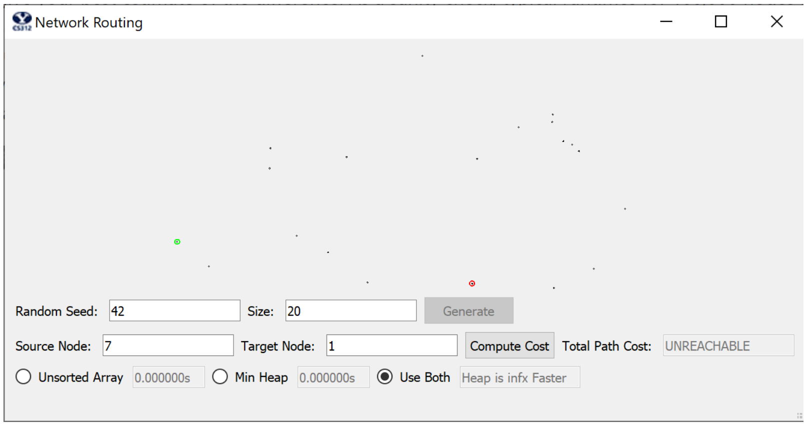
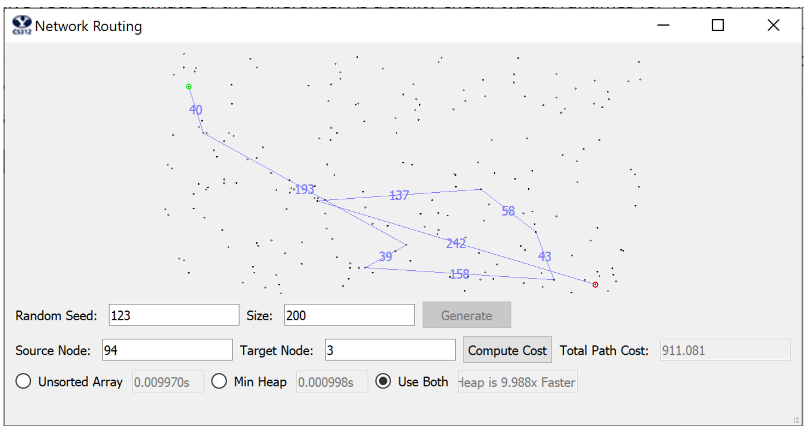
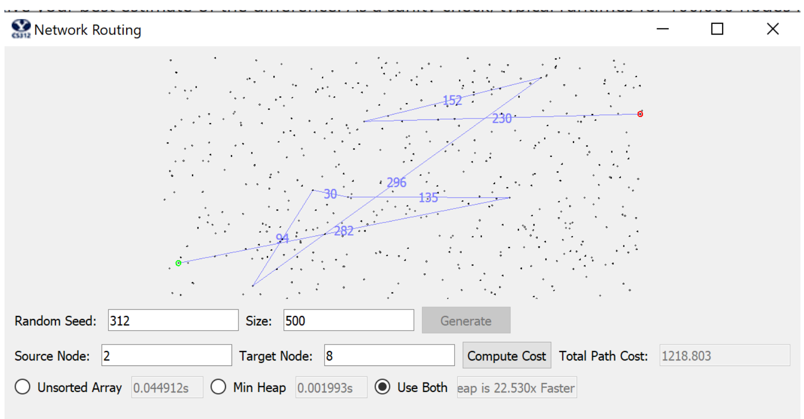
1. **See Appendix**
2. **I implemented both versions :)**
3. **Algorithm discussion**
   a. **Overall**
      ■ Time
         ● There's nothing in the methods that call dijkstrasArray() and dijkstrasHeap() that has a greater time complexity than either of them. Therefore the time complexity will depend on which method is chosen. This would give us a $O(v^2)$ since the worst option is the array implementation.
      ■ Space
         ● Space complexity is always the same, just O(v). This is because the only space we have taken up is the network which contains an array of nodes. Each node has a constant number of elements, so space complexity is just O(n)
   b. **Array**
      ■ Time
         ● Time complexity of the array implementation is $O(v^2)$ where v is the number of total points. This is because in the worst case scenario, you have to add each point to the priority queue, and take each point off. This would result in performing the deleteMin() operation v times, and deleteMin() takes v time. Thus, $O(v^2)$
      ■ Space
         ● Space complexity is always the same, just O(v). This is because the only space we have taken up is the network which contains an array of nodes. Each node has a constant number of elements, so space complexity is just O(n)
   c. **Heap**
      ■ Time
         ● Time complexity is O(vlogv), where v is the number of total points. This is because in the worst case scenario, we add each element to the priority queue and have to take it off with deleteMin(). WIth the heap implementation, deleteMin() time is only O(logv), because we use a method to keep the heap sorted and the min node at the top at all times. Thus, we would call a O(logv) method v times, and therefore O(vlogv)
      ■ Space
         ● Space complexity is always the same, just O(v). This is because the only space we have taken up is the network which contains an array of nodes. Each node has a constant number of elements, so space complexity is just O(n)
4. Screenshots
   a.


Network Routing
Random Seed: 42
Size: 20
Generate
Source Node: 7
Target Node: 1
Compute Cost
Total Path Cost: UNREACHABLE
Unsorted Array 0.000000s
Min Heap 0.000000s
Use Both
Heap is infx Faster

b.

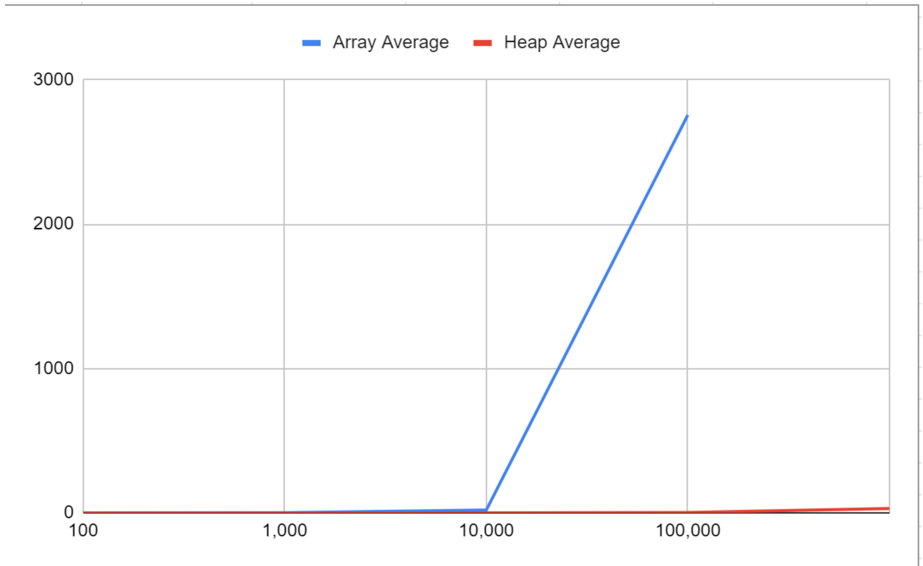

Network Routing
Random Seed: 123
Size: 200
Generate
Source Node: 94
Target Node: 3
Compute Cost
Total Path Cost: 911.081
Unsorted Array 0.009970s
Min Heap 0.000998s
Use Both
Heap is 9.988x Faster

c.


Network Routing
Random Seed: 312
Size: 500
Generate
Source Node: 2
Target Node: 8
Compute Cost
Total Path Cost: 1218.803
Unsorted Array 0.044912s
Min Heap 0.001993s
Use Both
eap is 22.530x Faster

5. **Analysis**

a. **Graph**



b. **Table**

		100	1,000	10,000	100,000	1,000,000
Array	**1**	0.001992	0.171508	18.292114	2750.36142	-
	2	0.001994	0.142653	19.256527	2723.49851	-
	3	0.004986	0.186535	19.479906	2798.17236	-
	4	0.001992	0.180555	18.257201	2714.81629	-
	5	0.004017	0.182511	18.582301	2795.97573	-
	Array Average	**0.0029962**	**0.1727524**	**18.7736098**	**2756.564862**	**550000 (Estimate)**
Heap	**1**	0.000998	0.030918	0.117722	2.014616	29.890149
	2	0	0.005985	0.125664	2.265952	28.329319
	3	0	0.004987	0.08976	2.628972	28.216634
	4	0	0.005979	0.104721	2.107331	27.139482
	5	0.000997	0.012966	0.126661	2.740671	34.838882
	Heap Average	**0.000399**	**0.012167**	**0.1129056**	**2.3515084**	**29.6828932**

c. **Discussion**

- The heap has run very predictably and very quickly, increasing by logarithmic amount each time. This seems very in line with what you would expect from the heap implementation. The array on the other hand is not what I expected. It's possible this is the effect of non-theoretical uses of time. For example, because the time complexity is v^2, it may be that I'm doing a v^2 operation a constant number of times, but if that constant is even 3 or so, it could have disastrous effects on real life results of this operation when using large numbers. This is my best explanation for why the numbers turned out the way they did.

Appendix

```
#!/usr/bin/python3
import math
import random

from CS312Graph import *
import time


class NetworkRoutingSolver:

    shortestPath = [[]]

    def __init__( self):
        pass

    def initializeNetwork( self, network ):
        assert( type(network) == CS312Graph )
        self.network = network

    def getShortestPath( self, destIndex ):
        self.dest = destIndex

        def findDestinationNodeIndex():
            for index in range(len(self.shortestPath)):
                if self.shortestPath[index][0] == destIndex:
                    return index

        path_edges = []
        total_cost = 0
        currentIndex = findDestinationNodeIndex()
        if currentIndex == None:
            return {'cost': math.inf, 'path': []}
        backPointer = self.shortestPath[currentIndex][1]

        while True:
            if backPointer == None:
                break

            while self.shortestPath[currentIndex][1] != backPointer:
                currentIndex -= 1

            nodesIndex = self.shortestPath[currentIndex][0]

            if self.network.nodes[backPointer].neighbors[0].dest.node_id ==
nodesIndex:
                edge = self.network.nodes[backPointer].neighbors[0]
```

```
            elif self.network.nodes[backPointer].neighbors[1].dest.node_id ==
nodesIndex:
                edge = self.network.nodes[backPointer].neighbors[1]
            else:
                edge = self.network.nodes[backPointer].neighbors[2]

            path_edges.append((edge.src.loc, edge.dest.loc,
'{:.0f}'.format(edge.length)))
            total_cost += edge.length

            while self.shortestPath[currentIndex][0] != backPointer:
                currentIndex -= 1
            backPointer = self.shortestPath[currentIndex][1]

        return {'cost': total_cost, 'path': path_edges}


    def computeShortestPaths( self, srcIndex, use_heap ):
        self.source = srcIndex
        t1 = time.time()

        if not use_heap:
            self.shortestPath = self.dijkstrasArray(srcIndex)
        else:
            self.shortestPath = self.dijkstrasHeap(srcIndex)

        t2 = time.time()
        return (t2-t1)


    def dijkstrasArray(self, srcIndex):

        def makeQueue(srcIndex):
            pQueue = [[UNREACHABLE, None] for _ in
range(len(self.network.nodes))]
            pQueue[srcIndex][0] = 0
            return pQueue

        def decreaseKey(pQueue, index, distance, lastNode):
            pQueue[index] = [distance, lastNode]

        def deleteMin(pQueue):

            minValue = UNREACHABLE
            minIndex = 0
            for index in range(len(pQueue)):

                if ((minValue == UNREACHABLE or pQueue[index][0] < minValue)
                        and pQueue[index][0] != UNREACHABLE
```

```
                    and pQueue[index][0] != VISITED):
                minIndex = index
                minValue = pQueue[index][0]

        pQueue[minIndex][0] = VISITED
        return minIndex, minValue

    VISITED = -2
    UNREACHABLE = -1
    shortestPath = []
    pQueue = makeQueue(srcIndex)

    while True:
        minIndex, minValue = deleteMin(pQueue)
        if minValue == UNREACHABLE:
            break

        shortestPath.append([minIndex, pQueue[minIndex][1]])
        if len(shortestPath) == len(self.network.nodes):
            break

        neighbors = self.network.nodes[minIndex].neighbors
        for index in range(3):

            neighborID = neighbors[index].dest.node_id
            neighborDistance = neighbors[index].length
            if (pQueue[neighborID][0] != VISITED
                    and ((pQueue[neighborID][0] == UNREACHABLE) or
(neighborDistance + minValue) < pQueue[neighborID][0])): # Replaced index with
neighborID in (neighborDistance + minValue) < pQueue[neighborID][0]))
                decreaseKey(pQueue, neighborID, neighborDistance +
minValue, minIndex)

    return shortestPath


  def dijkstrasHeap(self, srcIndex):

    def makeQueueAndPointerArray(srcIndex):
        pq = [[None, UNREACHABLE, None] for _ in
range(len(self.network.nodes))]  #NodeNumber, currentCost, backpointer
        pointerArray = [UNREACHABLE for _ in range(len(self.network.nodes)
+ 1)]
        pq[0] = [srcIndex, 0, None]
        pointerArray[srcIndex] = 0
        pointerArray[len(pointerArray) - 1] = 1  # Last index keeps track
of where the next Null spot is

        return pq, pointerArray
```

```
    def insert(value: [int, int, int | None]):

        nextNullSpot = pointerArray[len(pointerArray) - 1]

        pq[nextNullSpot] = value
        pointerArray[value[0]] = nextNullSpot
        pointerArray[len(pointerArray) - 1] += 1
        bubbleUp(nextNullSpot)

    def deleteMin():
        lowestItemIndex = pointerArray[len(pointerArray) - 1] - 1
        smallestNode = pq[0]
        pointerArray[smallestNode[0]] = VISITED
        pq[0] = pq[lowestItemIndex]
        if lowestItemIndex != 0:
            pointerArray[pq[0][0]] = 0
        pq[lowestItemIndex] = [None, -1, None]
        pointerArray[len(pointerArray) - 1] = lowestItemIndex

        siftDown()

        return smallestNode

    def decreaseKey(newNodeValue):
        pqIndex = pointerArray[newNodeValue[0]]
        pq[pqIndex] = newNodeValue
        bubbleUp(pqIndex)

    def siftDown():

        parentIndex = 0

        while True:
            child1Index = (2 * parentIndex) + 1
            child2Index = (2 * parentIndex) + 2
            outOfBoundsIndex = pointerArray[len(pointerArray) - 1]

            # Both children out of bounds
            if child1Index >= outOfBoundsIndex and child2Index >=
outOfBoundsIndex:
                break

            # child2 out of bounds
            elif child2Index >= outOfBoundsIndex:
                if pq[parentIndex][1] < pq[child1Index][1]:
                    break
                else:
                    swapIndex = child1Index
```

```
            # Neither child out of bounds
            else:
                if pq[parentIndex][1] < pq[child1Index][1] and
pq[parentIndex][1] < pq[child2Index][1]:
                    break
                elif pq[child1Index][1] < pq[child2Index][1]:
                    swapIndex = child1Index
                else:
                    swapIndex = child2Index

            parent = pq[parentIndex]
            child = pq[swapIndex]
            pq[parentIndex] = child
            pq[swapIndex] = parent
            pointerArray[pq[parentIndex][0]] = parentIndex
            pointerArray[pq[swapIndex][0]] = swapIndex
            parentIndex = swapIndex


    def bubbleUp(newNodeLocation):

        currentParentIndex = math.floor((newNodeLocation - 1) / 2)

        while pq[newNodeLocation][1] < pq[currentParentIndex][1]:
            childValue = pq[newNodeLocation]
            parentValue = pq[currentParentIndex]
            pq[newNodeLocation] = parentValue
            pq[currentParentIndex] = childValue

            pointerArray[parentValue[0]] = newNodeLocation
            pointerArray[childValue[0]] = currentParentIndex

            newNodeLocation = currentParentIndex
            currentParentIndex = math.floor(newNodeLocation / 2)


    UNREACHABLE = -1
    VISITED = -2
    pq, pointerArray = makeQueueAndPointerArray(srcIndex)
    shortestPath = []

    while pointerArray[len(pointerArray) - 1] != 0:
        nextNode = deleteMin()
        shortestPath.append([nextNode[0], nextNode[2]])
        neighbors = self.network.nodes[nextNode[0]].neighbors

        for index in range(3):
```

```
                neighborID = neighbors[index].dest.node_id
                neighborDistance = neighbors[index].length
                if pointerArray[neighborID] == VISITED:
                    continue
                elif pointerArray[neighborID] == UNREACHABLE:
                    insert([neighborID, neighborDistance + nextNode[1],
nextNode[0]])

                elif (neighborDistance + nextNode[1]) <
pq[pointerArray[neighborID]][1]:
                        decreaseKey([neighborID, neighborDistance +
nextNode[1], nextNode[0]])

        return shortestPath
```