```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Drawing.Drawing2D;
using System.Linq;
using System.Text;
using System.Windows.Forms;

namespace NetworkRouting
{
    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() * pictureBox.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;
        private double[] distance;                  // Stores the distances of the
          points
        private int[] previous;                     // Stores how to draw lines


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


            // First do the heap. time it
            System.Diagnostics.Stopwatch stopwatchHEAP =
              System.Diagnostics.Stopwatch.StartNew(); //creates and start the
              instance of Stopwatch

            heapCALCULATE();

            stopwatchHEAP.Stop();
            long heapTicks = stopwatchHEAP.ElapsedTicks;
            double heapSecs = ((double)stopwatchHEAP.ElapsedMilliseconds) / 1000;
            Console.WriteLine("heap ticks:" + heapTicks);
            Console.WriteLine("heap ms:" + heapSecs);
            heapTimeBox.Text = heapSecs.ToString();



            // Then do the array. Time it.
            System.Diagnostics.Stopwatch stopwatchARRAY =
              System.Diagnostics.Stopwatch.StartNew(); //creates and start the
              instance of Stopwatch

            arrayCALCULATE();

            stopwatchARRAY.Stop();
            long arrayTicks = stopwatchARRAY.ElapsedTicks;
            double arraySecs = ((double)stopwatchARRAY.ElapsedMilliseconds) / 1000;
            Console.WriteLine("array ticks:" + arrayTicks);
            Console.WriteLine("array ms:" + arraySecs);
            arrayTimeBox.Text = arraySecs.ToString();

            differenceBox.Text = ((double)arrayTicks/ (double)heapTicks).ToString();
```

```
            //differenceBox.Text = (arraySecs/heapSecs).ToString();

            // now we have answers
            Console.WriteLine("Last node " + stopNodeIndex + " distance: " + distance
              [stopNodeIndex]);
            int tamp = stopNodeIndex;

            SolidBrush brush = new SolidBrush(Color.Black);     // For writing
            Font font = new Font("Arial", 10);                  // The number
            PointF p = new PointF();                            // next to line
            while (tamp != startNodeIndex) {
                Console.WriteLine(tamp + " connects to ");
                // Draws lines between points
                this.graphics.DrawLine(new Pen(Color.SlateBlue), points[tamp], points
                  [previous[tamp]]);
                // draw the number next to the line
                string dist = distanceBetweenNodes(points[tamp], points[previous
                  [tamp]]).ToString("#.##");
                p.X = (points[tamp].X + points[previous[tamp]].X)/2;    // Gets
                  midpoint
                p.Y = (points[tamp].Y + points[previous[tamp]].Y)/2;    // of the line
                this.graphics.DrawString(dist, font, brush, p);         // Draws the
                  number
                tamp = previous[tamp];
            }
            Console.WriteLine(tamp);

            pathCostBox.Text = distance[stopNodeIndex].ToString();      // put total
              path in
        }

        // A simple function to get the distance between two points
        public static double distanceBetweenNodes(PointF point1, PointF point2) {
            double a = point2.X - point1.X;
            double b = point2.Y - point1.Y;
            return Math.Sqrt(a * a + b * b);
        }

        // Gets the next closest unvisited node. Returns -1 if there is no node to
          visit
        // Currently doing a naive search through the entire list every time.
        private int getNextNode(HashSet<int>unvisited, double[] distance) {//
          List<double>distance) {
            double currentBest = double.PositiveInfinity;
            int currentBestIndex = -1;

            for (int i = 0; i< distance.Length; i++) {
                if (distance[i] != double.PositiveInfinity && unvisited.Contains(i) &&
                   distance[i] < currentBest) {
                    currentBest = distance[i];
                    currentBestIndex = i;
                }
```

```
            }
            return currentBestIndex;
        }

        //                                                                          ➥
          ***************************************************************************➥
          ***************
        // This is the priority list array implementation
        int[] priorityArray;                        // Stores the points in distance   ➥
          order
        int[] priorityArrayPointers;                // Points to where in array point  ➥
          is
        int priorityArrayIndex;
        private void initPriorityArray() {
            priorityArrayIndex = 0;

            priorityArray = new int[points.Count];
            priorityArrayPointers = new int[points.Count];
            for (int i=0;i<points.Count;i++) {
                priorityArray[i] = i;               // Make every point its index
                priorityArrayPointers[i] = i;
            }

            priorityArray[0] = startNodeIndex;  // But the highest-priority is start
            priorityArrayPointers[startNodeIndex] = 0;
            priorityArray[startNodeIndex] = 0;  // gotta put zero somewhere
            priorityArrayPointers[0] = startNodeIndex;
        }

        // for debugging. Prints the priority array
        private void printPriorityArray() {
            Console.Write("printing priority array:\n");
            for (int i=0; i < priorityArray.Length; i++) {
                if (i == priorityArrayIndex) {
                    Console.Write("|start| ");
                }
                Console.Write(priorityArray[i] + " ");
            }
            Console.Write("\n\n");


            Console.Write("printing POINTERS:\n");
            for (int i = 0; i < priorityArrayPointers.Length; i++) {
                Console.Write(priorityArrayPointers[i] + " ");
            }
            Console.Write("\n\n");
        }

        // O(1). Don't need to change the array at all.
        private int getNextNodeArray() {
            // Return the next one and increment the index
            if (priorityArray[priorityArrayIndex] == double.PositiveInfinity)
```

```
            return -1;                          // If we can't reach the next node, give up.
        return priorityArray[priorityArrayIndex++];
    }

    // Given a point id, we need to sort it again in the queue array.
    // It is a bubble sort that only happens in one direction, ever.
    // O(n)
    private void reprioritizeArray(int changed) {
        int currentPriorityIndex = priorityArrayPointers[changed];
        // While our current thing has a higher priority than what came before it
        while (distance[priorityArray[currentPriorityIndex]] < distance[priorityArray[currentPriorityIndex-1]]
            && currentPriorityIndex > priorityArrayIndex) {
            // We need to swap our current thing with the thing before.
            int temp = priorityArray[currentPriorityIndex];
            priorityArray[currentPriorityIndex] = priorityArray[currentPriorityIndex - 1];
            priorityArray[currentPriorityIndex - 1] = temp;

            // And fix the pointers
            priorityArrayPointers[changed] = priorityArrayPointers[changed]-1;   // our current thing moved up one
            priorityArrayPointers[priorityArray[currentPriorityIndex]] = priorityArrayPointers[priorityArray[currentPriorityIndex]] + 1;   // Other thing moved back one

            currentPriorityIndex--;     // We are going back on the array
        }
    }

    // Takes O(V^2)
    private void arrayCALCULATE()
    {
        // We make our sets.
        HashSet<int> visited = new HashSet<int>();
        HashSet<int> unvisited = new HashSet<int>();

        // And our distance list
        distance = new double[points.Count];

        // And our graph-following list
        previous = new int[points.Count];


        // Now we need to set up our unvisited set and distance list
        for (int i = 0; i < points.Count; i++)
        {    // Go through all our points
            unvisited.Add(i);                       // Add that point to the unvisited set
            distance[i] = double.PositiveInfinity;  // distances are infinity
            previous[i] = -1;                       // Tree is not at all
```

```
            connected at the start
        }

        // FOR THE PRIORITYARRAY IMPLEMENTATION
        initPriorityArray();                        // Need to init our array

        visited.Add(startNodeIndex);                // Start by visiting the first node
        unvisited.Remove(startNodeIndex);           // So it isn't unvisited anymore
        distance[startNodeIndex] = 0;               // Distance to the start node is zero!

        int currentNode = currentNode = getNextNodeArray(); //startNodeIndex;           // Start at the beginning!

        //Can happen up to O(V) times
        while (!visited.Contains(stopNodeIndex))
        {  // As soon as we've visited the end node, we've won!
            // 2. For all the (unvisited) nodes it can go to, update their distance if it is now less from the current node

            // Only ever happens 3x max. O(1)
            foreach (int outNode in adjacencyList[currentNode])
            {
                if (unvisited.Contains(outNode)     // If outNode is not yet visited and the new distance is less than the current distance
                    && distanceBetweenNodes(points[currentNode], points[outNode]) + distance[currentNode] < distance[outNode]
                    )
                {
                    // Store the new distance!
                    distance[outNode] = distanceBetweenNodes(points[currentNode], points[outNode]) + distance[currentNode];
                    previous[outNode] = currentNode;// Set up graph.

                    //Takes O(V)
                    reprioritizeArray(outNode);     // Gotta fix outNode's priority now.
                }
            }


            // 1. visit the next closest unvisited node
            // O(1)
            currentNode = getNextNodeArray();
            if (currentNode == -1)
            {
                Console.WriteLine("CANNOT VISIT ANY MORE NODES");
                return;
            }
            visited.Add(currentNode);               // We are now visiting this
```

```
                node
            unvisited.Remove(currentNode);          // so it is not unvisited any
                more
        }
    }

    // End the priority list array implementation
    //
      ***************************************************************************
      ***************


    //
      ***************************************************************************
      ***************
    // This is the HEAP implementation
    int[] priorityHeap;                         // Doing our heap as an array!
    int[] priorityHeapPointers;                 // Where in the array is any given
       point
    int priorityHeapLAST;                       // Used for deleteMin- index of
      last thing in the heap
    private void initPriorityHeap() {
        priorityHeapLAST = points.Count;        // Just the last thing [shrugs]
        priorityHeap = new int[points.Count+1]; // Because we start at index 1
        priorityHeapPointers = new int[points.Count];
        for (int i=0;i<points.Count;i++) {
            priorityHeap[i + 1] = i;            // +1 because we are 1-indexing it
            priorityHeapPointers[i] = i+1;      // Things are at +1 their own
                index.
        }
        priorityHeap[0] = -1;                   // Error value
        priorityHeap[1] = startNodeIndex;       // swap start node
        priorityHeapPointers[startNodeIndex]=1; // and 0
        priorityHeap[startNodeIndex + 1] = 0;
        priorityHeapPointers[0] = startNodeIndex + 1;
    }

    // This is deleteMin for our heap,
    // O(log(V))
    private int getNextNodeHeap() {
        // 1 Replace root with last node
        int highestPriority = priorityHeap[1];
        priorityHeap[1] = priorityHeap[priorityHeapLAST];
        priorityHeapPointers[highestPriority] = priorityHeapLAST;
        priorityHeapPointers[priorityHeap[1]] = 1;
        priorityHeap[priorityHeapLAST] = highestPriority;

        // 2 Remove the last node
        priorityHeapLAST--;                     // Cause we do this

        // 3 Swap new root with its child until correct position
        int l, r;
```

```
            int currentIndex = 1;

            while (true) {                          // We only escape by breaking
                if (currentIndex * 2 > priorityHeapLAST) {
                    break;                          // current has no children
                } else if (currentIndex * 2 == priorityHeapLAST) {
                    l = priorityHeap[currentIndex * 2];
                    if (distance[l] < distance[priorityHeap[currentIndex]]) {
                        // l is higher priority than current!
                        // SWAP
                        swapHeap(currentIndex * 2, currentIndex);
                    }
                    break;                          // won't be no more children
                }
                l = priorityHeap[currentIndex * 2];
                r = priorityHeap[(currentIndex * 2) + 1];
                if (distance[l] < distance[r]) {
                    // l is higher priority than r
                    if (distance[l] < distance[priorityHeap[currentIndex]]) {
                        // l is higher priority than current!
                        // SWAP
                        swapHeap(currentIndex*2, currentIndex);
                        currentIndex = currentIndex * 2;
                    } else {
                        break;
                    }
                } else {
                    // r is higher (or equal) priority than r
                    if (distance[r] < distance[priorityHeap[currentIndex]]) {
                        // r is higher priority than current!
                        // SWAP
                        swapHeap((currentIndex*2)+1, currentIndex);
                        currentIndex = currentIndex * 2+1;
                    } else {
                        break;
                    }
                }
            }

            if (distance[highestPriority] == double.PositiveInfinity) {
                return -1;
            }
            return highestPriority;
        }

        // Swaps two nodes of the heap
        // O(1)
        private void swapHeap(int a, int b) {
            int temp = priorityHeap[a];
            priorityHeap[a] = priorityHeap[b];
            priorityHeap[b] = temp;
```

```
            priorityHeapPointers[temp] = b;
            priorityHeapPointers[priorityHeap[a]] = a;
        }

        // This reprioritizes the point IDed by changed
        // O(log(V))
        private void reprioritizeHeap(int changed) {

            int currentIndex = priorityHeapPointers[changed];
            while (true) {
                if (currentIndex / 2 == 0) {
                    // can't get higher priority than this
                    return;
                }

                if (distance[priorityHeap[currentIndex]] < distance[priorityHeap
                  [currentIndex/2]]) {
                    // The current node is higher priority than its parent!
                    swapHeap(currentIndex, currentIndex / 2);
                    currentIndex = currentIndex / 2;
                } else {
                    return;
                }
            }
        }

        private void printHeap()
        {
            Console.Write("printing heap:\n");
            for (int i = 0; i < priorityHeap.Length; i++)
            {
                if (i == priorityHeapLAST)
                {
                    Console.Write("|end| ");
                }
                Console.Write(priorityHeap[i] + " ");
            }
            Console.Write("\n\n");


            Console.Write("printing POINTERS:\n");
            for (int i = 0; i < priorityHeapPointers.Length; i++)
            {
                Console.Write(priorityHeapPointers[i] + " ");
            }
            Console.Write("\n\n");
        }

        // O(logV) things happening O(V) times?
        // O(VlogV)
        private void heapCALCULATE() {
            // We make our sets.
```

```
            HashSet<int> visited = new HashSet<int>();
            HashSet<int> unvisited = new HashSet<int>();

            // And our distance list
            distance = new double[points.Count];

            // And our graph-following list
            previous = new int[points.Count];


            // Now we need to set up our unvisited set and distance list
            for (int i = 0; i < points.Count; i++)
            {    // Go through all our points
                unvisited.Add(i);                       // Add that point to the
                  unvisited set
                distance[i] = double.PositiveInfinity;  // distances are infinity
                previous[i] = -1;                       // Tree is not at all
                  connected at the start
            }

            initPriorityHeap();

            visited.Add(startNodeIndex);                // Start by visiting the first
               node
            unvisited.Remove(startNodeIndex);           // So it isn't unvisited
              anymore
            distance[startNodeIndex] = 0;               // Distance to the start node
              is zero!

            int currentNode = getNextNodeHeap();

            // This goes O(V) times
            while (!visited.Contains(stopNodeIndex))
            {  // As soon as we've visited the end node, we've won!
                // 2. For all the (unvisited) nodes it can go to, update their
                  distance if it is now less from the current node
                foreach (int outNode in adjacencyList[currentNode])
                {
                    if (unvisited.Contains(outNode)     // If outNode is not yet
                      visited and the new distance is less than the current distance
                        && distanceBetweenNodes(points[currentNode], points[outNode])
                      + distance[currentNode] < distance[outNode]
                        )
                    {
                        // Store the new distance!
                        distance[outNode] = distanceBetweenNodes(points[currentNode],
                      points[outNode]) + distance[currentNode];
                        previous[outNode] = currentNode;// Set up graph.

                        // max O(logV)
                        reprioritizeHeap(outNode);
                    }
```

```
            }


            // 1. visit the next closest unvisited node
            // Max O(logV)
            currentNode = getNextNodeHeap();
            if (currentNode == -1)
            {
                Console.WriteLine("CANNOT VISIT ANY MORE NODES");
                return;
            }
            visited.Add(currentNode);               // We are now visiting this
              node
            unvisited.Remove(currentNode);          // so it is not unvisited any
              more
        }
    }

    // end HEAP implementation
    //
      ***************************************************************************
      ***************


    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 - mouseDownLocation.Y,2));
                if (dist < minDist)
                {
                    minIndex = i;
                    minDist = dist;
                }
            }

            return minIndex;
        }
    }
}
```