that due to our assumption, $\delta(s,t|e)$ will be 1 if the shortest path from s to t passes through e, and 0 otherwise.

Your task: In this section, you need to implement the following file:

- `CentralityMeasures.c`

See `CentralityMeasures.h` for details on the data structure that needs to be returned. It is highly recommended that you use the Floyd-Warshall algorithm from Part 1 to compute the shortest paths between vertices. **Note:** the edge betweenness of a non-existant edge should be set to -1.

Part 3: Discovering Community

Discovering communities in a network, known as community detection/discovery, is a fundamental problem in network science. A network is said to have community structure if the nodes of the network can be easily grouped into sets of nodes such that each set of nodes is densely connected internally.

The **Girvan-Newman algorithm** detects communities by *progressively removing edges* from the original network. The connected components of the remaining network are the communities. The Girvan-Newman algorithm focuses on **edges** that are most likely "**between**" communities. It adopts the "**edge betweenness**" centrality introduced in Part 2. The algorithm's steps for community detection are summarized below:

1. Calculate the edge betweenness of all edges in the network.
2. Remove the edge(s) with the highest edge betweenness.
3. Recalculate the edge betweenness of all edges affected by the removal.
4. Repeat Steps 2 and 3 until no edges remain.

After the removal of each edge, we only have to recalculate the betweennesses of those edges that were affected by the removal, which is at most only those in the same component as the removed edge.

The end result of the Girvan-Newman algorithm is a *dendrogram*. As the Girvan-Newman algorithm runs, the dendrogram is produced from the *top down* (i.e. the network splits up into different communities with the successive removal of links). The leaves of the dendrogram are individual nodes.

Please see the following simple example, it may answer many of your questions!

part-3-example.xlsx (MS Excel file)

Note: If there are multiple edges with the highest edge-betweenness, they should all be removed in the same iteration. Note that, removing these edges may create more than one new connected component. For simplicity, in this assignment, we assume that removing edges during any iteration won't create more than one new component. All the test graphs provided in the Ass2_Testing.zip satisfy this assumption.

Your task: In this section, you need to implement the following file:

- `GirvanNewman.c`

See `GirvanNewman.h` for details on the data structure that needs to be returned. It is highly recommended that you use the edge betweenness algorithm from Part 2 to help you calculate the edge betweenness of each edge.

Assessment Criteria

- Part 1: Floyd-Warshall Algorithm (25%)
- Part 2: Edge Betweenness Centrality (25%)
- Part 3: Discovering Community (30%)
- Style: (20%)

Submission

You need to submit the following files and only these files:

- `FloydWarshall.c`
- `CentralityMeasures.c`
- `GirvanNewman.c`

You can submit from the command-line using the following command:

```
give cs2521 ass2 FloydWarshall.c CentralityMeasures.c GirvanNewman.c
```

or you can submit via WebCMS.

Note: Currently, submission does not perform any dryrun testing. Thus, if you submit now, it is your responsibility to ensure that your code compiles and behaves as expected.

As mentioned earlier, please note that:

- For this assignment you can use source code that is available as part of the course material (lectures, exercises, tutes and labs). However, you must properly acknowledge it in your solution.
- All the required code for each part must be in the respective `*.c` file. You may not create additional C files.
- You may implement additional helper functions.
- Helper functions should be declared as static.
- After implementing the functions in `FloydWarshall.h` and `CentralityMeasures.h`, you may use these functions in other tasks.
- You must not modify any `.h` files.
- None of your submitted files should contain a "main" function.
- If you have not implemented any part, you must still submit an empty file with the corresponding file name.

Plagiarism

This is an individual assignment. Each student will have to develop their own solution without help from other people. You are not permitted to exchange code or pseudocode. If you have questions about the assignment, ask your tutor. All work submitted for assessment must be entirely your own work. We regard unacknowledged copying of material, in whole or part, as an extremely serious offence. For further information, read the Student Conduct.