Network Routing Report

Trevor Rydalch

10/25/17

1. Code:

Form.cs:

```
public partial class Form1 : Form
{
    public Form1()
    {
        InitializeComponent();
    }

    private void clearAll()
    {
        startNodeIndex = -1;
        stopNodeIndex = -1;
        sourceNodeBox.Clear();
        sourceNodeBox.Refresh();
        targetNodeBox.Clear();
        targetNodeBox.Refresh();
        arrayTimeBox.Clear();
        arrayTimeBox.Refresh();
        heapTimeBox.Clear();
        heapTimeBox.Refresh();
        differenceBox.Clear();
        differenceBox.Refresh();
        pathCostBox.Clear();
        pathCostBox.Refresh();
        arrayCheckBox.Checked = false;
        arrayCheckBox.Refresh();
        return;
    }

    private void clearSome()
    {
        arrayTimeBox.Clear();
        arrayTimeBox.Refresh();
        heapTimeBox.Clear();
        heapTimeBox.Refresh();
        differenceBox.Clear();
```

```
            differenceBox.Refresh();
            pathCostBox.Clear();
            pathCostBox.Refresh();
            return;
        }

        private void generateButton_Click(object sender, EventArgs e)
        {
            int randomSeed = int.Parse(randomSeedBox.Text);
            int size = int.Parse(sizeBox.Text);

            Random rand = new Random(randomSeed);
            seedUsedLabel.Text = "Random Seed Used: " + randomSeed.ToString();

            clearAll();
            this.adjacencyList = generateAdjacencyList(size, rand);
            List<PointF> points = generatePoints(size, rand);
            resetImageToPoints(points);
            this.points = points;
        }

        // Generates the distance matrix.  Values of -1 indicate a missing edge.  Loopbacks are at a cost of 0.
        private const int MIN_WEIGHT = 1;
        private const int MAX_WEIGHT = 100;
        private const double PROBABILITY_OF_DELETION = 0.35;

        private const int NUMBER_OF_ADJACENT_POINTS = 3;

        private List<HashSet<int>> generateAdjacencyList(int size, Random rand)
        {
            List<HashSet<int>> adjacencyList = new List<HashSet<int>>();

            for (int i = 0; i < size; i++)
            {
                HashSet<int> adjacentPoints = new HashSet<int>();
                while (adjacentPoints.Count < 3)
                {
                    int point = rand.Next(size);
                    if (point != i) adjacentPoints.Add(point);
                }
                adjacencyList.Add(adjacentPoints);
            }

            return adjacencyList;
```

```
        }

        private List<PointF> generatePoints(int size, Random rand)
        {
            List<PointF> points = new List<PointF>();
            for (int i = 0; i < size; i++)
            {
                points.Add(new PointF((float) (rand.NextDouble() * pictureBox.Width), (float) (rand.NextDouble() * p
ictureBox.Height)));
            }
            return points;
        }

        private void resetImageToPoints(List<PointF> points)
        {
            pictureBox.Image = new Bitmap(pictureBox.Width, pictureBox.Height);
            Graphics graphics = Graphics.FromImage(pictureBox.Image);
            Pen pen;

            if (points.Count < 100)
                pen = new Pen(Color.Blue);
            else
                pen = new Pen(Color.LightBlue);
            foreach (PointF point in points)
            {
                graphics.DrawEllipse(pen, point.X, point.Y, 2, 2);
            }

            this.graphics = graphics;
            pictureBox.Invalidate();
        }

        // These variables are instantiated after the "Generate" button is clicked
        private List<PointF> points = new List<PointF>();
        private Graphics graphics;
        private List<HashSet<int>> adjacencyList;

        // Use this to generate paths (from start) to every node; then, just return the path of interest from start
 node to end node
        private void solveButton_Click(object sender, EventArgs e)
        {
            // This was the old entry point, but now it is just some form interface handling
            bool ready = true;
```

```
            if(startNodeIndex == -1)
            {
                sourceNodeBox.Focus();
                sourceNodeBox.BackColor = Color.Red;
                ready = false;
            }
            if(stopNodeIndex == -1)
            {
                if(!sourceNodeBox.Focused)
                    targetNodeBox.Focus();
                targetNodeBox.BackColor = Color.Red;
                ready = false;
            }
            if (points.Count > 0)
            {
                resetImageToPoints(points);
                paintStartStopPoints();
            }
            else
            {
                ready = false;
            }
            if(ready)
            {
                clearSome();
                solveButton_Clicked();  // Here is the new entry point
            }
        }

        private void solveButton_Clicked()
        {
            // Initializes the HeapQueue arrays.
            g_distanceHQ = new int[points.Count];
            g_previousHQ = new int[points.Count];
            g_arrayHQ = new int[points.Count];

            // Stopwatch used to measure performance.
            Stopwatch sw = new Stopwatch();
            sw.Start();
            runHeapQueue();
            sw.Stop();
            double heapTime = sw.Elapsed.TotalMilliseconds;
            string elapsed = (heapTime/ 1000).ToString();
            heapTimeBox.Text = elapsed;
```

```
            // If the compare array box is checked, run the test on the ArrayQueue
            if (arrayCheckBox.Checked)
            {
                g_distanceAQ = new int[adjacencyList.Count];
                g_previousAQ = new int[adjacencyList.Count];
                g_arrayAQ = new int[adjacencyList.Count];

                sw.Reset();
                sw.Start();
                runArrayQueue();
                sw.Stop();
                double arrayTime = (sw.Elapsed.TotalMilliseconds);
                elapsed = (arrayTime / 1000).ToString();
                arrayTimeBox.Text = elapsed;

                double speedup = arrayTime / heapTime;
                differenceBox.Text = speedup.ToString();
            }

            display(arrayCheckBox.Checked);
        }

        // These arrays are used for the ArrayQueue. They are passed by reference in makequeue.
        int[] g_distanceAQ;         // Holds the distance values for each node. The indexes represent the nodes.
        int[] g_previousAQ;         // Holds the previous node number for each node.
                                    // if g_previousAQ[2] = 3, then there is an edge 3->2 that is currently the lowe
st cost edge to node 2
        int[] g_arrayAQ;            // Array structure that becomes the queue. Just given default values and then pa
ssed in.

        int[] g_distanceHQ;         // Same function, just for the heap
        int[] g_previousHQ;         // Same function, just for the heap
        int[] g_arrayHQ;            // Same function, just for the heap

        // Creates an ArrayQueue and calls Dykstra's with the ArrayQueue Arrays from above.
        private void runArrayQueue()
        {
            ArrayQueue queue= new ArrayQueue();
            Dykstra(queue, g_distanceAQ, g_previousAQ, g_arrayAQ);
        }

        // Creates an HeapQueue and calls Dykstra's with the ArrayQueue Arrays from above.
        private void runHeapQueue()
```

```
        {
            HeapQueue queue = new HeapQueue();
            Dykstra(queue, g_distanceHQ, g_previousHQ, g_arrayHQ);
        }

        /** Dykstra's Algorithm:
         * Dykstra(G,l,s)
            for all v in V
                dist(v) = inf
                prev(v) = null
            distance(s) = 0
            H.makeQueue(V)
            while H not empty
                u = H.deleteMin() //pops the smalles distance node
                for all (u,v) in Edges
                    if dist(v) > dist(u) + cost(u,v)
                        dist(v) = dist(u) + cost(u,v)
                        prev(v) = u
                        H.decreaseKey(e) // Tells the queue that somebody's priority has changed. Might need to bump
 it up the priority queue
         */
        private void Dykstra(IQueue queue, int[] distance, int[] previous, int[] array)
        {
            // Initialize the values in the arrays.
            for (int i = 0; i < adjacencyList.Count; i++)
            {
                distance[i] = int.MaxValue;
                previous[i] = -1;
                array[i] = 0;
            }
            // Set distance to start node to 0
            distance[startNodeIndex] = 0;
            // Call makeQueue. This has different run time depending on the type of IQueue
            queue.makeQueue(distance, previous, array, adjacencyList.Count);
            // Executes the loop until everything has been popped from the loop.
            while (!queue.isEmpty())
            {
                // Call to pop from front of priority queue.
                int node = queue.deleteMin();
                // Error Checking. This should never evaluate to true.
                if (distance[node] == int.MaxValue) return;
                // For all the nodes that this node is directionally connected to, see if the distance can be shorte
ned.
                for (int i = 0; i < adjacencyList[node].Count; i++)
```

```
                {
                    int neighbor = adjacencyList[node].ToList()[i];
                    int cost = (int) calculateCost(node, neighbor);
                    // If the current distance is greater than if it went through this node.
                    if (distance[neighbor] == int.MaxValue || distance[neighbor] > (distance[node] + cost))
                    {
                        distance[neighbor] = distance[node] + cost;
                        previous[neighbor] = node;
                        queue.decreaseKey(neighbor);
                    }
                }
            }

        }

        // This function prints the path onto the pictureBox. The boolean value determines which search's results ar
e used to print.
        // If true, it prints the results from the ArrayQueue. It's now irrelevant, as they both are completed. But
 was helpful in debugging.
        private void display(bool showArray)
        {
            int[] backtrace;
            int[] dist;
            if (showArray)
            {
                backtrace = g_previousAQ;
                dist = g_distanceAQ;
            }
            else
            {
                backtrace = g_previousHQ;
                dist = g_distanceHQ;
            }
            Graphics graphics = Graphics.FromImage(pictureBox.Image);
            Pen pen = new Pen(Color.Black);

            int currIndex = stopNodeIndex;
            // Start from the last node, and work backwards through the path using the previous array until I arrive
 at the start node.
            while (currIndex != startNodeIndex)
            {
                // This catches if there is no path from the target to the source. If backtrace[currIndex] == -1, we
 reached a root node that isn't the source node. So there's no path.
                if (backtrace[currIndex] == -1 && backtrace[currIndex] != stopNodeIndex)
```

```
                {
                    resetImageToPoints(points);
                    MessageBox.Show("There is no path from Source: " + startNodeIndex + " to Target: " + stopNodeInd
ex);
                    return;
                }
                graphics.DrawLine(pen, points[currIndex], points[backtrace[currIndex]]);
                drawDistance(ref graphics, currIndex, backtrace);
                this.graphics = graphics;
                pictureBox.Invalidate();
                currIndex = backtrace[currIndex];
            }
            // Update the total distance of the path.
            pathCostBox.Text = dist[stopNodeIndex].ToString();
        }

        // Draws the cost of each segment of the path next to the line.
        private void drawDistance(ref Graphics graphics, int currIndex, int[] backtrace)
        {
            graphics.DrawString(((int)calculateCost(currIndex, backtrace[currIndex])).ToString(),
                                                new Font("Arial", 11), new SolidBrush(Color.Black),
                                                midpoint(points[currIndex], points[backtrace[currIndex]]));
        }

        // Calculates the cost between two points by using the pythagorean theorem.
        private double calculateCost(int _node, int _neighbor)
        {
            PointF node = points[_node];
            PointF neighbor = points[_neighbor];
            float rise = node.Y - neighbor.Y;
            float run = node.X - neighbor.X;
            return (Math.Sqrt(rise * rise + run * run));
        }

        // Calculates the midpoint between two points. Used to draw the costs.
        private PointF midpoint(PointF p1, PointF p2)
        {
            return new PointF((p1.X + p2.X) / 2, (p1.Y + p2.Y) / 2);
        }

        private Boolean startStopToggle = true;
        private int startNodeIndex = -1;
        private int stopNodeIndex = -1;
        private void pictureBox_MouseDown(object sender, MouseEventArgs e)
```

```
        {
            if (points.Count > 0)
            {
                Point mouseDownLocation = new Point(e.X, e.Y);
                int index = ClosestPoint(points, mouseDownLocation);
                if (startStopToggle)
                {
                    startNodeIndex = index;
                    sourceNodeBox.ResetBackColor();
                    sourceNodeBox.Text = "" + index;
                }
                else
                {
                    stopNodeIndex = index;
                    targetNodeBox.ResetBackColor();
                    targetNodeBox.Text = "" + index;
                }
                resetImageToPoints(points);
                paintStartStopPoints();
            }
        }

        private void sourceNodeBox_Changed(object sender, EventArgs e)
        {
            if (points.Count > 0)
            {
                try{ startNodeIndex = int.Parse(sourceNodeBox.Text); }
                catch { startNodeIndex = -1; }
                if (startNodeIndex < 0 | startNodeIndex > points.Count-1)
                    startNodeIndex = -1;
                if(startNodeIndex != -1)
                {
                    sourceNodeBox.ResetBackColor();
                    resetImageToPoints(points);
                    paintStartStopPoints();
                    startStopToggle = !startStopToggle;
                }
            }
        }

        private void targetNodeBox_Changed(object sender, EventArgs e)
        {
            if (points.Count > 0)
            {
```

```
                try { stopNodeIndex = int.Parse(targetNodeBox.Text); }
                catch { stopNodeIndex = -1; }
                if (stopNodeIndex < 0 | stopNodeIndex > points.Count-1)
                    stopNodeIndex = -1;
                if(stopNodeIndex != -1)
                {
                    targetNodeBox.ResetBackColor();
                    resetImageToPoints(points);
                    paintStartStopPoints();
                    startStopToggle = !startStopToggle;
                }
            }
        }

        private void paintStartStopPoints()
        {
            if (startNodeIndex > -1)
            {
                Graphics graphics = Graphics.FromImage(pictureBox.Image);
                graphics.DrawEllipse(new Pen(Color.Green, 6), points[startNodeIndex].X, points[startNodeIndex].Y, 1,
 1);
                this.graphics = graphics;
                pictureBox.Invalidate();
            }

            if (stopNodeIndex > -1)
            {
                Graphics graphics = Graphics.FromImage(pictureBox.Image);
                graphics.DrawEllipse(new Pen(Color.Red, 2), points[stopNodeIndex].X - 3, points[stopNodeIndex].Y -
3, 8, 8);
                this.graphics = graphics;
                pictureBox.Invalidate();
            }
        }

        private int ClosestPoint(List<PointF> points, Point mouseDownLocation)
        {
            double minDist = double.MaxValue;
            int minIndex = 0;

            for (int i = 0; i < points.Count; i++)
            {
                double dist = Math.Sqrt(Math.Pow(points[i].X-mouseDownLocation.X,2) + Math.Pow(points[i].Y - mouseDo
wnLocation.Y,2));
```

```
            if (dist < minDist)
            {
                minIndex = i;
                minDist = dist;
            }
        }

        return minIndex;
    }
}
```

ArrayQueue.cs

```
class ArrayQueue : IQueue
{
    int[] distance;
    int[] previous;
    int[] queue;
    int size = 0;
    int removed = 0;

    public void makeQueue(int[] _distance, int[] _previous, int[] _array, int numNodes)
    {
        distance = _distance;
        previous = _previous;
        queue = _array;
        size = numNodes;
    }

    public int deleteMin()
    {
        long minDistance = long.MaxValue;
        int minNode = -1;
        for (int i = 0; i < size; i++)
        {
            if (distance[i] < minDistance && queue[i] != -1)
            {
                minDistance = distance[i];
                minNode = i;
            }
        }
        if (minNode != -1)
        {
            queue[minNode] = -1;    // effectively removes it from the queue.
```

```
                removed++;
            }
            return minNode;
        }

        public void insert(int node)
        {
            // Not necessary, because the queue was created with everything in it.
            queue[size] = node;
            size++;
        }

        public void decreaseKey(int node)
        {
            // do nothing. Because I iterate over the entire distance array to find the minimum ever time.
        }

        public bool isEmpty()
        {
            return removed == size;
        }

    }
```

HeapQueue.cs

```
    class HeapQueue : IQueue
    {
        int[] previous;
        int[] distance;
        int[] queue;
        int[] indexes;
        int size = 0;
        int numRemoved = 0;


    // Runs in O(|V|) time.
    public void makeQueue(int[] _distance, int[] _prev, int[] _array, int _numNodes)
        {
            distance = _distance;
            previous = _prev;
            indexes = new int[_numNodes];
            queue = new int[_numNodes + 1];
            queue[0] = 0;
```

```
            for (int i = 0; i < _numNodes; i++)
            {
                insert(i);
            }
        }

        // Inserts a node at the end of the queue, bubbles it up to as high as it should go according to its distanc
e.
        public void insert(int node)
        {
            bubbleup(node, queue[0]);
            queue[0]++;
        }

        // called when a node's distance has been changed. Checks if it should be moved in the queue.
        public void decreaseKey(int node)
        {
            int index = findIndex(node);
            bubbleup(node, index);
        }

        // Returns the root node, and then rearranges the queue by putting the last element in the first position, a
nd then sifting it down.
        public int deleteMin()
        {
            if (queue[0] == 0)
            {
                return -1;
            }
            else
            {
                int node = queue[1];
                siftdown(queue[queue[0]], 1);
                queue[0]--;
                return node;
            }
        }

        // The size is stored at queue[0]. Allows it to be accessed quickly, and made the queue indexed by 1, which
 simplified the math.
        public bool isEmpty()
        {
            return queue[0] == 0;
        }
```

```
// Moves a node up the tree until it has less cost than it's children, but more than it's parent.
private void bubbleup(int node, int position)
{
    int parent = position / 2;
    while (position != 1 && distance[queue[parent]] > distance[node])
    {
        queue[position] = queue[parent];
        indexes[queue[parent]] = position;
        position = parent;
        parent = position/ 2;
    }
    queue[position] = node;
    indexes[node] = position;
}

// Sifts down the tree, looking for the appropriate place to insert the node.
private void siftdown(int node, int position)
{
    int child = minchild(position);
    while (child != 0 && distance[queue[child]] < distance[node])
    {
        queue[position] = queue[child];
        indexes[queue[child]] = position;
        position = child;
        child = minchild(position);
    }
    queue[position] = node;
    indexes[node] = position;
}

// Helper function. Finds the child with the smallest distance.
private int minchild(int pos)
{
    if (2 * pos > queue[0])
        return 0;       // no children
    else if (2 * pos + 1 > queue[0])
        return 2 * pos;
    else
        return distance[queue[2 * pos]] > distance[queue[2 * pos + 1]] ? 2 * pos + 1 : 2 * pos;
}

// Helper function to help find index of a node. Runs in O(|1|). Called by decrease key.
private int findIndex(int node)
```

```
        {
            return indexes[node];
        }

    }
```

2. Implement the two priority queue versions. Convince that it runs in the optimal time.

Array Queue:

It was fairly easy to get the `ArrayQueue` to have constant time in the `insert()` and `decreasekey()` functions. In fact, they were largely unnecessary, due to the structure of the `deletemin()` function. We don't actually care about keeping the nodes in order in the array, because we will just loop over all the nodes and check their distances in order to find the next one to pop off the queue. In Dykstra's algorithm, insert is only called when makeQueue is called. And because we don't care about the the order of the array, I just set the queue array equal to the array that is passed into the makeQueue function. Indexes 0 to n-1 represent the nodes, and the values at indexes 1-n are are either 0 (that node is in the queue and has not been visited yet) or -1 (no longer in the queue). All nodes start in the queue.

Thus, `insert()` and `decreaseKey()` run in `O(1)`. Because they don't actually have to do anything in the `ArrayQueue`. The array never gets reordered, and if I needed to insert into the queue, I could use the size to insert at the end of the arrays.

DeleteMin() on the other hand, runs in `O(|V|)` where `V` is the number of nodes, because it must iterate over the entire `distance` array to find the minimum cost node that is still on the queue.

Heap Queue:

The arrays are managed in a more complex way in the `HeapQueue`. Firstly, to simplify a lot of the indexing math, I made the queue[] array store the size of the queue at `queue[0]` and the queue itself start at `queue[1]`. This helped avoid dividing by zero, and allowed for simple tree traversal from parent to child and vice versa.

The queue[] array is reordered with each `insert()`, `deletemin()`, and with some `decreaseKey()` calls. Optimally, each operation runs in `O(log(|V|))` time. But this is assuming that you can index directly to any desired node. I did not do that, so for my `decreaseKey()` function, the run time is `O(|V| + log(|V|))` which simplifies to `O(|V|)`. The optimal solution would store an array with pointers to each node that are updated every time that they are placed at a different index.

`Insert()` and `deleteMin()` are both `O(log(|V|))`, as they both traverse array as a tree, jumping from level to level, not touching every node. Each child node for a parent node at index i is at indexes $2i$ and $2i + 1$ respectively. The reflexive statement also holds. Each parent node of a child node at index j is at $j/2$. This property is taken advantage of by both of these functions, when they call `siftdown()` and `bubbleup()`.

3. Time & Space Complexity

Array Queue Implementation:

Time Complexity: `O(|V|^2 + O|V|)`, which can be simplified to `O(|V|^2)`

- Initially, the nodes must all be looped over to initialize the distance and previous arrays. This subsection is common to both queue implementations, and runs in `O(|V|)`.
- Next, `makeQueue()` is called. For the array queue, this is completed in `O(1)`.

- Then, while the queue is not empty. As each node will be visited one time, this outer loop is executed `O(|V|)` times. Inside this loop, `deleteMin()` is called. For the array, this runs in `O(|V|)` time. Then the inner loop is executed m times, where *m* is the number of adjacent nodes. For our exercise it was 3. Inside that nested for loop, `decreaseKey()` can be called a maximum of *m*|V| times, and has the run time of O(1). So, for the whole while loop, the time complexity is `O(|V|^2)` and is polynomial.

Space Complexity: `O(3|V| + |E|)` which simplifies to `O(|V| + |E|)`.

- The only space that is consumed are the three arrays used to store the distance, previous, and queue values. Each is of length `|V|`, so the total space is `3|V|`.

Heap Queue Implementation:

Time Complexity: O((|V| + |E|) log(|V|)

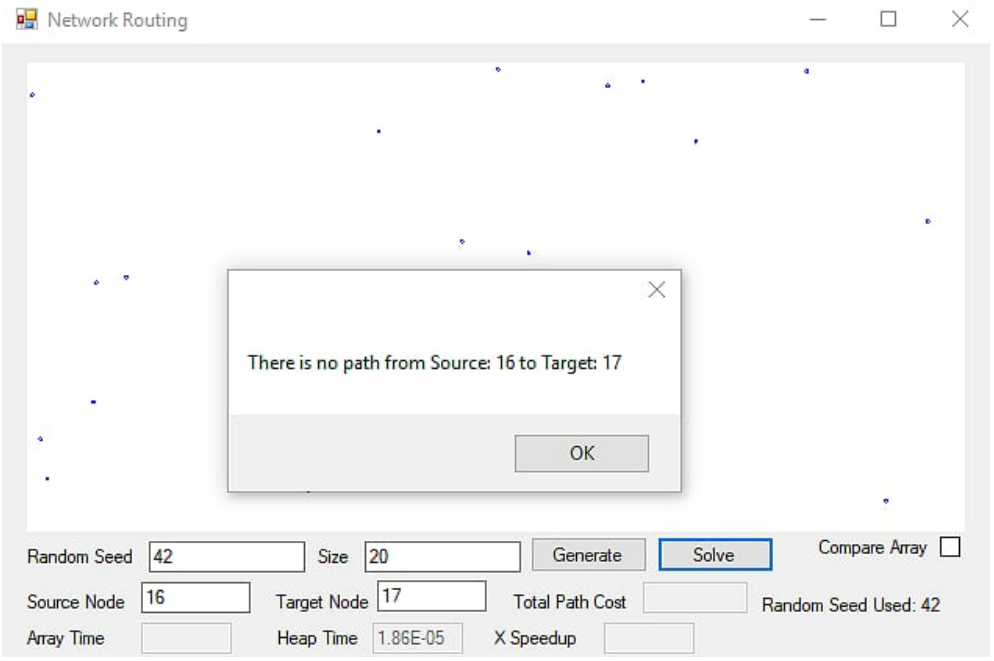
- Initially, the nodes must all be looped over to initialize the distance and previous arrays. This subsection is common to both queue implementations, and runs in `O(|V|)`.
- Next, `makeQueue()` is called. For the heap queue, this is completed in `O(|V| * log(|V|))` time. The reason is that `makeQueue()` calls `insert()` `|V|` times, and insert runs in `O(log(|V|))` time.
- Then, while the queue is not empty, the search loop is executed. It will pop each node from the queue one and only one time. So it will be executed `|V|` times. Everything inside is multiplied by this. `deleteMin()` is called, and it has a run time of `O(log(|V|)`, for reasons stated above in section 2. Then, for each adjacent node, `decreaseKey()` may be called. This runs in `O(log(|V|))`. I optimized this by using an additional array to keep track of where nodes were in the queue.

Space Complexity: `O(|V| + |E|)`

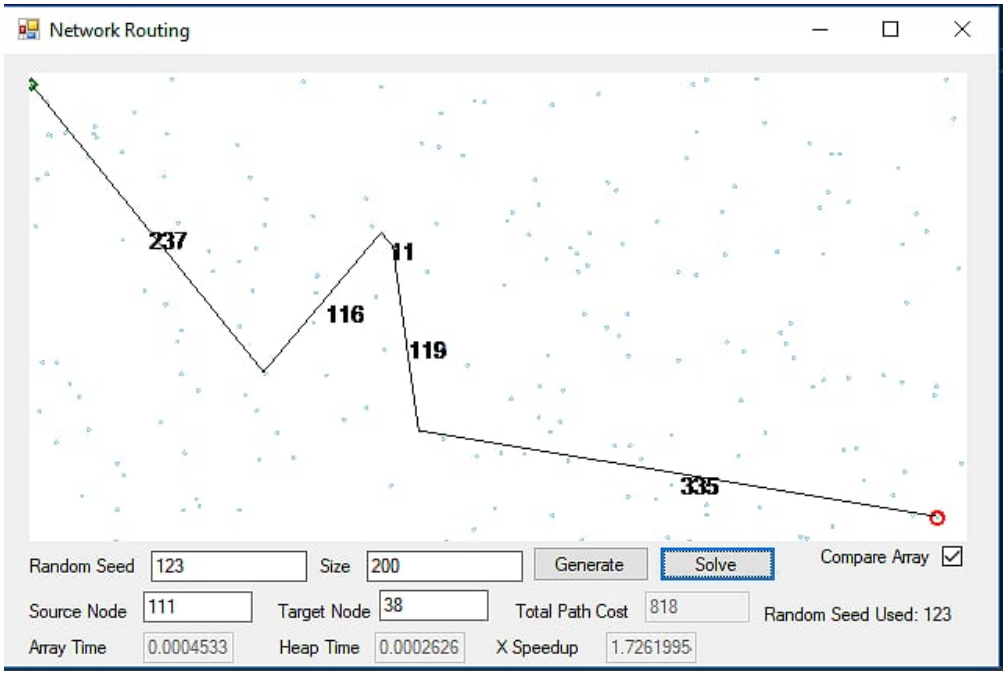
- The space that is consumed is from the arrays of size |V|. There is one additional array for the heap than for the array queue.

4. Screenshots

a. Random seed 42 - Size 20, 16 to 17:



b. Random seed 123 - Size 200, 111 to 38



5. Empirical Tests

I'm not sure how to do these estimates... So I just plugged the number of nodes into my time complexity formulas. My estimated speedups are way off with what I actually measure, even though I found the same time complexity as the book.

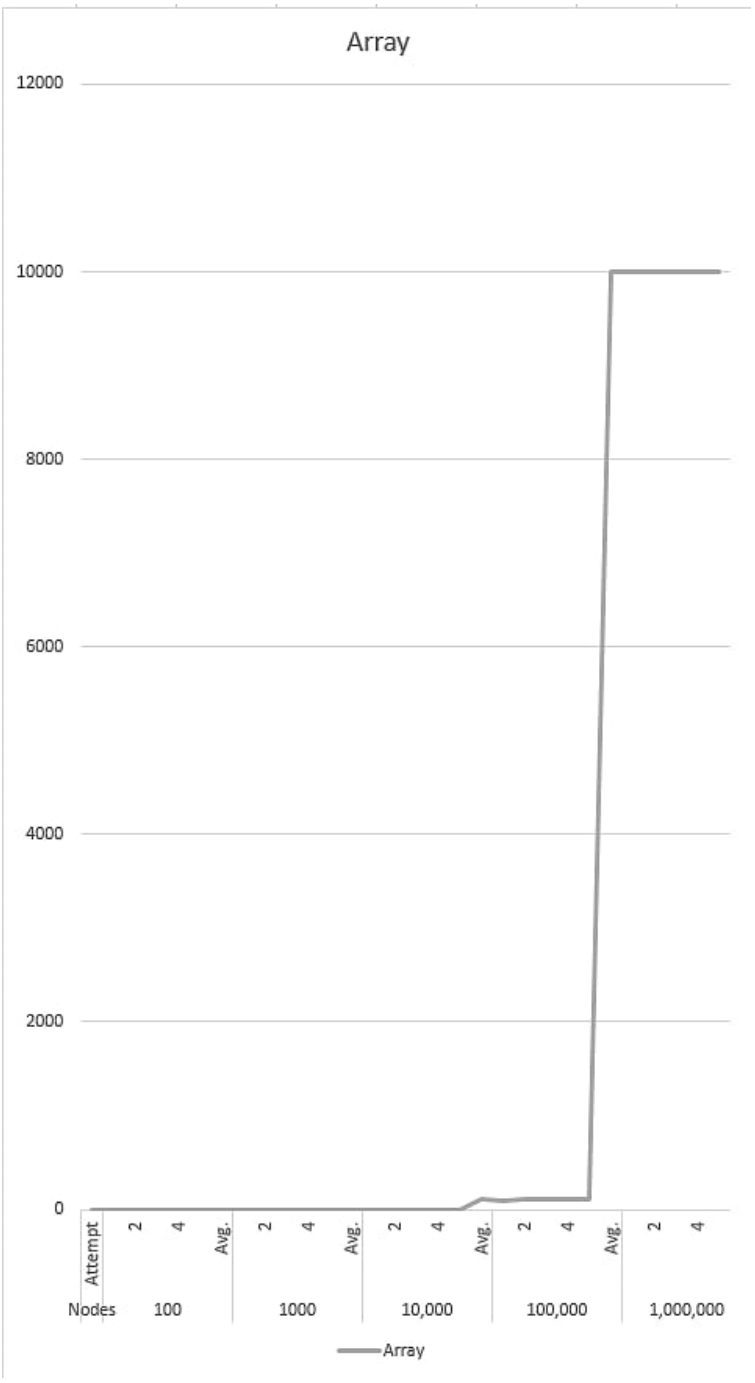
# Nodes	Heap Complexity	Array Complexity	Est. Difference
100	206	10,000	48.54
1,000	3,009	1,000,000	33.323
10,000	400,012	100,000,000	250
100,000	500015	10,000,000,000	20,000
1,000,000	6000018	1,000,000,000,000	166,667

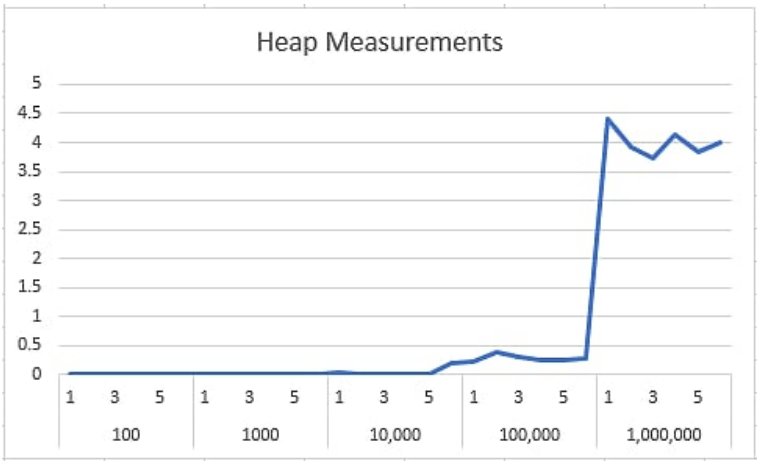
As the input size increased, the disparity in run-time between the two priority queues increased at an increasing rate. This in part could be due to some unidentified failures to optimize the array, but overall this was expected as the array queue loops over the entire set of nodes every time `deletemin()` is called. With the optimizations for `decreasekey()` in the heap queue, it handles the increase in input substantially better, as it only need traverse the levels of the tree, which takes `O(log(|V|))`

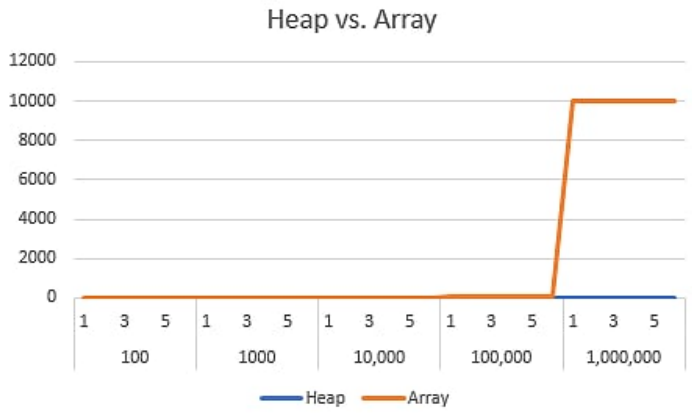
Table:

Nodes	Attempt	Heap (seconds)	Array (seconds)
100	1	0.0000857	0.0001704
	2	0.0000934	0.0001773
	3	0.000094	0.0001907
	4	0.0000824	0.0001363
	5	0.0001442	0.000207
Avg.		0.0000994	0.00017634
1000	1	0.00098	0.0151431
	2	0.0010064	0.116847
	3	0.0009918	0.0144411
	4	0.0011913	0.129371
	5	0.0025281	0.0095669
Avg.		0.0013395	0.05707382
10,000	1	0.0233315	1.0089219
	2	0.0180524	1.0295572
	3	0.0202991	1.0330499
	4	0.0175896	1.0069662
	5	0.0166418	1.0290474
Avg.		0.191829	1.02150852
100,000	1	0.2346402	101.72230
	2	0.3804675	99.058846
	3	0.3020957	101.42583
	4	0.2520122	101.75914
	5	0.2604811	103.42154
Avg.		0.2859393	101.477531
1,000,000	1	4.4103186	10,000
	2	3.9290851	(estimate)
	3	3.7368376	
	4	4.1222126	
	5	3.8433245	
Avg.		4.0083557	

Graphs:


Array
12000
10000
8000
6000
4000
2000
0
Attempt
2
4
Avg.
Nodes
100
1000
10,000
100,000
1,000,000
Array


Heap Measurements
5
4.5
4
3.5
3
2.5
2
1.5
1
0.5
0
1
3
5
100
1000
10,000
100,000
1,000,000


Heap vs. Array
12000
10000
8000
6000
4000
2000
0
1 3 5 1 3 5 1 3 5 1 3 5 1 3 5
100
1000
10,000
100,000
1,000,000
Heap
Array