INB371 – Data Structure and Algorithms

2014 – Assignment 2

Shortest Distances

Due Date: 11:59 pm – 1 June, 2014

Weighting: 30%

Individual Assignment

Introduction

Navigation devices utilising Global Positioning System (GPS) satellite technology have become commonplace in today's society. Road networks can be modelled and information about the distances between locations are included so that choices can be made by the users of GPS about their choice of route between two locations based on the shortest distance.

Road networks are modelled utilising Graph Theory. Locations are used as the vertices in the graph while the roads between locations are used as the edges of the graph. The edges are given weights based on the distance of the road section between two locations.

The government plans on building a road network between a set of locations and want to investigate the shortest distances and routes from the government's capital to all other locations in the land. Locations have been mapped with Cartesian co-ordinates i.e. x,y co-ordinates, and the roads are to be built as straight lines between the various locations.

Due to various geographic features some road segments between locations cannot be built or the cost of doing so would be prohibitive. The underlying graph that models the road network is, therefore, not complete[1].

For a set of locations and possible road segments which have been randomly generated (see Task 1 below), you are to store the location and segment data in a graph (see Task 2 below). You will then investigate the distance and route of the shortest path from the government's capital to all other locations based on:

1. the graph which includes all of the given the road segments that make up the road network (see Task 3 below)
2. the graph which includes only those segments included in the minimum cost set of road segments which connects all vertices (see Tasks 4 and 5 below)

[1] A complete graph is one in which there is an edge from each vertex to every other vertex.

Task

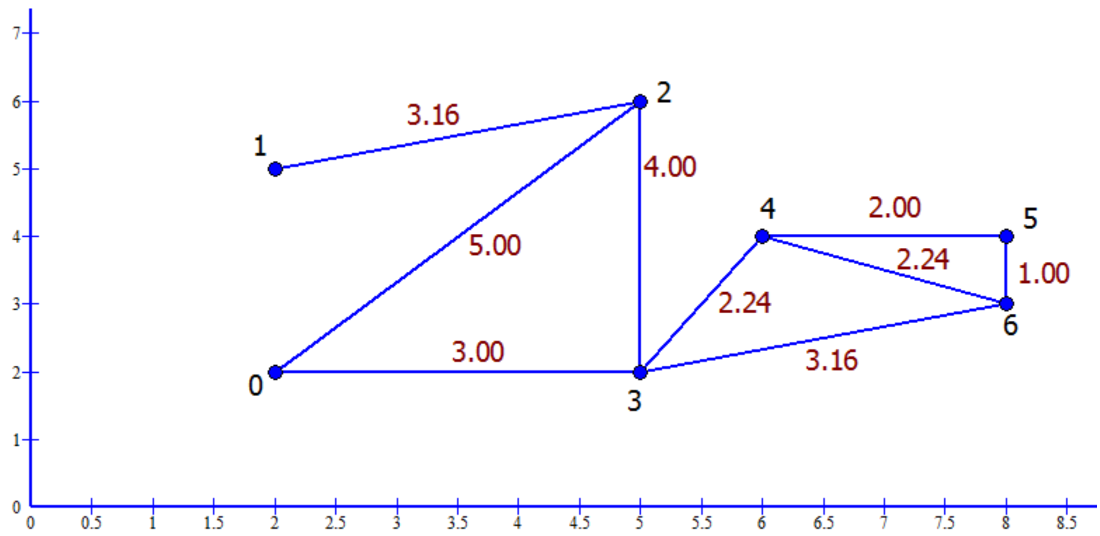
A driver program roads.cpp has been provided. This program does the following:

1. A number of points on a Cartesian plane with x and y co-ordinates between 0 and 100 inclusive will be selected to be the locations for the government's road network. **Point 0** will be designated the location of the government capital.
2. From the points, an undirected graph will be constructed where the points are the vertices of a graph. A random chance value will be used to determine if a road segment between two locations is included in the possible road network. Each road length will be the Euclidean distance (d) between the two vertices for the edge when constructing the graph:

$$d = \sqrt{(x_0 - x_1)^2 + (y_0 - y_1)^2}$$

3. Calculate the shortest distance from the government capital to all other locations and describe the route between the capital and each location as a series of location numbers. In cases where no path exists between the capital and a location, a message should be printed to indicate this. This task can be accomplished with Dijkstra's Single Source Shortest Path algorithm.

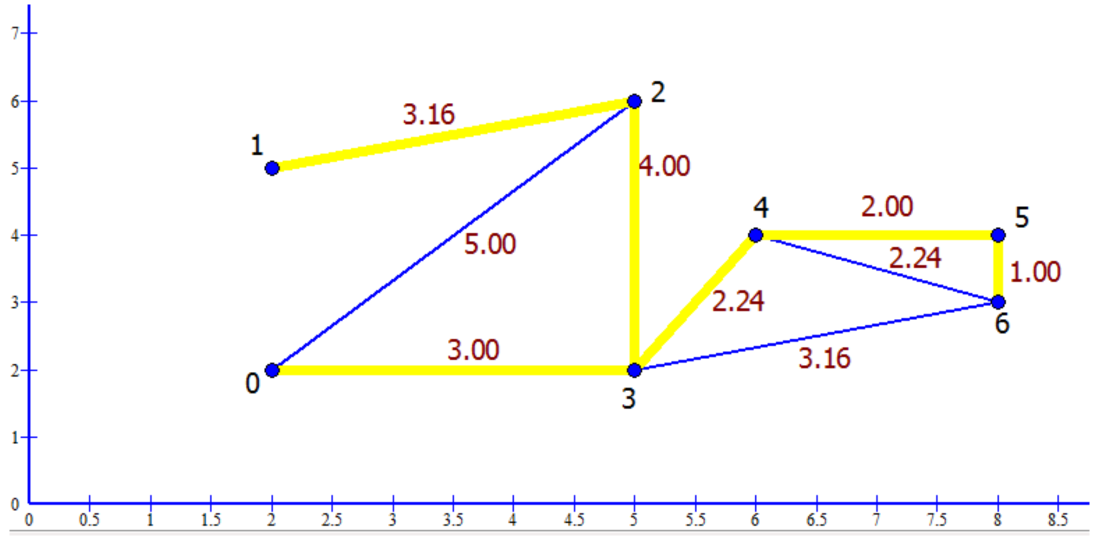
   The diagram below shows a graph with 7 vertices (labelled 0 to 6 in black) with edges in blue and edge weight in red.



   For this graph the shortest distance to each of the other vertices and the path to get there is as follows:

Destination	Distance	Path
1	8.16	0 2 1
2	5.00	0 2
3	3.00	0 3
4	5.24	0 3 4
5	7.16	0 3 6 5
6	6.16	0 3 6

4. Calculate the length of the minimum road network that can be built that connects all of the locations. This task can be accomplished with a Minimum Cost Spanning Tree algorithm.



The yellow edges indicate those that form the minimum cost spanning tree for the given graph. The sum total of these edge weights is approximately15.40. Note that the MST contains no cycles and if any of the other edges had been selected, the total weight would not have been a minimum.

5. Calculate the shortest distance from the government capital to all other locations and describe the route between the capital and each location as a series of location numbers assuming that the road segments that can be built are those in the road network described by the Minimum Cost Spanning Tree. This task can be accomplished with either the Breadth First Search algorithm or the Depth First Search algorithm.

Using only the edges in the MST the shortest distance to each of the other vertices and the path to get there is as follows:

Destination	Distance	Path
1	10.16	0 3 1
2	7.00	0 3 2
3	3.00	0 3
4	5.24	0 3 4
5	7.24	0 3 4 5
6	8.24	0 3 4 5 6

6. The program must also be able to read test files from the command line (as for Assignment 1).

Design Decisions

The following design decisions may aid in your implementation. These guidelines do not have to be followed but it is strongly recommended that you do.

The solution that produces the given output was built by creating the following classes:

- **`Random`** – used to produce random numbers for point locations and whether or not a road segment is to be included in the road network
- **`Point`** – a two-dimensional representation of a Cartesian point with an x co-ordinate and a y co-ordinate
- **`Vertex`** – encapsulates a vertex identifier and other information required by Dijkstra’s algorithm, Kruskal’s algorithm and the Breadth First Search algorithm.
- **`Edge`** – encapsulates information about a weighted undirected edge in a graph
- **`Graph`** – encapsulates graph representations along with functionality to determine shortest paths, and the minimum spanning tree for the graph.
- **`DisjointSet`** – A data structure used to perform Union-Find operations as required in Kruskal’s Minimum Cost Spanning Tree algorithm

The `Random` Class

This class is based on the exercises included in Workshop 3. To ensure that different random values are generated on successive runs, the **`randomise()`** method must be called by the constructor.

Private	
`randomise()`	Initialises the random-number generator so that its results are unpredictable. If this function is not called the other functions will return the same values on each run.
Public	
`Random()`	Constructor - calls the randomise method
`int randomInteger(int, int)`	Generates a random integer number greater than or equal to the first parameter and less than or equal to the second parameter.
`bool randomChance(double)`	Generates a true/false outcome based on the parameter value. For example, calling **`randomChance(0.30)`** returns true 30% of the time.

The `Point` Class

This class encapsulates the *x* and *y* co-ordinates of a point in a Cartesian plane and has functionality to determine the distance between two instances of **`Point`** objects.

Public	
`Point(double, double)`	Constructor that sets the x and y co-ordinates for the **`Point`** object
`~Point()`	Destructor
`double distanceTo(Point*)`	Returns the Euclidean distance between this **`Point`** and the **`Point*`** parameter to the function
`friend ostream& operator<<(ostream&, Point&)`	Produces a string representation of this **`Point.`** e.g.`"2 13"`

The `Vertex` Class

This class encapsulates the **`identifier`** of a **`Vertex`** object and a collection of **`Vertex`** objects in the minimum spanning tree of the graph which are adjacent to this **`Vertex`**.

Private	
`unsigned int identifier`	Instance field to store the identifier for this Vertex
`set<unsigned int> adjacencies`	Stores vertex identifiers for vertices adjacent to this vertex as discovered during the Graph::Minimum Spanning Tree method
`bool discovered`	Keeps track of whether or not this vertex has been visited during Dijkstra's algorithm or discovered during the Breadth First Search method
`unsigned int predecessorId`	Stores the predecessor vertex identifier for path discovery
`double minDistance`	Store the minimum distance from the source to this vertex. Used by Dijkstra's algorithm.
Public	
`Vertex()`	No argument constructor. Can be left with an empty body.
`Vertex(unsigned int)`	Constructor which sets the vertex **`identifier`** (useful for indexing into collections of vertices)
`~Vertex()`	Destructor
`unsigned int getId()`	Accessor for this vertex' **`identifier`**
`void addAdjacency(unsigned int)`	Adds a **`Vertex'`** identifier to this **`Vertex`**' adjacency list. Used by the Graph::Minimum Spanning Tree method
`set<unsigned int>* getAdjacencies()`	Returns a pointer to a collection of **`int`**, being the vertices adjacent to this **`Vertex.`** Used by the Graph::Breadth First Search method
`void setDiscovered(bool)`	Mutator for the **`discovered`** field
`bool isDiscovered()`	Accessor for the **`discovered`** field
`void setPredecessorId(unsigned int)`	Mutator for the **`predecessorId`** field
`unsigned int getPredecessorId()`	Accessor for the **`predecessorId`** field
`void setMinDistance(double)`	Mutator for the **`minDistance`** field
`double getMinDistance()`	Accessor for the **`minDistance`** field

`bool operator()(Vertex*, Vertex*)`	Function operator implementation to provide an ordering for two **`Vertex`** instances. Returns true if the **`minDistance`** of the first parameter is greater than that of the second parameter
`friend ostream& operator<<(ostream&, Vertex&)`	Provides a string representation of this object. Useful for debugging.

The `Edge` Class

This class encapsulates pointers to the source and destination **`Vertex`** objects and the **`weight`** of the **`Edge`**. An alternative would be to encapsulate **`Vertex`** identifiers (**`int`**s) rather than pointers to **`Vertex`**.

Private	
`Vertex* source`	Instance field for the source vertex
`Vertex* destination`	Instance field for the destination vertex
`double weight`	Instance field for the edge weight
Public	
`Edge()`	No argument constructor. Can be left with an empty body.
`Edge(Vertex*, Vertex*, double)`	Constructor which sets the **`source`** vertex, the **`destination`** vertex and the weight for this **`Edge`**
`~Edge()`	Destructor
`Vertex* getSource()`	Returns a pointer to the **`source`** vertex
`Vertex* getDestination()`	Returns a pointer to the **`destination`** vertex
`double getWeight()`	Returns the weight of this **`Edge`**
`bool operator()(Edge*, Edge*)`	Function operator provides an ordering for edges. Returns true if the weight of the first parameter is greater than that of the second paramter.
`friend ostream& operator<<(ostream&, Edge&)`	Returns a string representation of this **`Edge`**. Useful for debugging purposes.

The **Graph** Class

This class encapsulates a weighted undirected graph.

The calculations require fast lookup of edge weights, so they are best stored in an adjacency matrix (two dimensional array of weight values).

A vector of **Vertex*** will be used to store the **Vertex** instances.

The vertices required for calculating the shortest paths using Dijkstra's algorithm need to be in sorted order based on the minimum distance from the source vertex to the vertex instance under consideration. The vertices can be stored in a priority queue. The Vertex class requires the function operator to be overloaded to provide the ordering.

The edges required for calculating Kruskal's Minimum Spanning Tree need to be in sorted order. They can be stored in a priority queue. The Edge class requires the function operator to be overloaded to provide the ordering. Kruskal's algorithm will store adjacent vertex identifiers from the minimum spanning tree in the adjacency list encapsulated in each vertex.

Private	
unsigned int numVertices	Instance field for the number of vertices in the graph.
double weights**	The adjacency matrix for this graph. Two-dimensional array of weights.
priority_queue<Edge*, vector<Edge*>, Edge> edges	Storage for edges to be used by Kruskal's algorithm for calculating the minimum cost spanning tree.
vector<Vertex*> vertices	Storage for graph vertices
Public	
Graph(unsigned int)	Constructor sets the number of vertices in this **Graph.** Initialises the two dimensional array of weights by setting all values to INFINITY (some value larger than any possible edge weight) except the diagonal of the array where the weight is set to 0.
~Graph()	Destructor
void addVertex(Vertex*)	Adds pointer to **Vertex** to the collection of vertices for this **Graph**.
Vertex* getVertex(int)	Accessor returns a pointer to the **Vertex** with the identifier/index in the parameter
void addEdge(Edge*)	Adds pointer to **Edge** to the edge list for this **Graph**. Using the source and destination identifiers from the edge, sets the weight of the **undirected edge** in the adjacency matrix.
double minimumSpanningTreeCost()	Uses Kruskal's algorithm to find the Minimum Spanning Tree (MST) for this Graph. Stores the edges of the MST in the

	adjacency list of each Vertex. Returns the cost of the minimum spanning tree.
void dijkstra(unsigned int)	Determines the shortest path from the source vertex to all other vertices. Prints the length of the path and the vertex identifiers in the path.
void bfs(unsigned int)	Determines the shortest path from the source vertex to all other vertices using only the adjacencies in the minimum spanning tree. Prints the length of the path and the vertex identifiers in the path.
friend ostream& operator<<(ostream&, Graph&)	Outputs the adjacency matrix for the graph. If an edge weight is INFINITY, - should be printed instead of a number.

The `DisjointSet` Class

This data structure is used for very efficient look-up (Find) to determine which set an element is in. It also provides a very efficient way to join (Union) two sets together.

It is based on arrays. For implementation details, see Lecture 11.

DisjointSet(int)	Constructor which sets the **size** of this **DisjointSet**
~DisjointSet()	Destructor
int find(int)	Returns the **index** of the parent set of the element in the parameter
void join(int, int)	Creates the union of two disjoint sets whose **index**es are passed as parameters
bool sameComponent(int, int)	Returns **true** if the two **index**es passed as parameters are in the same set

Dijkstra's Single Source Shortest Path Algorithm

This algorithm is discussed in Lecture 11.

A source vertex is specified and the shortest path from the source vertex to all other vertices is determined.

In the initialisation steps, each vertex is marked as not visited, its predecessor for the path is set as the source vertex and the minimum distance to reach the vertex is set as the weight of the edge from the source vertex to the vertex. If no edge exists between the source vertex and the vertex, the minimum distance is set to INFINITY. Each vertex is pushed onto a priority queue.

As each unvisited vertex, **`u`**, is removed from the priority queue it is marked visited. The minimum distance to reach this vertex has already been calculated. Each of the vertices adjacent to **`u`** are then tested to see if the minimum distance to reach **`u`** plus the distance from **`u`** to the adjacent vertex is less than the minimum distance currently recorded for the adjacent vertex. If the calculated distance is less, the minimum distance for the adjacent vertex is updated. The predecessor vertex for the path is also updated to **`u`**.

Algorithm

```
dijkstra(sourceId)
   create a priority queue of Vertex*
   for all vertices
      set visited to false
      set the predecessorId to sourceId
      set minimum distance to distance from source to this
               vertex from the adjacency matrix
      push vertex onto priority queue
   end for

   while priority queue is not empty
      poll the priority queue (let this vertex be called u)
      mark u as visited
      for each unvisited vertex adjacent to u
         if u.minDistance + weight[u][v] < v.minDistance
            set v.minDistance to the new value
            set v.predecessorId to u.id
            push v onto priority queue
         end if
      end for
   end while
```

```
   // output the length of the shortest paths and the paths
   for each vertex other than the source
      retrieve the minimum distance
      if distance == INFINITY
         output NO PATH message
      else
         output distance
         output path (see Lecture 11)
      end if
   end for
end
```

Kruskal's Algorithm

Kruskal's algorithm selects the shortest edge from an edge list and incorporates that edge into the minimum spanning tree if and only if the edge does not form a cycle. The Disjoint Set data structure is used to keep track of whether or not cycles are formed.

As edges are added to the minimum spanning tree, the cost of the minimum spanning tree is calculated by totalling the weights of the edges that are included in the minimum spanning tree.

As the edges are added, the adjacency list held by each vertex is now updated. The vertex identifier at the destination end of the undirected edge is added to the source vertex' adjacency list **and vice versa**.

For details of this algorithm see Lecture 11.

Breadth First Search Algorithm

The shortest paths of the road segments included in the minimum spanning tree can be calculated using a breadth first search traversal of the minimum spanning tree starting at the source vertex.

In the initialisation step, all vertices in the graph are marked as not discovered.

An empty queue is created, the source vertex is marked discovered and is then pushed onto the queue.

While the queue is not empty, the vertex at the front of the queue is removed and is known as the current vertex. Its adjacencies (in the adjacency list populated by the minimum spanning tree method) are inspected. If the adjacent vertex has not been discovered, the adjacent vertex's predecessor field is set to the current veretx' id. The adjacent vertex is marked as discovered and is pushed onto the queue.

Algorithm

```
bfs(sourceId)
   for each vertex in the graph
      set vertex discovered field to false
   end for

   create an empty queue
   set source vertex discovered to true
   push source vertex onto queue

   while queue is not empty
      set current to front of queue (i.e. remove front)
      for each vertex adjacent to current
         if not adjacent.discovered
            set adjacent.predecessor to current id
            set adjacent.discovered to true
            push adjacent onto queue
         end if
      end for
   end while

   // output the length of the shortest paths and the paths
   for each vertex other than the source
      walk the path from vertex back to source calculating the
              distance of the path
      if distance == INFINITY
         output NO PATH message
      else
         output distance
         output path (see Lecture 11)
      end if
   end for
end
```

Example Output

```
I:\TSP\bin\Debug\roads.exe

City  0 co-ordinates : 50 93
City  1 co-ordinates : 46 77
City  2 co-ordinates : 47 88
City  3 co-ordinates : 46 68
City  4 co-ordinates : 62  4
City  5 co-ordinates : 41 57
City  6 co-ordinates : 33 88
City  7 co-ordinates : 65 98
City  8 co-ordinates : 92 29
City  9 co-ordinates : 26 49

Edge Weights
============
   0.00      -      -  25.32      -      -      -      -      -      -
      -   0.00      -      -  74.73  20.62      -  28.32  66.48  34.41
      -      -   0.00  20.02  85.33      -      -      -  74.20      -
  25.32      -  20.02   0.00  65.97      -      -      -      -      -
      -  74.73  85.33  65.97   0.00      -  88.87      -      -  57.63
      -  20.62      -      -      -   0.00      -      -      -      -
      -      -      -      -  88.87      -   0.00  33.53  83.44      -
      -  28.32      -      -      -      -  33.53   0.00      -      -
      -  66.48  74.20      -      -      -  83.44      -   0.00  68.96
      -  34.41      -      -  57.63      -      -      -  68.96   0.00


Shortest Paths
==============
Distance from 0 to  1 = 166.02 travelling via  0  3  4  1
Distance from 0 to  2 =  45.34 travelling via  0  3  2
Distance from 0 to  3 =  25.32 travelling via  0  3
Distance from 0 to  4 =  91.29 travelling via  0  3  4
Distance from 0 to  5 = 186.64 travelling via  0  3  4  1  5
Distance from 0 to  6 = 180.15 travelling via  0  3  4  6
Distance from 0 to  7 = 194.34 travelling via  0  3  4  1  7
Distance from 0 to  8 = 119.55 travelling via  0  3  2  8
Distance from 0 to  9 = 148.92 travelling via  0  3  4  9

MST Weight = 352.29
===================

Shortest Paths on MST
=====================
Distance from 0 to  1 = 183.33 travelling via  0  3  4  9  1
Distance from 0 to  2 =  45.34 travelling via  0  3  2
Distance from 0 to  3 =  25.32 travelling via  0  3
Distance from 0 to  4 =  91.29 travelling via  0  3  4
Distance from 0 to  5 = 203.94 travelling via  0  3  4  9  1  5
Distance from 0 to  6 = 245.17 travelling via  0  3  4  9  1  7  6
Distance from 0 to  7 = 211.64 travelling via  0  3  4  9  1  7
Distance from 0 to  8 = 249.81 travelling via  0  3  4  9  1  8
Distance from 0 to  9 = 148.92 travelling via  0  3  4  9


Process returned 0 (0x0)   execution time : 0.048 s
Press any key to continue.
```

Academic Integrity

You must submit only your own work for the assignment. Your attention is drawn to QUT's rules on academic integrity and plagiarism (*Manual of Policies and Procedures*, Section C/5.3 *Academic Integrity* and Section E/2.1 *Student Code of Conduct*).