CSCI 222

Spring 2019

Lab #5

For this lab assignment, you will build a Google map version of the Palomar campus. You will be given a shapefile of the Palomar campus from which you will use the tools provided in OpenCV to read into a shapefile class.

Part 1 of this lab will apply template classes, template functions, inheritance, and STL vector concepts to build a weighted undirected graph data structure from the shapefile and allow the user to find the shortest path to take to get between any two points on campus. Your task will be to build a undirected template class inherited from a generalized graph template. Provide the necessary member data and member functions (see description of the graph data structure below) to perform the necessary operations. For finding the shortest path, you will use Dijkstra algorithm (see description below).

Part 1 of this lab will expand the map area to include part of the city of San Marcos. Your task in this lab is to allow the user to input some of his/her favorite restaurants and hangout locations and will make suggestions of places to go to based on his/her given location by searching for the closest venues within some user specified radius (e.g., within 5 miles radius). For this part of the lab, you will also be given a shapefile from which to build your graph from.

Shortest Path Calculation Using Graph

Graph objects are widely used in the fields of computer science, chemistry, and mathematics. They are native to real world applications dealing with the general notion of connectivity or routing. These applications are wide and varied and range from the areas of computer network design and electronic circuit layout and verification to airline flight scheduling and social planning. For example, in the field of electronic circuit design, one may wish to query a system to ask if all the circuit connections are in place. In the area of airline routine, one may want to know what is the number of stops a plane must make between San Francisco and Washington DC, and thus what the shortest flight path would be.

Defintion: A graph G is defined to be a collection (union) of a set V, of vertices (or nodes), and a set E, of edges. Then for a finite n,

$G = \{V, E\}$, where $V = \{V_1, V_2, \ldots, V_n\}$

$E = \{E_{ij} = (V_i, V_j), 1 \le i \le n, 1 \le j \le n\}$

An edge, or arc, defines the connection between two nodes, or vertices, say A and B, and is referred to as the pair (A, B) or undirected edge. If there is a direction, say from initial vertex A to terminal vertex B, the edge is called a directed edge, and is denoted by the ordered pair <A, B>. An arrow is used to denote the direction. A directed edge defines some ordering between the vertices. A directed graph, also referred to as a digraph or network, is a set of vertices and a set of directed edges. An undirected graph does not have any directed edges. Figures 1 and 2 below show an undirected graph and a directed graph, respectively. In Figure 2, <A, B> is a directed edge from vertex A to vertex B.

Undirected connected Directed connected Weighted connected undirected

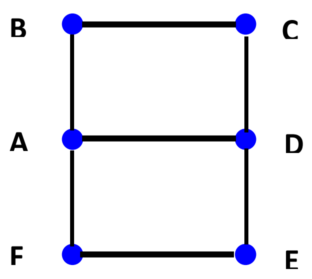

Figure 1

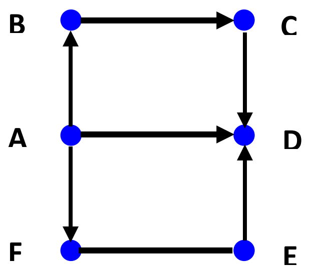

Figure 2

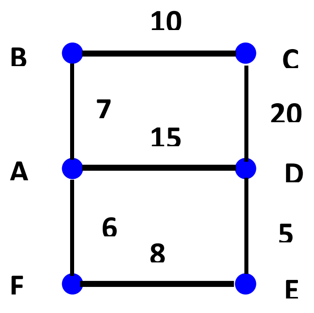

Figure 3

Two vertices, A and B, are adjacent or neighbors if there exists an edge between them. If e denotes an undirected or directed edge between two vertices, A and B (A and B being terminal vertices in the case of a directed graph), the edge e is said to be incident with vertices A and B.

A cycle or loop in a graph is a path in which the source and destination nodes are the same. An acyclic graph is one without any cycles (loops). An edge is a loop if it connects a vertex to itself. For a directed graph, a loop has the same initial and terminal vertices. The degree of a vertex A, deg(A), in an undirected graph is the number of edges incident with the vertex A. A loop for a vertex A adds two to deg(A).

In a directed graph, the in-degree of a vertex A is the number of directed edges terminating at A, and is denoted by deg(A). In a directed graph, the out-degree of a vertex A is the number of directed edges with A as their initial vertex, and is denoted by $\deg^{+}(A)$.

A vertex A is isolated or not connected if deg(A) = 0. A directed or undirected graph is connected if there is an edge between every pair of vertices. That is, a connected graph does not have any isolated vertices. A graph is unconnected if it has at least one isolated vertex.

A directed or undirected graph is said to be weighted if there is a numerical value (e.g., cost, distance) associated with each (directed or undirected) edge. Figure 3 shows a weighted connected undirected graph. The weights for each edge are shown next to the edge. For example, the weight for the edge (A, D) is 15.

A path from a vertex A to a vertex B in a directed (undirected) graph is a sequence of one or more directed (undirected) edges from A to B; it is denoted by p(A, B). A path length m, for vertex A to vertex B, is a sequence of m vertices

$$v0 = A, v_1, v_2, \ldots, v_m = B$$

where the undirected (directed) edges (v_i, v_{i+1}), for i = 0, …, (m-1), are in the graph; the number m represents the hops from A to B.

The shortest path from a vertex A to a vertex B in a weighted directed or undirected graph is the path with a minimum sum of weights of the edges in the path over all paths between A and B.

Graph Traversals

In the list data structure, the traversal always goes from the beginning data element of the list to the end of the list. Because of the special form of the stack and queue data structures, the traversal or visit of any random data element, except one or two data elements, is not allowed. In the binary tree data structure, the traversal uses one of the three most common methods – preorder, postorder, and inorder; these methods dictate the sequence in which the visits of the nodes are performed. A graph is a more general data structure of a tree in the sense that one vertex may have more than one adjacent vertex. This leads us to take extra care to avoid visiting a vertex multiple times. This becomes further complicated with the presence of loops. In order to avoid this problem, the key technique is to tag a vertex after it has been visited.

The graph traversal methods that are devised to avoid this problem, are

1. Depth-first traversal (search) (DFS), and
2. Breadth-first traversal (search) (BFS).

Depth-first Traversals

The process of depth-first traversing a graph, also referred to as backtracking, is very similar to a preorder traversal of a binary tree. That is, the traversal moves away from a current vertex as soon as possible. For an undirected graph, assume that DFS starts its visit at a vertex A, which has the adjacent vertices A_i, for i = 1, …, p. The start vertex A is visited. Then DFS moves to visit any unvisited adjacent vertex, say A_1, of A, and stores the remaining adjacent vertices A_i, for i = 2, …, p, in a stack for a later visit if they remain unvisited. After all adjacent vertices of each adjacent vertices of each adjacent vertex of A_1 are visited, DFS backs up to visit the remaining adjacent vertices A_i, (i = 2, …, p) in the stack that have not been previously visited. The process continues until all vertices in the graph are visited.

Breadth-first Traversals

Breadth-first traversals (BFS) in a graph is very similar to the inorder traversal for binary trees. BFS visits a current vertex, say A, and all its adjacent vertices A_i, (i = 1, …, p) if not visited previously, before visiting any adjacent vertices of A_i, (i = 1, …, p). Here is an algorithm for performing a breadth-first traversal:

Assume BFS starts at vertex, A, which has p number of adjacent vertices. The steps of BFS are:

1. Create and initialize queue object Que_obj.
2. Initialize the queue object to be empty.
3. For the set of n vertices, allocate an array object, visit[ ], of size n.

4. Initialize visit[ ] by setting all its b entries to FALSE.
5. Set visit[A] to TRUE.
6. Visit (e.g. print) the vertex A.
7. Add the start vertex A to queue object, Que_obj.
8. While "Que_obj is not empty", do steps 9 through 15
9. Remove the vertex at the head of Que_obj, and set it to curr_start_vrtx, (i.e., curr_start vrtx = Dequeue(Que_obj)) (Note: In the first pass of the whiel loop, curr_start_vrtx = A)
10. Get all vertices adjacent to curr_start_vrtx.
11. For all vertices, say A_i, (i = 1, …, p), adjacent to the current vertex curr_start_vrtx, do steps 12 through 15.
12. If A_i, has not been visited previously, (i.e., visit[A_i] = FALSE), do steps 13 through 15.
13. Visit[A_i] = TRUE;
14. Visit (e.g. print) A_i,.
15. Add the vertex A_i to to Que_obj to visit its adjacent vertices.
16. Deallocate and remove the queue object Que_obj.
17. Deallocate and remove the array object visit[ ].
18. Return.

Graph Object

A graph object is specified by an Abstract Data Type (ADT) graph.

Definition: An ADT graph is a data structure that contains a set of finite number of vertices and a set of finite number of edges, together with a set of operations that is used for defining, manipulating, and abstracting the vertices and edges.

A set of possible methods on the graph object is:

1. Instantiate an initialize a graph object.
2. Destroy (delete) a graph object.
3. Check if the set of graph vertices is empty.
4. Check if the set of graph edges is empty.
5. Check if two vertices are adjacent.
6. Build a graph object from a given set of vertices and edges.
7. Add a vertex in a graph object.
8. Search for a vertex specified by a single key.
9. Delete a vertex identified by a single search key value.
10. Update information of a given vertex in the graph.
11. Add an edge in a graph object.
12. Search for an edge specified by a key.
13. Delete an edge from a graph object.
14. Traverse a graph using depth-first traversal.
15. Traverse a graph using breadth-first traversal.
16. Print a graph object.
17. Determine attributes of a graph object.
18. Compute the shortest path between two given vertices.
19. Display the shortest path between two given vertices.

A vertex can be as simple as the types int, char, or a structure with complex data types. The vertices and edges are the data elements of an ADT graph. These are internally represented using a linked-list. In either case, the internal representation of vertices and edges may change, but the operations (actions) on them will not change. The same definition applies to both the ADT undirected graph and the ADT directed graph. For the ADT directed graph, the direction of edges must be kept in mind.

A graph object is an instance of a derived implementation spedific class. The base class, form which the graph object is derived, is implementation unspecific; it provides a set of uniform methods (actions or interfaces) for a graph object, which are implemented in the derived class. Let us define an abstract base class Graph from which all graph objects will be derived from:

```
// Define abstract base class for graph objects.

class Graph  {

   public:

        virtual int      build_graph(void) = 0;

        virtual int      is_graph_empty(void) = 0;

        virtual void     find_shortest_path(void) = 0;

        virtual int      get_min(void) = 0;

        virtual void     show_paths(KEY_TYPE src_vrtx, KEY_TYPE dst_vrtx) = 0;

        virtual int      check_graph(void) = 0;

        virtual void     print_graph(void) = 0;

};
```

Implementations of Graph Objects

The data elements for an undirected or directed graph object are the set of vertices and the set of edges, and weights for an undirected and directed weighted graph object. For your project, the vertices are stored in an array, and the adjacent vertices are stored in singly linked data structures. You may assume that the maximum number of vertices is 20. You can represent the KEY_TYPE using either a character or an integer to uniquely defines each vertex. For the Object Oriented Programming characteristics, define a template Wt-digraph class that is derived from the abstract base class Graph. The template class will be used to determine the data type contained at each vertex. You may assume that the weights are of type int.

Write and test an OOP with the following actions:

1. Instantiates a Wt_digraph object to store int data type.
2. Constructor.
3. Copy constructor.
4. Overloaded assignment operator.
5. Destructor.
6. Create an undirected or directed and unweighted or weighted graph object.
7. Check to see if graph is empty.
8. Print the graph.
9. Add a vertex.
10. Delete a vertex
11. Search for a vertex.
12. Add an adjacent vertex.
13. Get the source and destination vertices.
14. Print the object in (4).
15. Perform DFS of the object in (4).
16. Perform BFS of the object in (4).
17. Determine the shortest path from a specified source vertex to a specified destination vertex.
18. Display the shortest path from a specified source vertex to a specified destination vertex.
19. Display the number of hops in the shortest path from a specified source vertex to a specified destination vertex.

Shortest-Path Algorithm (Edsger Dijkstra)

Routing problems: Class of problems that require search algorithms to find an optimal path in a network – a shortest path in a graph or a digraph, a cheapest path in a weighted graph or a digraph, a cheapest path in a weighted graph or digraph, etc… Examples: airline network in which the vertices represent cities and the directed arcs represent flights connecting these cities. An interesting problem is to find the most direct route between two cities, that is, the route with the fewest intermediate stops.

Dijkstra Algorithm

Input: A digraph, a start vertex, and a destination vertex.

Output: The vertices on a shortest path from start to destination or a message indicating that destination is not reachable from start.

```
1. Visit start and label it with 0.
2. Initialize distance to 0.
3. Initialize a queue to contain only start.
4. While destination has not been visited and the queue is not empty, do the following:
     a. Remove a vertex v from the queue.
     b. If the label of v is greater than distance, incremement distance by 1.
     c. For each vertex w adjacent to v:
           If w has not been visited, then
```

```
                i.   Visit w and label it with distance + 1.
               ii.   Add w to the queue
   5.  If destination has not been visited then
         Display “Destination not reachable from start vertex.”

       Else find the vertices p[0], …, p[distance] on the shortest path as follows:

           a.  Initialize p[distance] to destination.
           b.  For each value of k ranging from distance – 1 down to 0:
                 Find a vertex p[k] adjacent to p[k+1] with label k.
```

References:

1. Graph theory and its applications / Jonathan Gross, Jay Yellen (1999).
2. Modern graph theory / Béla Bollobás (1998).
3. Graph theory : flows, matrices / Bela Andrasfai (1991).