Assignment 2

Social Network Analysis

Aims

* To implement graph-based data analysis functions to mine a given social network
* To give you further practice with C and data structures (Graph ADT)

Introduction

The main focus of this assignment is to implement graph-based data analysis functions that could be used to identify influencers, followers and communities in a given social network.

Setting Up

Change into the directory you created for the assignment and run the following command:

**unzip** [**/web/cs2521/22T1/ass/ass2/downloads/files.zip**](https://cgi.cse.unsw.edu.au/~cs2521/22T1/ass/ass2/downloads/files.zip)

If you're working at home, download files.zip by clicking on the above link and then unzip the downloaded file.

This bundle of files includes the files you need to implement, as well as a few completed ADTs that you may use, some testing programs/scripts and sample graphs.

We have provided complete implementations of the Graph and Priority Queue ADTs. You may use these ADTs for any parts of the assignment as long as you do not modify them.

Part 1 - Dijkstra's Algorithm

In order to discover influencers, we need to repeatedly find shortest paths between pairs of nodes. Your task is to implement a variant Dijkstra's algorithm to discover the shortest paths from a given source node to all other nodes in the graph. The algorithm has **one important additional feature**: if there are multiple shortest paths from a source node to another node, it keeps track of all of them by allowing each node to have multiple predecessors. In the code, this is achieved by each node having a linked list of predecessors (see Dijkstra.h). Each predecessor list must be in ascending order.

In the following example, while discovering shortest paths from node 0, we find that there are two possible shortest paths from node 0 to node 1 (0 -> 1 and 0 -> 2 -> 1), so node 1 has two possible predecessors: node 0 and node 2, as shown below.

|  |  |
| --- | --- |
| Diagram, schematic  Description automatically generated | Source node: 0  Distances  0: 0  1: 2  2: 1  Predecessors  0: NULL  1: [0] -> [2] -> NULL  2: [0] -> NULL  Source node: 1  Distances  0: 2  1: 0  2: 3  Predecessors  0: [1] -> NULL  1: NULL  2: [0] -> NULL  Source node: 2  Distances  0: 3  1: 1  2: 0  Predecessors  0: [1] -> NULL  1: [2] -> NULL  2: NULL |

What you need to do:

Complete the file [Dijkstra.c](https://cgi.cse.unsw.edu.au/~cs2521/22T1/ass/ass2/files/Dijkstra.c) that implements all the functions declared in [Dijkstra.h](https://cgi.cse.unsw.edu.au/~cs2521/22T1/ass/ass2/files/Dijkstra.h).

Part 2 - Centrality Measures for Social Network Analysis

Centrality measures play a very important role in analysing a social network. For example, nodes with higher betweenness often correspond to influencers in a social network. Your task is to implement two well-known centrality measures for a given directed weighted graph.

Closeness Centrality

The closeness centrality of a node x is calculated as the reciprocal of the sum of the lengths of the shortest paths between node x and all other nodes y (y≠x) in the graph. Generally closeness is defined as below:

C(x)=1∑yd(y,x)

where d(y,x) is the shortest distance between vertices x and y.

However, considering we may have more than one connected components, **for this assignment** you need to use the **Wasserman and Faust formula** to calculate the closeness of a node in a directed graph as described below:

CWF(u)=n−1N−1∗n−1∑all v reachable from ud(u,v)

where d(u,v) is the shortest-path distance in a directed graph from vertex u to v, n is the number of nodes reachable from u (which includes u itself), and N denotes the number of nodes in the graph.

The Wasserman and Faust formula is useful for graphs with more than one connected components. However, if a node is not connected to any other node (isolated), its closeness value CWF should be 0.

Betweenness Centrality

The betweenness centrality of a node v is given by the expression:

g(v)=∑s≠v≠t≠s, and t is reachable from sσst(v)σst

where σst is the total number of shortest paths from node  s to node  t and σst(v) is the number of those paths that pass through v.

What you need to do:

Complete the file [CentralityMeasures.c](https://cgi.cse.unsw.edu.au/~cs2521/22T1/ass/ass2/files/CentralityMeasures.c) that implements all the functions declared in [CentralityMeasures.h](https://cgi.cse.unsw.edu.au/~cs2521/22T1/ass/ass2/files/CentralityMeasures.h). You are encouraged to use the Dijkstra API you implemented in Part 1. You may assume that all given graphs will contain at least three vertices.

For a more detailed explanation of closeness and betweenness centrality, please refer to: [Explanations for Part 2](https://cgi.cse.unsw.edu.au/~cs2521/22T1/ass/ass2/part-2-explanations.html)

Part 3 - Discovering Community

Your task is to implement the Hierarchical Agglomerative Clustering (HAC) algorithm to discover communities in a given graph. In particular, you need to implement the Lance-Williams algorithm, as described below. In the lecture we will discuss how this algorithm works, and what you need to do to implement it. You may find the following resources useful for this part:

* [Hierarchical Clustering](https://en.wikipedia.org/wiki/Hierarchical_clustering) (Wikipedia), for this assignment we are only interested in the "agglomerative" approach.
* Three videos by Victor Lavrenko, watch in order:
  + [Agglomerative Clustering: how it works](https://www.youtube.com/watch?v=XJ3194AmH40)
  + [Hierarchical Clustering 3: single-link vs. complete-link](https://www.youtube.com/watch?v=VMyXc3SiEqs)
  + [Hierarchical Clustering 4: the **Lance-Williams algorithm**](https://www.youtube.com/watch?v=aXsaFNVzzfI)

You need to use the following (adapted) Lance-Williams HAC algorithm to derive a dendrogram:

* Calculate distances between each pair of vertices as described below.
* Create clusters for every vertex i, say ci. Every vertex begins in its own cluster (but they will be merged later).
* Let Dist(ci,cj) represent the distance between cluster ci and cj. Initially, it represents the distance between vertex i and j (since initially each vertex is in its own cluster).
* Until there is one cluster remaining:
  + Find the pair of clusters, ci and cj, with the smallest distance between them. If there are multiple alternatives, you can select any one of the pairs of closest clusters.
  + Remove the clusters ci and cj from the collection of clusters and add a new cluster cij (that contains all the vertices in ci and cj) to the collection of clusters.
  + Update the dendrogram to reflect the merging of ci and cj.
  + Calculate the distances Dist(cij,ck) between the newly added cluster cij and each other cluster ck in the collection using the Lance-Williams formula using the selected method ('*Single linkage*' or '*Complete linkage*' - see below).
* Return the dendrogram

Distance Measure

For this assignment, we define the distance between a pair of vertices as follows: Let wt represent the maximum edge weight of all available weighted edges between a pair of vertices v and w. The distance d between vertices v and w is defined as d=1/wt. If v and w are not connected by an edge, d is infinity (you may use DBL\_MAX to represent infinity).

For example, if there is one directed link between v and w with weight wt, the distance between them is 1/wt. If there are two links between v and w, we take the maximum of the two weights and the distance between them is 1/max(wtvw,wtwv). Please note that in reality one can also consider alternative approaches, such as taking the average, minimum, etc. However, we need to pick one approach for this assignment and we will use the above distance measure.

Lance-Williams Formula

The general Lance-Williams formula is:

Dist(cij,ck)=αi∗Dist(ci,ck)+αj∗Dist(cj,ck)+β∗Dist(ci,cj)+γ∗abs(Dist(ci,ck)−Dist(cj,ck))

where αi, αj, β, and γ are determined by the chosen agglomerative criterion (i.e., single linkage, complete linkage, etc.).

However, given a specific agglomerative criterion, we can simplify the formula as follows:

For the *single linkage method*, the formula can be simplified to:

Dist(cij,ck)=min(Dist(ci,ck),Dist(cj,ck))

For the *complete linkage method*, the formula can be simplified to:

Dist(cij,ck)=max(Dist(ci,ck),Dist(cj,ck))

What you need to do:

Complete the file [LanceWilliamsHAC.c](https://cgi.cse.unsw.edu.au/~cs2521/22T1/ass/ass2/files/LanceWilliamsHAC.c) that implements all the functions declared in [LanceWilliamsHAC.h](https://cgi.cse.unsw.edu.au/~cs2521/22T1/ass/ass2/files/LanceWilliamsHAC.h).

For a simple demonstration of the Lance-Williams algorithm, please refer to: [Part 3 Simple Example](https://cgi.cse.unsw.edu.au/~cs2521/22T1/ass/ass2/downloads/part-3-example.xlsx) (MS Excel file)

Note: Do not use the BSTree ADT for this task - the BSTree ADT is intended to be used by the testing program only.

Testing

We have provided some testing programs/scripts and sample graphs for you to get started. However, please note:

* You need to add more advanced test cases (graphs) to properly test your implementations.
* We will use more advanced test cases that are not included in the test cases provided to you when automarking your code.

Graphs

The sample graphs are in the graphs/ directory. Each graph file consists of an integer representing the number of vertices, followed by a series of directed edges (one per line), where each edge is represented by three comma-separated values: (1) the source node, (2) the target node, (3) the edge weight. You can create your own graphs for testing by following this format.

Part 1

You can use the following commands to test your Dijkstra API. All graphs in the graphs/ directory are applicable for testing this part.

**make** # compiles the program

**./testDijkstra *graph-file*** # tests with a specific graph, outputs to terminal

# now manually compare with expected output in DijkstraTests/

**./testDijkstra graphs/1.in** # for example, tests with the graph in graphs/1.in

**sh testDijkstra.sh *graph-number*** # runs a specific test

**sh testDijkstra.sh 2** # for example, runs test for graph 2

**sh testDijkstra.sh** # runs all provided tests

Part 2

You can use the following commands to test your CentralityMeasures API. All graphs in the graphs/ directory are applicable for testing this part.

# compiles the program

**make**

# tests with a specific graph and centrality type, outputs to terminal

**./testCentralityMeasures *graph-file* *centrality-type***

# for example, tests with the graph in graphs/1.in

**./testCentralityMeasures graphs/1.in c** # closeness centrality

**./testCentralityMeasures graphs/1.in b** # betweenness centrality

# runs a specific test

**sh testCentralityMeasures.sh *graph-number* *centrality-type***

# for example, runs test for graph 2

**sh testCentralityMeasures.sh 2 c** # closeness centrality

**sh testCentralityMeasures.sh 2 b** # betweenness centrality

# for example, runs all tests for graph 2

**sh testCentralityMeasures.sh 2**

# runs all provided tests

**sh testCentralityMeasures.sh**

Part 3

You can use the following commands to test your LanceWilliamsHAC API. The graphs applicable for testing Part 3 are: graphs/1.in, graphs/2.in, graphs/3.in, graphs/4.in.

**make** # compiles the program

**./testLanceWilliamsHAC *graph-file*** # tests with a specific graph

**sh testLanceWilliamsHAC.sh *graph-number*** # runs a specific test

**sh testLanceWilliamsHAC.sh 2** # for example, runs test for graph 2

**sh testLanceWilliamsHAC.sh** # runs all provided tests

Note that the testLanceWilliamsHAC program may produce different output to the .out files produced by the testLanceWilliamsHAC.sh script. This is because the testing script *sorts* all the lines of the output before writing it to the .out files. If your program produces debugging output then you should test with the testLanceWilliamsHAC program and not the testing script.

Note: We have only provided expected output for the single linkage method. You are expected to test and check the output for complete linkage yourself. (Hint: modify testLanceWilliamsHAC.c to test complete linkage.)

How to interpret the output of testLanceWilliamsHAC.sh

The testing script produces output like this in the .out files:

0: {0, 1, 2}

1: {0} (leaf)

1: {1, 2}

2: {1} (leaf)

2: {2} (leaf)

Each line of the output describes a node in the dendrogram. The number to the left of the colon is the level that the node appears on (the level of the root node is 0). The set to the right of the colon lists the vertices that are contained within that sub-dendrogram (sub-tree). Leaf nodes are indicated by (leaf) at the end of the line. The above output corresponds to the dendrogram:

However, since the order of the lines does not matter, it can also correspond to any of these dendrograms:

Note that all of these dendrograms are essentially the same - in all of them, vertex 0 is on level 1, vertices 1 and 2 are on level 2, there is a cluster containing the vertices 1 and 2, and there is a cluster containing all the vertices. It doesn't matter which one of these dendrograms you produce - the testing script sorts the output lines, so all correct dendrograms will result in the same output.

Assumptions/Clarifications/Notes

* In a graph, vertices are numbered 0 to N−1, where N is the number of vertices.
* All edge weights will be positive.
* Graphs will not contain [parallel edges](https://en.wikipedia.org/wiki/Multiple_edges) or self-loop edges.
* All input graphs will be valid.
* You may not #include any additional libraries, as all the libraries you require are already included.
* You are not required to use the priority queue ADT, but it is expected to make Task 1 easier.
* You can assume that the graphs used to test complete linkage in Part 3 will be complete. That is, there will always be at least one edge between every pair of nodes.

Frequently Asked Questions

* **Are we allowed to create our own functions?** Of course...
* **Are we allowed to create our #defines and structs?** Yes.
* **Are we allowed to change the signatures of the given functions?** No. If you change these, your code won't compile and we won't be able to test it.
* **What is the difference between a "node" and a "vertex"?** Nodes and vertices are the same thing.
* **How do we return an array?** To return a new array from a function, the array must be malloc'd. Otherwise, the array will go out of scope (i.e., no longer exist) once the function returns.
* **What errors do we need to handle?** You should handle common errors such as NULL returned from malloc by printing an error message to stderr and terminating the program. You are not required to handle other errors.
* **The explanation of Part 3 in the Excel file has sets as indices for the dist and dendrogram arrays.** Those aren't indices - they're clusters. When you implement your solution, you will need to find a way to associate each cluster with an array index.

Submission

You need to submit the following files:

* Dijkstra.c
* CentralityMeasures.c
* LanceWilliamsHAC.c

**You must submit all of these files, even if you did not complete all of the tasks.** No supporting files are permitted in this assignment. This means all your code must be in these three files.

You can submit via the command line using the give command:

**give cs2521 ass2 Dijkstra.c CentralityMeasures.c LanceWilliamsHAC.c**

or you can submit via WebCMS. You may not submit any other files. You can submit multiple times. Only your last submission will be marked. You can check the files you have submitted [here](https://www.cse.unsw.edu.au/~cs2521/22T1/view/main.cgi/).

After you submit, you **must** check that your submission was successful by going to your [submissions page](https://www.cse.unsw.edu.au/~cs2521/22T1/view/main.cgi/). Check that the timestamp is correct. If your submission does not appear under Last Submission or the timestamp is not correct, then resubmit.

Compilation

You must ensure that your final submission compiles on CSE machines. Submitting non-compiling code leads to extra administrative overhead and will result in a 10% penalty.

Every time you make a submission, a dryrun test will be run on your code to check that it compiles. Please ensure that your final submission successfully compiles, even for parts that you have not completed.

Assessment Criteria

This assignment will contribute 20% to your final mark.

Correctness

80% of the marks for this assignment will be based on the correctness of your code, and will be based on autotesting. We will test your program using the same testing programs that we have given you (testDijkstra, testCentralityMeasures and testLanceWilliamsHAC), but we will use different graphs from those provided to you.

Marks for correctness will be distributed as follows:

|  |  |  |
| --- | --- | --- |
| Part 1 | Dijkstra's algorithm | 30% of the correctness mark |
| Part 2 | Closeness centrality | 20% of the correctness mark |
| Betweenness centrality | 30% of the correctness mark |
| Part 3 | Discovering community | 20% of the correctness mark |

Efficiency

There is no strict efficiency requirement, but tests that take too long to run (i.e., 5 real time seconds on a single execution of one of the testing programs without valgrind) will be terminated automatically and you will receive 0 for the test. The graphs we use during testing will be about the same size as those provided to you.

Memory errors/leaks

Additionally, you must ensure that your code does not contain memory errors or leaks, as code that contains memory errors is unsafe and it is bad practice to leave memory leaks in your program. Note that our tests will always free the memory associated with the returned arrays/structures from your functions by calling freeNodeData or free. See the test programs for more details.

Submissions that contain memory errors or leaks will receive a penalty of 10% of the correctness mark, deducted from the attained mark. Specifically, there will be a separate memory error/leak check for each part, and the penalty will be 10% of the marks for that part. You can check for memory errors and leaks with valgrind. For example:

**valgrind -q --leak-check=full ./testDijkstra *graph-file* > /dev/null**

A program that contains no memory errors or leaks will produce no output from this command. You can learn more about memory errors and leaks in the [Debugging with GDB and Valgrind](https://cgi.cse.unsw.edu.au/~cs2521/22T1/lab/11/questions) lab exercise.

Note that you should not run valgrind on testDijkstra.sh or any other files ending with .sh. valgrind can only be run on C executables.

Style

20% of the marks for this assignment will come from hand marking of the readability of the code you have written. These marks will be awarded on the basis of clarity, commenting, elegance and style. The following is an indicative list of characteristics that will be assessed, though your program will be assessed wholistically so other characteristics may be assessed too (see the [style guide](https://cgi.cse.unsw.edu.au/~cs2521/22T1/resources/style_guide.html) for more details):

* Consistent and sensible indentation and spacing
* Using blank lines and whitespace
* Using functions to avoid repeating code
* Decomposing code into functions and not having overly long functions
* Using comments effectively and not leaving planning or debugging comments

The course staff may vary the assessment scheme after inspecting the assignment submissions but it will remain broadly similar to the description above.