

Project #4 – Graph Implementation

Learning Objectives

- Demonstrate an understanding of both an adjacency list and adjacency matrix graph representation by porting one to the other
- Demonstrate an understanding of common graph-based algorithms such as BFS, DFS, Dijkstra's algorithm and Prim's algorithm
- Demonstrate an ability to work with graphs by expanding an existing graph implementation to provide additional functionality

Overview

Your task for this assignment is to modify the graph data structure we have been implementing in lecture to use an adjacency matrix instead of an adjacency list and extend the implementation to support additional functionality.

Ground Rules

For this project you should begin with the graph-related code I have posted to Pilot (vertex, graph, and graph_driver). To complete the assignment you may use any classes from the C++ Standard Template Library (STL), including vector, queue, set, unordered_map, etc., in your implementation. You may NOT use significant portions of any code or libraries from any other source (i.e. other than mine and the C++ STL). If you have any questions about what is allowable, ASK ME rather than risk an academic integrity violation.

Requirements

0. Change the existing code to use an adjacency matrix for the representation instead of an adjacency list.

The code's end functionality should not be changed by this, so you can test your work by running the existing implementation with the current graph_driver.cpp code and then using your implementation and checking that it produces the same (or equivalent) results.

1. Add a new constructor to your graph class that takes a filename and a boolean indicating whether the graph is directed or not.

The constructor should read in the file and create a graph to represent it. Data files will be organized as shown below. If the graph is directed, the first v_label of an edge should be considered the "from" vertex and the second one the "to" vertex. There is a sample on Pilot called mine.graph, which should be considered a weighted directed graph.

```
Vertices
v1_label
v2_label
...
Edges
e1_label_1 e1_label_2 e1_weight
```

```
e2_label_1 e2_label_2 e2_weight
...
```

You can assume that all labels are characters and all weights are integers. You cannot assume that the graph will be a single connected component!

3. Add the following methods related to vertex degree to your graph implementation:

```
int getInDegree(char vertexLabel)

int getOutDegree(char vertexLabel)

vector<Vertex*> getMaxInDegree(int& maxDegree)

vector<Vertex*> getMaxOutDegree(int& maxDegree)
```

A vertex's out degree is the number of edges that have this vertex as a starting point. Similarly, a vertex's in degree is the number of edges that have this vertex as an end point. For undirected graphs, every vertex's in degree is the same as its out degree. These methods should not consider an edge's weight when computing degree.

For example in `mine.graph`, `getInDegree('A')` should return 1 and `getOutDegree('A')` should return 4.

The `getMaxInDegree` and `getMaxOutDegree` methods should return a pointer to the vertex in the graph that has the highest in/out degree (again, ignoring edge weights). If two or more vertices "tie", then the methods should return pointers to all of the tying vertices (this is the reason the method's return type is a vector rather than a single vertex pointer). These methods should also fill in the pass-by-reference int parameter `maxDegree` with the maximum in/out degree.

For example, in `mine.graph`, `getMaxOutDegree` returns a vector with just a pointer to the 'A' vertex in it and it sets `maxDegree` to 4, and `getMaxInDegree` should return a vector with just a pointer to the 'E' vertex with `maxDegree` set to 4.

4. Add the following method related to neighborhoods to your graph implementation:

```
vector<Vertex*> getNeighborhood(char vertexLabel, int neighborhoodSize)
```

A vertex's neighborhood is the set of vertices reachable from a starting vertex via paths of length less than or equal to `neighborhoodSize`. The order in which you list the neighbors is not important. Be sure to include the starting node itself as part of the neighborhood. Do not list the same vertex label multiple times.

For example, `getNeighborhood('H', 2)` should return a vector containing pointers to the vertices with labels H D F

5. Add the following method related to connected components to your graph implementation:

```
vector<Vertex*> getLargestConnectedComponent()
```

In mine.graph the largest connected component comprises the vertices with the labels A B C E G I J . If two or more connected components “tie” for the largest, you can return any of them.

Turn in and Grading

Turn in any source code files needed for your program. If you used one or more of my files as-is, upload them as part of your submission. You do not need to submit any driver code. You may upload the files individually or in a zip file – do not submit your code in a text file, PDF, or Word doc.

Your project will be graded according to the following rubric. Projects that don’t compile will receive a zero.

[30 pts] The graph uses an adjacency matrix instead of an adjacency list

[10 pts] The new constructor is implemented as specified

[10 pts] The `getInDegree` and `getOutDegree` methods work for both directed and undirected graphs.

[10 pts] The `getMaxInDegree` and `getMaxOutDegree` methods work for both directed and undirected graphs. If two or more vertices “tie” for the maximum in/out degree, all of them are returned. The `maxDegree` parameter is correctly filled in.

[20 pts] The `getNeighborhood` method works for both directed and undirected graphs.

[20 pts] The `getLargestConnectedComponent` method works for both directed and undirected graphs.