Scott Christensen

02/19/2016

C.S. 312

Project #3 Report

Code:

```
private void solveButton_Clicked()
        {
            // *** Implement this method, use the variables "startNodeIndex" and
"stopNodeIndex" as the indices for your start and stop points, respectively ***

            //Here we define the data objects we'll be using
            String s = sourceNodeBox.Text;
            int startIndex = Int32.Parse(s);
            s = targetNodeBox.Text;
            int endIndex = Int32.Parse(s);
            EuclideanDistanceImplementation edi = new EuclideanDistanceImplementation();
            Font arialFont = new Font("Arial", 11);
            Pen p = new Pen(Color.Black, 1);
            double totalDistance = 0;
            DateTime startTime;
            double arrayTime = 0;
            double heapTime = 0;
            double elapsedMillisecondsTime = 0;

            if (arrayCheckBox.Checked == true)
            {
                //start the timer
                startTime = DateTime.Now;

                //compute dijkstra's algorithm with the data we have.
                ArrayImplementation ai = new ArrayImplementation(startIndex, endIndex,
points, adjacencyList);
                List<double> previous = ai.dijkstras();//returns previous with all the
indexes till the endIndex.

                int currIndex = endIndex;

                //Draw the shortest path with the costs backwards
                //Time: O(|v|) Because you may have to draw each point at worst.
                //Space: O(1) Beacuse you only change one point.
                while (currIndex != startIndex)
                {
                    //Console.WriteLine("Here");
                    //Console.WriteLine("Prev " + previous.ElementAt(currIndex));
                    //make sure that we can actually reach this thing
                    if (previous.ElementAt(currIndex) == 2147483647)
                    {
                        this.pathCostBox.Text = "unreachable";
                        return;
                    }
```

```
                    //draw the line
                    graphics.DrawLine(p, points[currIndex],
points[(int)previous.ElementAt(currIndex)]);

                    //get and display length for every line drawn
                    double dist = edi.calculate(points[currIndex],
points[(int)previous.ElementAt(currIndex)]);
                    totalDistance = totalDistance + dist;
                    string str = Convert.ToString((int)dist);

                    //calculation for the slope of the line and put it in the middle
                    RectangleF rectf = new RectangleF(70, 90, 80, 80);//x, y, width,
height
                    rectf.X = Math.Abs(points[(int)previous.ElementAt(currIndex)].X +
points[currIndex].X) / 2;
                    rectf.Y = Math.Abs(points[(int)previous.ElementAt(currIndex)].Y +
points[currIndex].Y) / 2;
                    graphics.DrawString(str, arialFont, Brushes.Black, rectf);
                    currIndex = (int)previous.ElementAt(currIndex);

                    //Stop the timer and write the values to the forms
                    this.pathCostBox.Text = Convert.ToString(totalDistance);
                    elapsedMillisecondsTime = (DateTime.Now -
startTime).TotalMilliseconds;
                    arrayTime = elapsedMillisecondsTime / 10000;
                    this.arrayTimeBox.Text = Convert.ToString(arrayTime);
                }
            }


            //Now do the same thing with a Heap
            //O(log |v|))
            //start the timer
            startTime = DateTime.Now;

            HeapImplementation heapQueue = new HeapImplementation(points.Count);
            //ArrayQueue
            List<double> dist1 = new List<double>();
            List<int> prev1 = new List<int>();
            List<double> dist2 = new List<double>();
            List<int> prev2 = new List<int>();
            totalDistance = 0;

            for (int u = 0; u < points.Count; u++)
            {
                dist1.Add(Int16.MaxValue);
                prev1.Add(-1);
                dist2.Add(Int16.MaxValue);
                prev2.Add(-1);
            }
            dist1[startNodeIndex] = 0;
            dist2[startNodeIndex] = 0;

            //Heap implementation
            for (int u = 0; u < dist2.Count; u++)
            {
                Node node = new Node(u);
```

```
                heapQueue.Add(node, dist2[u]);
            }

            while (heapQueue.count > 0)
            {
                Node min = heapQueue.ExtractMin();
                foreach (int adj in adjacencyList[min.pointsIndex])
                {
                    double newDistance = dist2[min.pointsIndex] + edi.getMagnitude(min,
adj, points);
                    if (dist2[adj] > newDistance)
                    {
                        dist2[adj] = newDistance;
                        prev2[adj] = min.pointsIndex;
                        heapQueue.DecreasePriority(adj, newDistance);
                    }
                }
            }

            //Draw the shortest path with the costs backwards
            //Time: O(|v|) Because you may have to draw each point at worst.
            //Space: O(1) Beacuse you only change one point.
            int currIndexToDraw = stopNodeIndex;
            while (currIndexToDraw != startNodeIndex)
            {
                int prevIndex = prev2[currIndexToDraw];

                //Console.WriteLine("Here");
                //Console.WriteLine("Prev " + points[currIndexToDraw]);
                //make sure that we can actually reach this thing
                if(prevIndex > points.Count-1 || prevIndex < 0)
                {
                    this.pathCostBox.Text = "unreachable";
                    return;
                }

                graphics.DrawLine(p, points[currIndexToDraw], points[prevIndex]);

                double currDistance = edi.calculate(points[prevIndex],
points[currIndexToDraw]);
                totalDistance += currDistance;

                String text = Convert.ToString((int)currDistance);
                RectangleF drawingBox = new RectangleF(Math.Abs(points[prevIndex].X +
points[currIndexToDraw].X) / 2,
                    Math.Abs(points[prevIndex].Y + points[currIndexToDraw].Y) / 2, 80,
80);
                graphics.DrawString(text, new Font("Arial", 11), Brushes.Black,
drawingBox);

                currIndexToDraw = prev2[currIndexToDraw];
            }
            this.pathCostBox.Text = Convert.ToString(totalDistance);

            //Now set the time it takes for all this to happen
            elapsedMillisecondsTime = (DateTime.Now - startTime).TotalMilliseconds;
            heapTime = elapsedMillisecondsTime / 10000;
            this.heapTimeBox.Text = Convert.ToString(heapTime);
```

```
            if (arrayCheckBox.Checked == true)
            {
                this.differenceBox.Text = Convert.ToString(arrayTime / heapTime);
            }
        }

namespace NetworkRouting
{
    class ArrayImplementation
    {
        int startIndex;
        int endIndex;
        List<PointF> points;
        List<HashSet<int>> adjacencyList;

        List<double> distance = new List<double>();
        List<double> previous = new List<double>();
        List<double> queue = new List<double>();

        public ArrayImplementation(int startIndexIn, int endIndexIn, List<PointF>
pointsIn, List<HashSet<int>> adjacencyListIn)
        {
            startIndex = startIndexIn;
            endIndex = endIndexIn;
            points = pointsIn;
            adjacencyList = adjacencyListIn;
        }

        //Time: O(|v|^2)
        //Space: O(V)
        //The main function for doing dijkstra's algorithm and resurning the results.
        public List<double> dijkstras()
        {
            //Console.WriteLine("Inside array dijkstras");
            int max = Int32.MaxValue;

            //We start by adding all values to infinity
            for (int i = 0; i < points.Count; i++)
            {
                //Insert
                //Time: O(1) Beacuse your doing it for one point.
                //Space: O(1) Beacuse your making the queue.
                distance.Add(max);
                previous.Add(max);
                queue.Add(max);
            }
            distance[startIndex] = 0;
            queue[startIndex] = 0;
            int minIndex = startIndex;
            //to compare and change indexes

            int minusOnes = 0;
            EuclideanDistanceImplementation edi = new EuclideanDistanceImplementation();
            while (minusOnes < queue.Count)
            {
                double minValue = max;
                //Console.WriteLine("minusOnes " + minusOnes);
```

```
                //the deleteMin function
                //Time: O(|v|) Because you have to iterate over every point to find the
minimum.
                //Space: O(1) Nothing changed.
                for (int i = 0; i < queue.Count; i++)
                {
                    //Console.WriteLine("Dist is " + queue[i]);
                    if (queue[i] != -1 && queue[i] < minValue)
                    {
                        minIndex = i;
                        minValue = queue[i];
                        //Console.WriteLine("min index is " + i + " of " + queue[i]);
                    }
                }
                //Console.WriteLine("The current min Index is " + minIndex);
                //previous.Add(points[minIndex]);
                queue[minIndex] = -1;//set the value to -1 to take it out, basically

                //update the distances of each of this node's directed edges. O(3)
                for (int i = 0; i < adjacencyList[minIndex].Count; i++)
                {
                    int currAdj = adjacencyList[minIndex].ElementAt(i);
                    //Console.WriteLine("currAdj is " + currAdj + " of " +
adjacencyList[minIndex]);
                    double dist = edi.calculate(points[minIndex], points[currAdj]);
                    //Console.WriteLine("dist path of " + currAdj +"  is " +
distance[currAdj] + " versus " + (distance[minIndex] + dist));
                    if (distance[currAdj] > distance[minIndex] + dist)
                    {
                        distance[currAdj] = distance[minIndex] + dist;
                        //Console.WriteLine("Add " + currAdj + " from point " +
minIndex);
                        //previous.Add(minIndex);
                        previous[currAdj] = minIndex;

                        //decreaseKey
                        //Time: O(1) Because you can directly access the index.
                        //Space: O(1) Because you change only one thing.
                        queue[currAdj] = distance[currAdj];
                    }
                }
                minusOnes++;
            }

            previous.Add(endIndex);
            return previous;
        }
    }
}

namespace NetworkRouting
{
    public class Node
        {
            public int pointsIndex;
            public int position;
            public double distance;
            public Node(int ptIndex)
```

```
            {
                pointsIndex = ptIndex;
                position = -1;
                distance = 0;
            }
        }

    public class HeapImplementation
    {

        public List<Node> nodes;
        readonly Node[] data;
        readonly double[] distances;
        public int count;


        //Creates a new, empty priority queue with the specified capacity.
        public HeapImplementation(int capacity)
        {
            data = new Node[capacity];
            distances = new double[capacity];
            nodes = new List<Node>();
            count = 0;
        }

        //Insert
        //Time: O(log v) Because your using a Heap that must reorder itself with every
add.
        //Space: O(1) Because your adding one thing.

        //Adds an item to the queue.  Is position is determined by its priority relative
to the other items in the queue.
        public void Add(Node item, double priority)
        {
            if (count == data.Length)
                throw new Exception("Heap capacity exceeded");

            // Add the item to the heap in the end position of the array (i.e. as a leaf
of the tree)
            int position = count++;
            data[position] = item;
            item.position = position;
            distances[position] = priority;
            // Move it upward into position, if necessary
            MoveUp(position);
            nodes.Add(item);
        }

        //DeleteMin
        //Time: O(log v) Because we're using a Heap and it must rebalance itself after
every deletion.
        //Space: O(1) Because we only change one thing.

        //Extracts the item in the queue with the minimal priority value.
        public Node ExtractMin()
        {
            nodes[data[0].pointsIndex].distance = distances[data[0].position];
            //data[0].distance = distances[data[0].position];
```

```
            Node minNode = data[0];
            Swap(0, count - 1);
            count--;
            MoveDown(0);
            return minNode;
        }

        //DecreaseKey
        //Time: O(log v) Because if you change the weight of one of the Nodes, the Hea
needs to rebalance itself accordingly.
        //Space: O(log v) Because you are changing one thing.
        //Reduces the priority of a node already in the queue.
        public void DecreasePriority(int index, double priority)
        {
            int position = nodes[index].position;
            while ((position > 0) && (distances[Parent(position)] > priority))
            {
                int original_parent_pos = Parent(position);
                Swap(original_parent_pos, position);
                position = original_parent_pos;
            }
            distances[position] = priority;
        }

        //Moves the node at the specified position upward, it it violates the Heap
Property.
        //This is the while loop from the HeapInsert procedure in the slides.
        void MoveUp(int position)
        {
            while ((position > 0) && (distances[Parent(position)] > distances[position]))
            {
                int original_parent_pos = Parent(position);
                Swap(position, original_parent_pos);
                position = original_parent_pos;
            }
        }


        //Moves the node at the specified position down, if it violates the Heap Property
        void MoveDown(int position)
        {
            int lchild = LeftChild(position);//look at it's left child and get its value
            int rchild = RightChild(position); ;//look at it's left child and get its
value
            int largest = 0;
            if ((lchild < count) && (distances[lchild] < distances[position]))
            {
                largest = lchild;
            }
            else
            {
                largest = position;
            }
            if ((rchild < count) && (distances[rchild] < distances[largest]))
            {
                largest = rchild;
            }
            if (largest != position)
```

```
            {
                Swap(position, largest);
                MoveDown(largest);
            }
        }


        //Get the number of items waiting in queue
        public int Count
        {
            get
            {
                return count;
            }
        }

        //Swaps the nodes at the respective positions in the heap
        //Updates the nodes' position properties accordingly.
        void Swap(int position1, int position2)
        {
            Node temp = data[position1];
            data[position1] = data[position2];
            data[position2] = temp;
            data[position1].position = position1;
            data[position2].position = position2;

            double temp2 = distances[position1];
            distances[position1] = distances[position2];
            distances[position2] = temp2;
        }


        //Gives the position of a node's parent, the node's position in the queue.
        static int Parent(int position)
        {
            return (position - 1) / 2;
        }


        //Returns the position of a node's left child, given the node's position.
        static int LeftChild(int position)
        {
            return 2 * position + 1;
        }

        //Returns the position of a node's right child, given the node's position.
        static int RightChild(int position)
        {
            return 2 * position + 2;
        }
    }
}

    class EuclideanDistanceImplementation
    {
        //gets the euclidean distance between two points
        //Time: O(1)
        //Space: O(1)
```

```
        public double calculate(PointF one, PointF two)
        {
            //square root[(y2 - y1)^2 + (x2 - x1)^2]
            double first = Math.Pow(two.Y - one.Y, 2);
            double second = Math.Pow(two.X - one.X, 2);
            return (double)Math.Sqrt(first + second);
        }

        //gets the euclidean distance from a heap Node and the set of points
        //Time: O(1)
        //Space: O(1)
        public double getMagnitude(Node u, int adj, List<PointF> points)
        {
            //square root of [(y2-y1)^2 + (x2-x1)^2]
            double firstTerm = (points[adj].Y - points[u.pointsIndex].Y) * (points[adj].Y
- points[u.pointsIndex].Y);
            double secondTerm = (points[adj].X - points[u.pointsIndex].X) *
(points[adj].X - points[u.pointsIndex].X);
            return Math.Sqrt(firstTerm + secondTerm);
        }
    }
}
```

Time and Space Complexity Analysis

Array Implementation:

- Insert operation
    - Time complexity: O(1). This is because you are accessing the array once and inserting a distance value there. Reference to code: A simple array(Index) = value.
    - Space complexity: O(1). This is because you are simply adding the value by index directly.
- Delete Min operation
    - Time Complexity: O(|v|), where v is the number of points. This is because you have to iterate through the whole array to find the minimum value. Reference to code: It's a big for loop that checks every index in the array once.
    - Space Complexity: O(1) Because you don't need to store anything other than that value found to perform this operation.
- Decrease Key operation
    - Time Complexity: O(1). This is because you have direct access to the key via an index value. Reference to code: array[index] = new value;
    - Space Complexity: O(1). Because you don't need to store anything to perform this operation.

Heap Implementation:

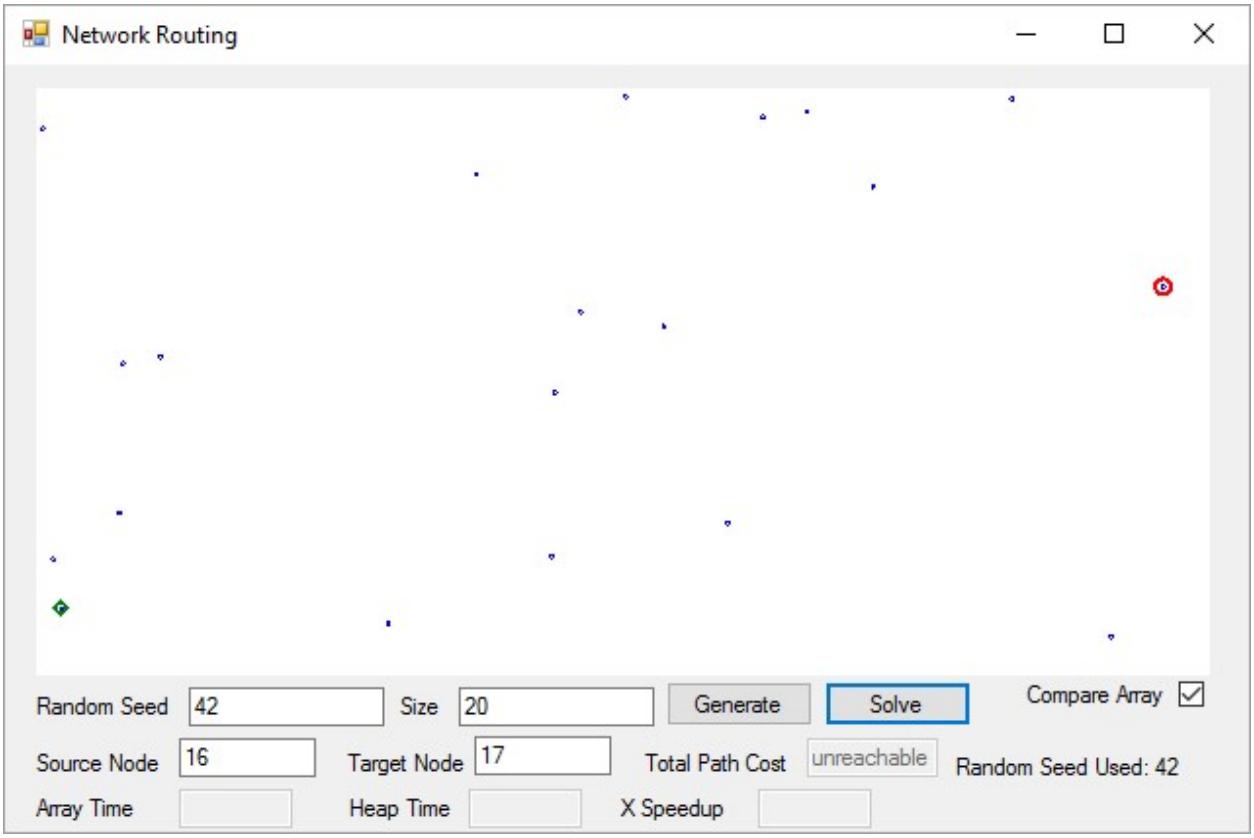
- Insert operation
    - Time complexity: O(log|v|). This is because with each node inserted, the heap must rebalance itself. Reference to code: In the Add operation in the Binary Heap

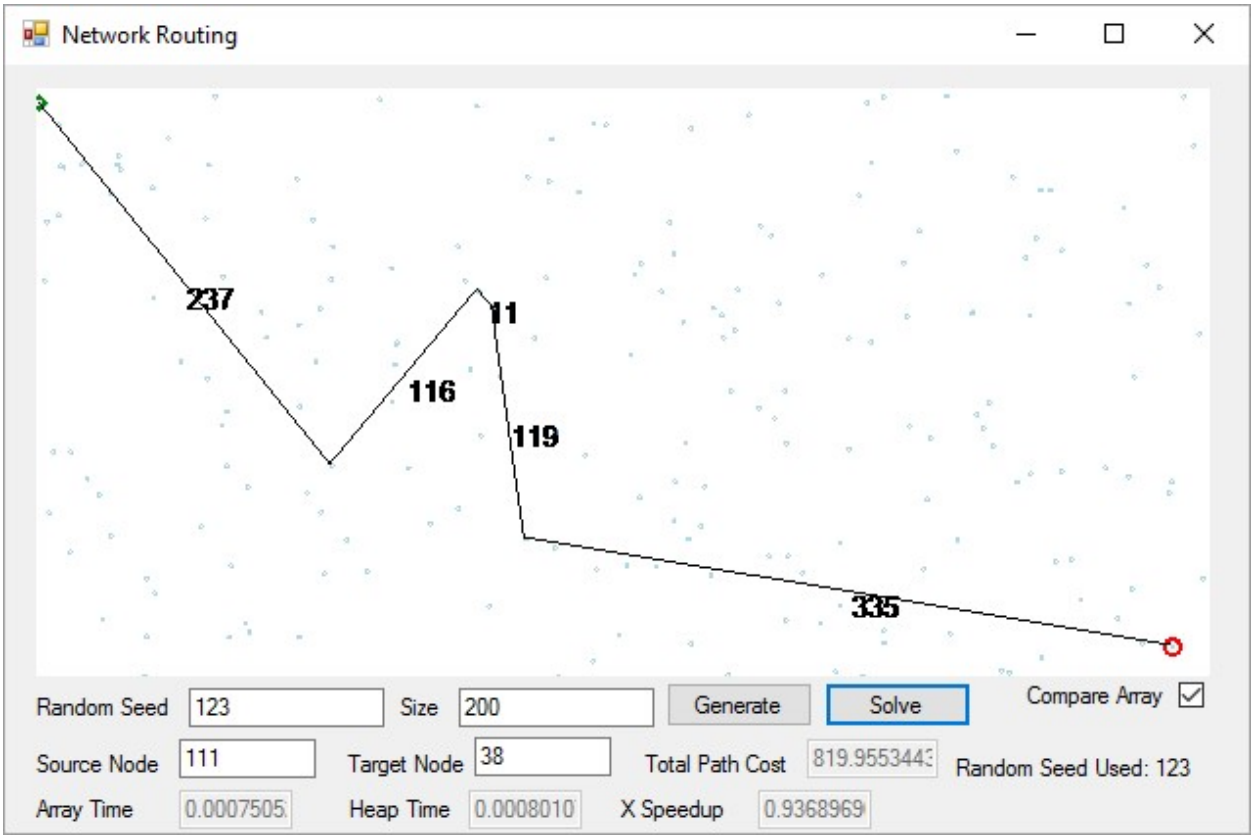
class, you would add the element at the end of the heap and move it up to its proper place. Since this is a Binary Data Structure, this operation is done in O(log2(v) time.
    - Space complexity: O(1). This is because you are simply adding the value by index directly.
- Delete Min operation
    - Time Complexity: O(log|v|).) This is because to delete a node, the heap must rebalance itself. Reference to code: In the Delete operation in the Binary Heap class, you delete a node by swapping the root node with the farthest root node, delete the original root value, then proceed to rebalance the heap accordingly. Since this is a Binary Data Structure, this operation is done in O(log2(v) time.
    - Space Complexity: O(1) Because you don't need to store anything other than that value found to perform this operation.
- Decrease Key operation
    - Time Complexity: O(log(v)). This is because when you want to access any node it the heap, you must recuse down the tree. Reference to code: In the Decease Priority function, you recursively go down to the desired value, change the value, then swap children to parents till the tree is balanced.
    - Space Complexity: O(1). Because you don't need to store anything to perform this operation.

Conclusion:

- Array Implementation
    - Time Complexity: O(|v|^2) Because there are V items in the array queue and you must keep deleting the min until the array is empty. So you call a O(|v|) operation V times. The lower order operations, Insert and Decrease Key, are ignored.
    - Space Complexity: O(|v|+|e|). Because if you sum up all of the operations, you notice that you must store something in the array for each point and edge, as you call the Insert V times.
- Heap Implementation
    - Time Complexity: O((|v| + |e|) log|v|), where v is the number of point & e is the number of edges, because there are |v| items in the heap queue and you have to keep deleting the min until the array is empty. We are also checking each edge for each node and decreasing the key for (another log|v| operation, hence the '+ |e|'). We are calling the delete min operation, which is order log|v|, |v| times, plus calling the decrease key operation |e| times, hence (|v| + |e|) log|v|. The lower order operations are also ignored when talking about Big O.
    - Space Complexity: O(|v| +|e|). Because if you sum up all of the operations, you notice that you must store something in the array for each points and edge, as you call the Insert V times. The fact that each insert takes O(log(|v|)) is ignored.

Screen Shots:

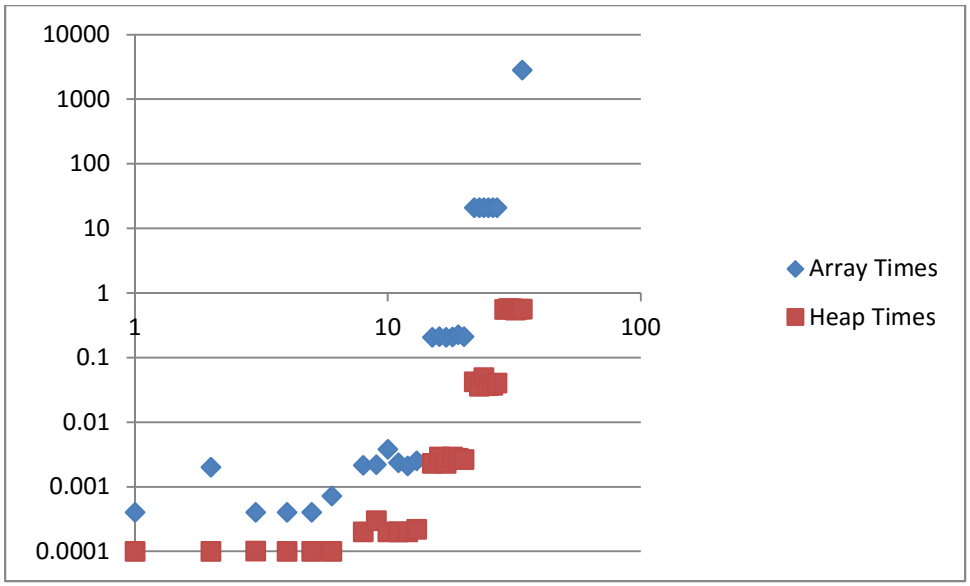



Network Routing
237
116
11
119
335
Random Seed
123
Size
200
Generate
Solve
Compare Array
Source Node
111
Target Node
38
Total Path Cost
819.9553443
Random Seed Used: 123
Array Time
0.0007505
Heap Time
0.0008010
X Speedup
0.9368969

Data:

N	Array Time Averages	Heap Time Averages	Speed up Factor
100	0.0004003	0.0001001	3.9972041
100	0.002001	0.0001	2.0005997
100	0.0004004	0.0001009	3.9681894
100	0.000401	0.0001	4.0086965
100	0.0004011	0.0001	4.009691
ave	0.00072076	0.0001002	3.59687614
1000	0.0021518	0.0002004	10.737974
1000	0.0022014	0.0003005	7.3246381
1000	0.0038023	0.0002	19.004248
1000	0.0023515	0.0002	11.753198
1000	0.0021013	0.0002001	10.500074
ave	0.00252166	0.0002202	11.86402642
10000	0.20569	0.002315	89.3687
10000	0.2097	0.0029019	72.2515
10000	0.2063	0.0023015	89.62146
10000	0.20869	0.0029019	71.915211
10000	0.226706	0.0027518	80.880353
ave	0.2114172	0.00263442	80.8074448
100000	20.783527	0.0422781	491.59016
100000	20.912188	0.0360738	579.700556
100000	20.835353	0.0491676	541.356337
100000	20.792346	0.0370043	519.555555
100000	20.913533	0.0374292	561.924135
ave	20.8473894	0.0403906	538.8253486
1000000		0.5588304	
1000000		0.5778765	
1000000		0.5676864	
1000000		0.5428139	
1000000		0.5603291	
ave	2807.5363	0.56150726	5000

Graph:



Conclusion:

Upon plotting the data produced by both implementations, the data heavily correlates with our expected outcome. Looking at the Array Times, the amount of time it takes grows drastically after 1000 points and continues to grow in a similar fashion. This growth most closely resembles O(n^2) growth, which is what we predicted it to be. In our design experience and from analysis of our finished algorithm to be, we found this to be O(|v|^2), where v are the number of points. True, there are a couple outlying data that resulted from running the program many times and some overhead produced with drawing and recording distances, but overall, the Array Implementation truly does follow a O(n^2) time complexity algorithm.

Looking at the Heap Times however, the amount of time it takes does increase, but not at a rate as show by the Array Implementation. In fact, it seems to maintain a consistent growth rate. This growth most closely resembles O(n log(n)) growth, which is what we also predicted it to be in our design experience and from analysis. This growth, O((|v| + |e|)log(|v|) is what we came up with, since the algorithm preforms it's insert, min value, and decrease key algorithm on each mode v and checks each node e in the process. Again, there are a couple outlying data points that resulted from running the program many times and some overhead produced with drawing and recording distances. But these outliers are a good deal less extreme than those seen resulting from the Array Implementation. Thus, the Heap Implementation truly does follow a O(n log(n)) time complexity algorithm.