Drew Aaron

Michael Beaver

Andrew Hamilton

Assignment 13, Group 4

CS 355

December 12, 2012

**Final System Design**

This system aims to establish a back-end system that may be interacted with by any reasonable front-end system. The user should be able to generate and modify graphs without having to interact with each component (e.g., nodes). The system should be such that the user may perform any graph-related commands by interfacing with the back-end system through the Graph class. By limiting the user’s scope of interaction, system damage and misuse may be minimized. The Graph class’s accessor and mutator methods act as a veritable bottleneck so the user may only interact with the Graph instead of its component parts. Thus, the front-end system may be written in any way; any front-end may be used with the back-end.

The Graph class is the heart of the back-end of the system. Following object-oriented principles, the Graph class is broken into its component parts, namely Edges and Nodes.[[1]](#footnote-1) The Graph class also has a minimum-priority queue (an implicit binary heap)[[2]](#footnote-2) for Dijkstra’s Algorithm with weighted shortest paths. Edges are composed of two Node pointers: the source Node and the adjacent Node. Hence, the Edge is a link between a Node and its adjacent Node. The Node class is the most elementary object in the Graph, and it holds any identifying information about the Node, such as its name. In addition, each Node has an adjacency list (a vector of Edges) for easy access. Likewise, the Graph class contains a vector of Node pointers. This vector is essentially the memory representation of the Graph object. To allow for generality, the Graph’s Edge type, Node type, and Edge weight units may be customized by the user from the front-end. For example, the user may wish for the Nodes to represent city names. Below is a visual model of the back-end system design:[[3]](#footnote-3)

Node

\* Node name

\* Adjacency list

Edge

\* Source and Adjacent Nodes

\* Edge weight/cost

Binary Heap

\* Vector of Node pointers (heap)

\* popMin()

Graph

\* Vector of Node pointers

\* Graph units

\* Interface methods

*composition*

Ideally, the user should only ever have to interact with the Graph class. To accommodate this design paradigm, the Graph class supports full file input/output operations. The system may import and/or export the user’s graph to a file. This exported file may also be imported, which allows for greater flexibility with file operations. The files follow a specific format (see the example below). The format of the Node adjacencies (i.e., Edges) in the input file is strict and simple to read. There may be any number of adjacencies in the Graph; however, for Dijkstra’s Algorithm to be valid, the Graph must be acyclic and directed. The comments in this sample file would not be present in a real file.

4 ; Total number of Nodes

Streets ; Edge type

Cities ; Node type

Miles ; Edge Weight units

Atlanta ; Node name

New York City ; Node name

Miami ; Node name

San Francisco ; Node name

Atlanta->Miami(431) ; Adjacency: Source->Adjacent(Weight)

Miami->San Francisco(925) ; Adjacency

Miami->New York City(834) ; Adjacency

New York City->San Francisco(1032) ; Adjacency

Atlanta->New York City(653) ; Adjacency

San Francisco->Atlanta(856) ; Adjacency

In addition, the Graph class also allows manual insertion from the front-end. A user may insert a single Node, an adjacency with a weight greater than one, or an adjacency with a weight of one. The front-end user may determine the insertion format representation. The back-end system merely handles how the inserted Node or Nodes are represented in memory.

The back-end system allows the user to find the weighted shortest path from one Node to another Node in the Graph. The user, from the front-end, only needs to pass in two Node pointers to the dijkstraHeap[[4]](#footnote-4) method. This method checks if the Nodes are the same, if they actually exist, if they are isolated, and if the path is valid. The method follows Dijkstra’s Algorithm to determine the weighted shortest path. The algorithm uses a minimum-priority queue (binary heap) to store “visited” Nodes as the algorithm traverses the Graph. As per Dijkstra’s Algorithm, each “eyeball” is popped off the front of the queue. If the eyeball, for some reason, does not change during the algorithm’s progression, then the path is invalid. For example, this may occur if the user attempts to find a path that does not exist. Once the weighted shortest path is determined, the path and path data are printed to the screen for the user. The user should reset the Graph by calling resetGraph[[5]](#footnote-5) before attempting to find another path. The user may completely clear (i.e., delete all Nodes and Edges) the Graph by calling clearGraph.[[6]](#footnote-6)

This system is designed for maximized portability and generality. In theory, any reasonable front-end system could be tied to this back-end system. The front-end user needs to only interact with a Graph object or objects. The user has full power to modify the Graph’s contents and its data representation (e.g., Edge weight units). This system also employs a minimum-priority queue implementation of Dijkstra’s Algorithm. The back-end system is generalized; the user may modify the Graph from the front-end system. Therefore, this system design is an optimal design for any practical simulation or application of Dijkstra’s Algorithm.

1. The initial Graph class design was inspired by a post on www.cplusplus.com (http://www.cplusplus.com/forum/general/371/#msg1238). This initial design underwent substantial revision and modification to suit our design purposes. [↑](#footnote-ref-1)
2. Parts of the binary heap implementation were inspired by and adapted from an article on www.algolist.net (http://www.algolist.net/Data\_structures/Priority\_queue). [↑](#footnote-ref-2)
3. Only relevant items are listed for each class. [↑](#footnote-ref-3)
4. This method would be called from a Graph object. For example: myGraph.dijkstraHeap(&start, &end); [↑](#footnote-ref-4)
5. This method would be called from a Graph object. For example: myGraph.resetGraph(); [↑](#footnote-ref-5)
6. This method would be called from a Graph object. For example: myGraph.clearGraph(); [↑](#footnote-ref-6)