Dijkstra’s Algorithm

**Due: May 4**

**Objectives:**

* Read, use and develop established code
* Get familiar with Min-Heap
* Get familiar with graph representation and Dijkstra’s algorithm

**Overview**

For our final programming assignment, we will look at the problem of pathing, also referred to as pathfinding: <https://en.wikipedia.org/wiki/Pathfinding>. In short, the goal of pathfinding is to locate an optimal path between two nodes on a graph.

## Pathfinding and Graphs

Pathing and graphs are intrinsically linked because graphs are used to model so many kinds of problems. Lists and trees can both be modeled using graphs, and graphs have many real-world applications like road layout for GPS, network topology for internet traffic, airport flight planning, flow of information between neurons in the brain, mutual connections, and relationships between nodes in social networks and search engines, and more. Graphs are the capstone of this class and will appear in future CS classes.

## Dijkstra’s Algorithm

The best-known algorithm for finding the shortest path between two vertices of a graph is *Dijkstra’s Algorithm*: <https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm>. This algorithm is distinct from the traversal methods we discussed (BFS and DFS) in that it makes determinations based on cost or weight of edges. A simple example is flight routing between airports; the edges could represent the distance in miles.

**Getting Started**

1. Download the [starter code](https://drive.google.com/drive/folders/1_P0NR3N2IMjEPN_H83E1mwmHAd4SrKbo?usp=share_link).
   1. The following files are fully implemented and will not need to be edited:
      1. LocatorHeap.h - a min-heap implementation with locators for direct access
   2. Both Graph.h and main.cpp have skeleton code.
2. I recommend implementing the graph structure first. In this assignment, you’ll use an adjacency list, but this version is slightly different from the one in class. Here, you save vertices as objects which contain a list of edge pointers. For simplicity, you can use the graph structure to hold both the vertices and edges to make dynamic memory operations easier.
3. The core task is to implement Dijkstra’s algorithm. The algorithm returns the shortest path as a list, starting with the start node and printing all the nodes to take until reaching the end node.
4. You will need to finish the main.cpp file to read the graph files.
   1. The files are written with the first line giving the number of vertices followed by the number of edges.
   2. The next lines are the edges themselves, along with their weight.
      1. **Note**: In this assignment, all graphs are undirected (bidirectional), so you should add bidirectional edges for each edge you’re given!
   3. The last line will be the start and end node between which to find the shortest path using Dijkstra’s algorithm.
5. Several tests are given and the output is in the file “\*\*\*\_output.txt”. You can use them to test your code. **test\_mem.cpp** is given to detect memory leaks.

**Task Breakdown**

### Graph.h

#### Vertex and Edge

Graph.h provides simple vertex and edge implementations. The final graph structure is going to be similar to the adjacency list you saw in class, we will store a vector of Vertex pointers. The Vertex objects contain vectors of their edges. This will yield a slightly more efficient adjacency list.

Additionally, we have simplified the path finding problem by saving space in the Vertex object to hold information like “visited” and “distanceTo”. You are free to ignore these if you choose. Having the ability to store them with a vertex can save the effort of putting them elsewhere, but you can certainly use auxiliary data structures to hold such information.

As further practice of overloading operators, you should implement the operator< to compare vertices. This is being included for pathfinding, so you will use the “distanceTo” for comparison.

|  |  |
| --- | --- |
| **Vertex and Edge** | **Description** |
| bool operator< (const Vertex &v) | This comparison will be used to determine the relative distance of two vertices. You can overload the operator by just returning the result of comparing the distance members. |

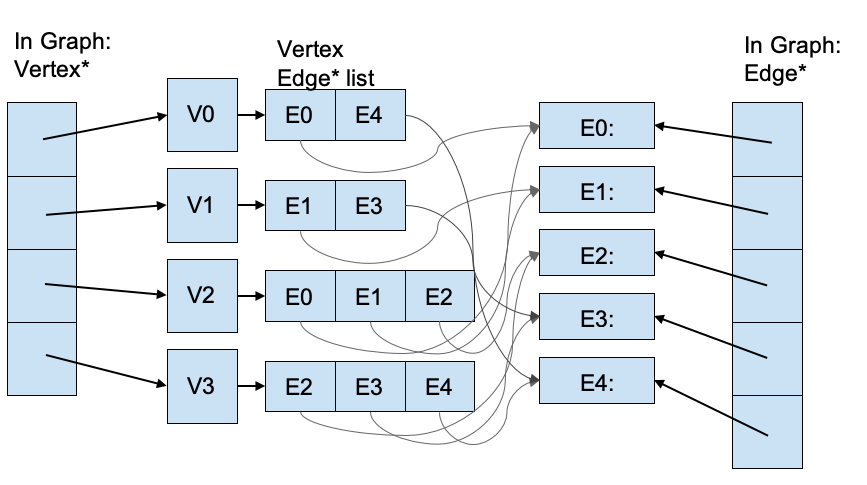
#### Graph

##### Data Members

The graph itself will save a list of Vertex and Edge pointers. Again, this is similar to the adjacency list you saw in class but with actual objects to hold the information.

The advantage of having lists (vectors) to hold the vertex and edge pointers at the graph level is that it moves the dynamic memory operations into the Graph. For example, you can use these members to clear all the dynamic memory in the Graph destructor.

The following image shows an illustration of how the information relates to each other in this Graph structure. The left-hand side represents a list or vector of Vertex\*, and the right-hand side has the corresponding Edge\* list. These are the only two members in the Graph. Everything else is stored in memory and just associated with each other through pointers.



##### Rule of Three

The Graph destructor should clear all vertices and edges associated with the graph. Also, to further practice “rule of 3” concepts, you should implement the copy constructor and copy assignment operator. It may be useful to create helper functions for clearing and copying a graph.

##### Inserting Vertices and Edges

When given a new vertex label, create the new Vertex object and push back the pointer into the graph’s vertex list. Note that we are using **integer labels** for ease; this allows all vertices to be indexed directly using their labels.

When given the information for an edge, create a new edge going from vertex l1 to l2 with the given weight . This should be pushed back onto the edge list of the graph. Also, you should push the edge into the Edge\* list belonging to the vertex.

All edges will be bidirectional / unidirectional in this assignment. You can just add the single edge in this function and in your main, make sure to call it with the opposite direction later when reading the file.

##### Finding Shortest Path

The shortestPath() function takes the labels of a starting node and an ending node. It will use Dijkstra’s algorithm to compute the shortest path and return the sequence of nodes (a vector of Vertex\*) which should be traversed from start to end.

We will discuss more about Dijkstra’s algorithm in lecture, including pseudocode, but the idea of the algorithm is very straightforward.

1. First, you should set all the vertex visited values to false, clear the pathTo, and reset the distanceTo to “infinite” (use the max value that can be stored in a float). Alternatively, you can create auxiliary structures to track some or all of this information. A vector of booleans can work well to track visited nodes in this example since all labels are integer-based.
2. Next, set the start node as the current node and set its distance to 0.
3. Use a priority queue to store all the nodes. A heap-based priority queue is provided in LocatorHeap.h. This is a min-heap where you can save Vertex pointers which can be compared using the ‘<’ operator you defined. The heap is important in selecting the next node to visit.
   1. **Important note**: This is a locator heap. We touched on the concept in lecture, even though we have not used one in practice. When we add elements, a locator heap returns a reference (pointer) to that element. You can use an auxiliary structure like a vector to save the locators, or store them inside the vertex object. These allow direct access to elements inside the heap and will be useful for efficiently updating distances of vertices during the shortest path calculation.
4. Remove the minimum element from the heap and check the distance to all neighbors. The edge list is useful here. If you find the current path yields a distance lower than the one already saved for a vertex, update the value for the distance in the vertex and heap.
   1. Removing the minimum element at each step makes Dijkstra’s algorithm a *greedy* algorithm in which it selects the locally optimal choice. There are other ways to do the selection, and a variety of algorithms that build on Dijkstra’s add heuristic information to enhance the result. More pathing algorithms will be seen in lecture and lab.
5. Mark the current node visited.
6. Repeat steps 4 & 5 until the end node is marked visited.

|  |  |
| --- | --- |
| **Graph** | **Description** |
| Data member to hold the Vertex\* and Edge\* for the graph | We’ll hold pointers so that it is easy to store the same object in multiple places, such as the Vertex edge list, the locator heap, or any other auxiliary data structure you use. |
| Rule of Three functions: - destructor (declaration in starter)  - copy constructor  - copy assignment | Vertex and Edge creation is handled by the graph, so you’ll want to follow the rule of 3 guidelines for supporting dynamic memory operations. This means there should be a destructor to free memory, as well as copy constructor and copy assignment for deep copies of graphs. |
| insertVertex(int)  insertEdge(int,int,float) | You only need to implement the insert operations for the graph structure. This consists of allocating a new vertex with the given integer label and saving it in the graph list of vertices. For inserting an edge, allocate the new edge and add it to both the graph list of edge pointers and the edge list for the originating vertex.  Also, note you are only implementing insertion; we are not testing removal for this graph. |
| vector<Vertex\*> shortestPath (int start, int end); | The shortestPath function should implement Dijkstra’s algorithm, which is described above, to return a list of vertices (vector of Vertex\* type) to traverse along the optimal path to the end node. |

### main.cpp

The main file has several snippets that need to be filled in, and the starter code provides comments for these segments.

1. You should read in the user-provided filename from *cin* and use fstream to open the file.
2. The file is organized into 3 segments:

* The first line is the number of vertices and then number of edges.
* The middle portion is the collection of edges, with each line representing the node labels and the weight of an edge.
  + **Note:** you are creating a unidirectional graph, so make sure to insert the bidirectional version of each edge into your graph!
* The last line is the start and end node to find the shortest path between.

An example file and possible representation of its graph is shown below. The file starts by identifying there are 4 vertices and 5 edges. The next 5 lines define the edges. The last line specifies that the goal is to find the shortest path between nodes 0 and 3. In this case, that would be 0, 2, 1, and 3 for a total cost of 6.



At the bottom of main.cpp you’ll see a for loop that looks slightly different than ones we’ve used before. This is a “for-each” loop. Just as we upgraded from simple counting loops to iterators that gave us pointers, we can upgrade further to “for-each” logic. These coding patterns use iterators to traverse an object. You get the dereferenced item itself, making it even easier to traverse without the hassle of specifying starting and ending conditions.

**Submitting**

After successful testing of your project, simply zip all .h and .cpp file(s) together and upload in Canvas.

