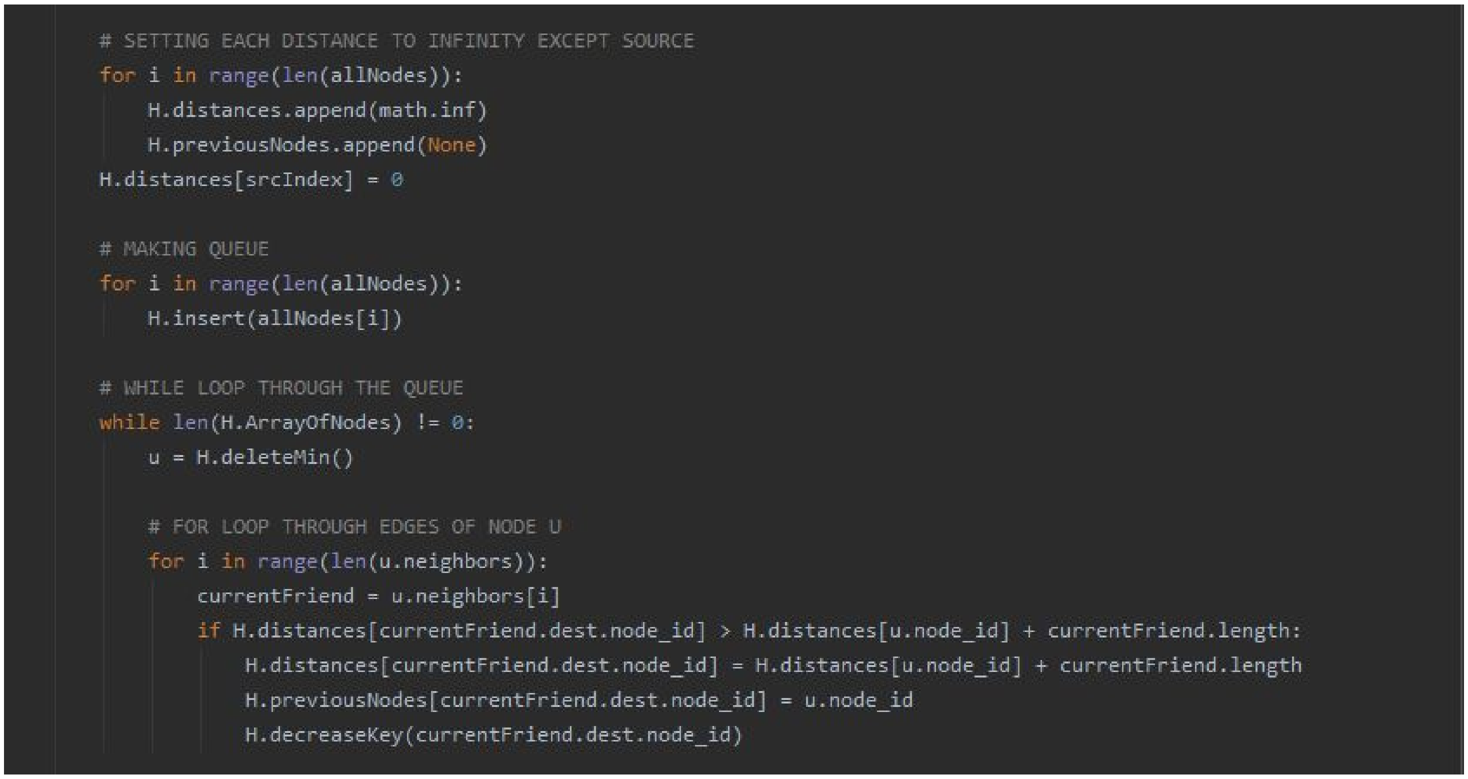
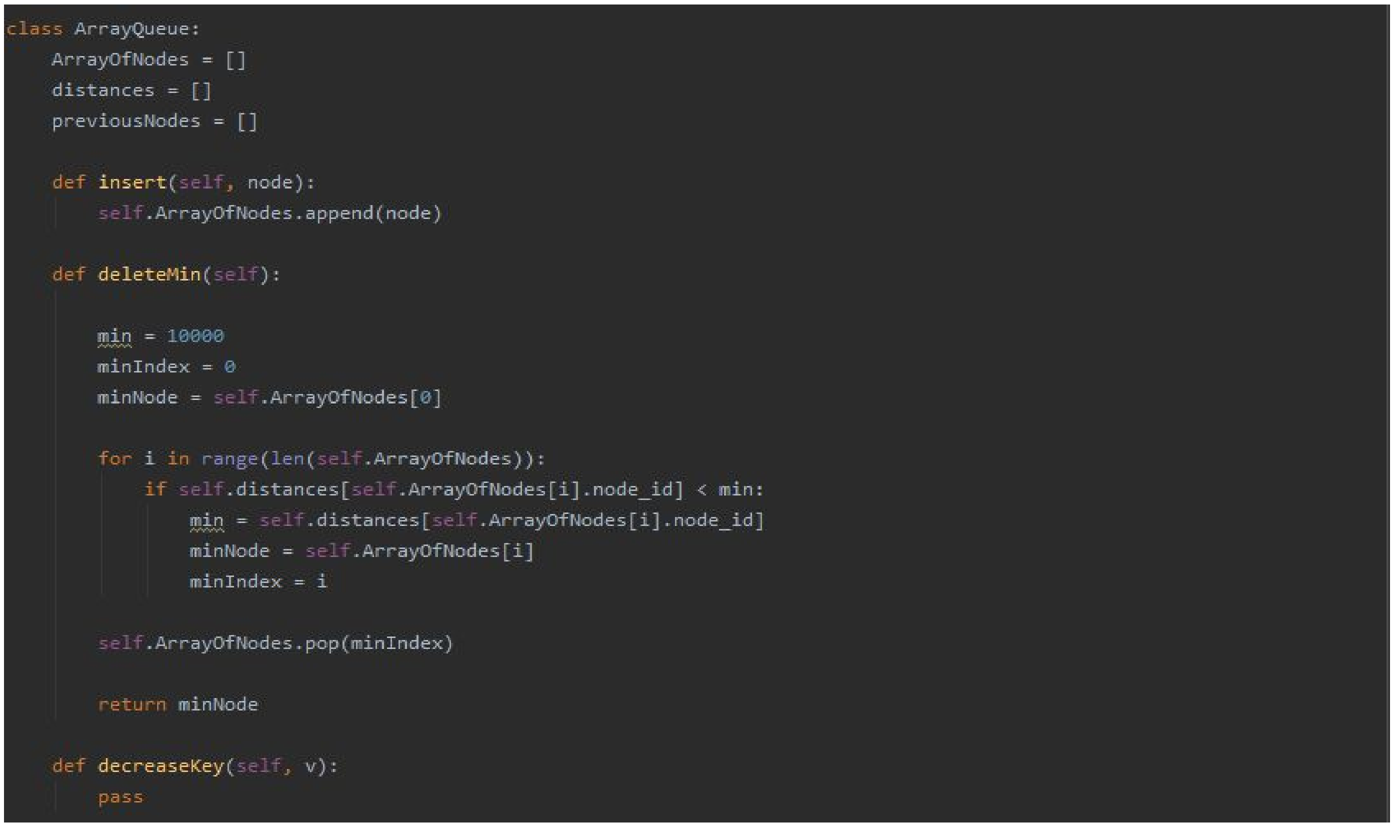

```
        # SETTING EACH DISTANCE TO INFINITY EXCEPT SOURCE
        for i in range(len(allNodes)):
            H.distances.append(math.inf)
            H.previousNodes.append(None)
        H.distances[srcIndex] = 0

        # MAKING QUEUE
        for i in range(len(allNodes)):
            H.insert(allNodes[i])

        # WHILE LOOP THROUGH THE QUEUE
        while len(H.ArrayOfNodes) != 0:
            u = H.deleteMin()

            # FOR LOOP THROUGH EDGES OF NODE U
            for i in range(len(u.neighbors)):
                currentFriend = u.neighbors[i]
                if H.distances[currentFriend.dest.node_id] > H.distances[u.node_id] + currentFriend.length:
                    H.distances[currentFriend.dest.node_id] = H.distances[u.node_id] + currentFriend.length
                    H.previousNodes[currentFriend.dest.node_id] = u.node_id
                    H.decreaseKey(currentFriend.dest.node_id)
```

Array Implementation:


```
class ArrayQueue:
    ArrayOfNodes = []
    distances = []
    previousNodes = []

    def insert(self, node):
        self.ArrayOfNodes.append(node)

    def deleteMin(self):

        min = 10000
        minIndex = 0
        minNode = self.ArrayOfNodes[0]

        for i in range(len(self.ArrayOfNodes)):
            if self.distances[self.ArrayOfNodes[i].node_id] < min:
                min = self.distances[self.ArrayOfNodes[i].node_id]
                minNode = self.ArrayOfNodes[i]
                minIndex = i

        self.ArrayOfNodes.pop(minIndex)

        return minNode

    def decreaseKey(self, v):
        pass
```

Insert:

O(1) run time because all that we're doing is appending the node to the end of an array.

Decrease Key:

O(1) run time because we're not doing anything here

Delete Min:

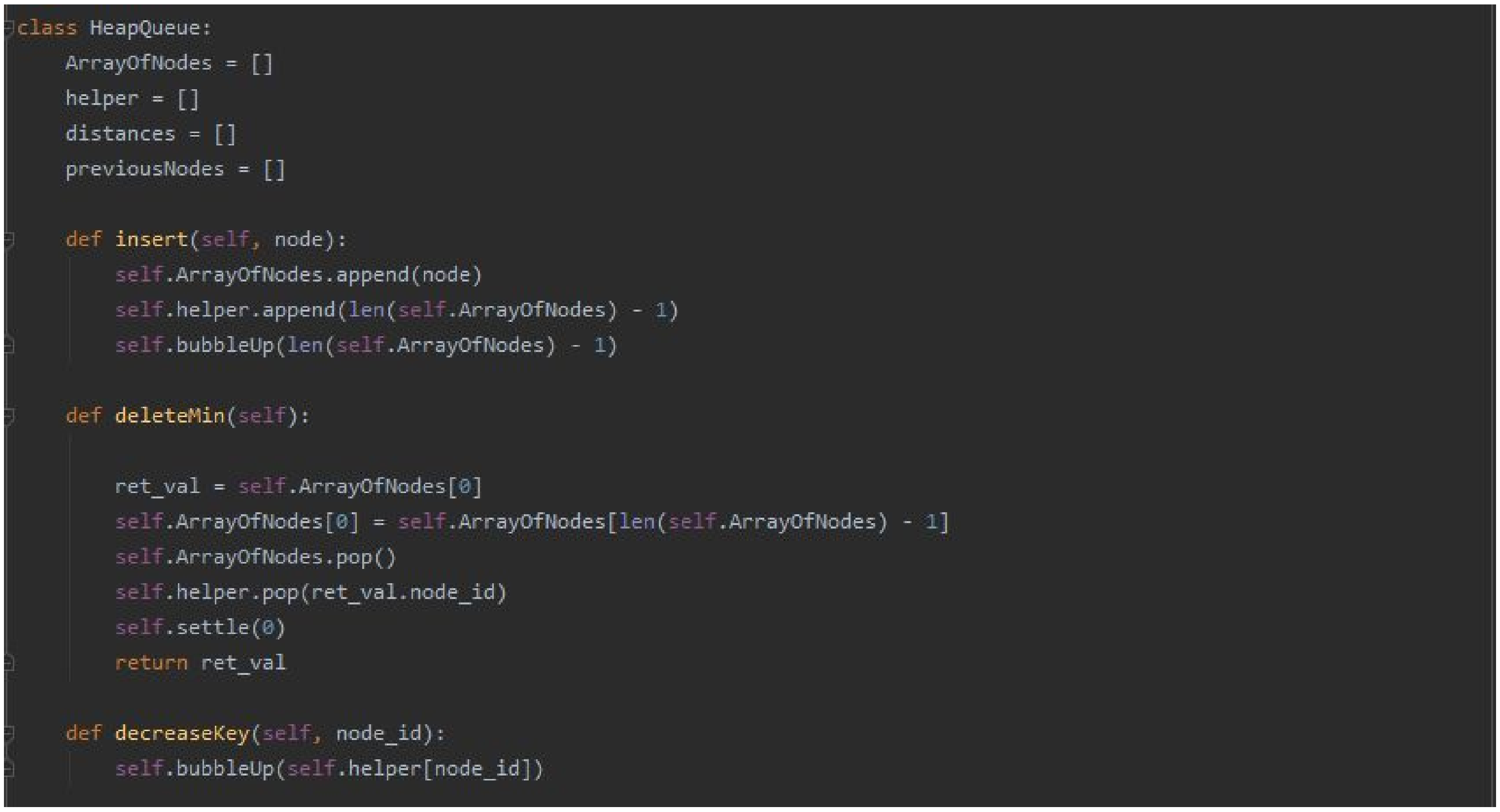
O(V) run time because we require a linear time scan as we go through the list in order to find the minimum value in the priority queue.

Overall Time and Space Complexity:

Time: O(E) for Dijkstra's while loop. O(1) from Inserting, O(1) from Decrease Key, and O(V) from Delete Min. Overall time complexity is O(E*V).

Space: O(V) from storing all nodes in an array, O(V) from storing all distances in an array, O(V) worst case for storing previous nodes. Overall space complexity is O(V)

Heap Implementation:



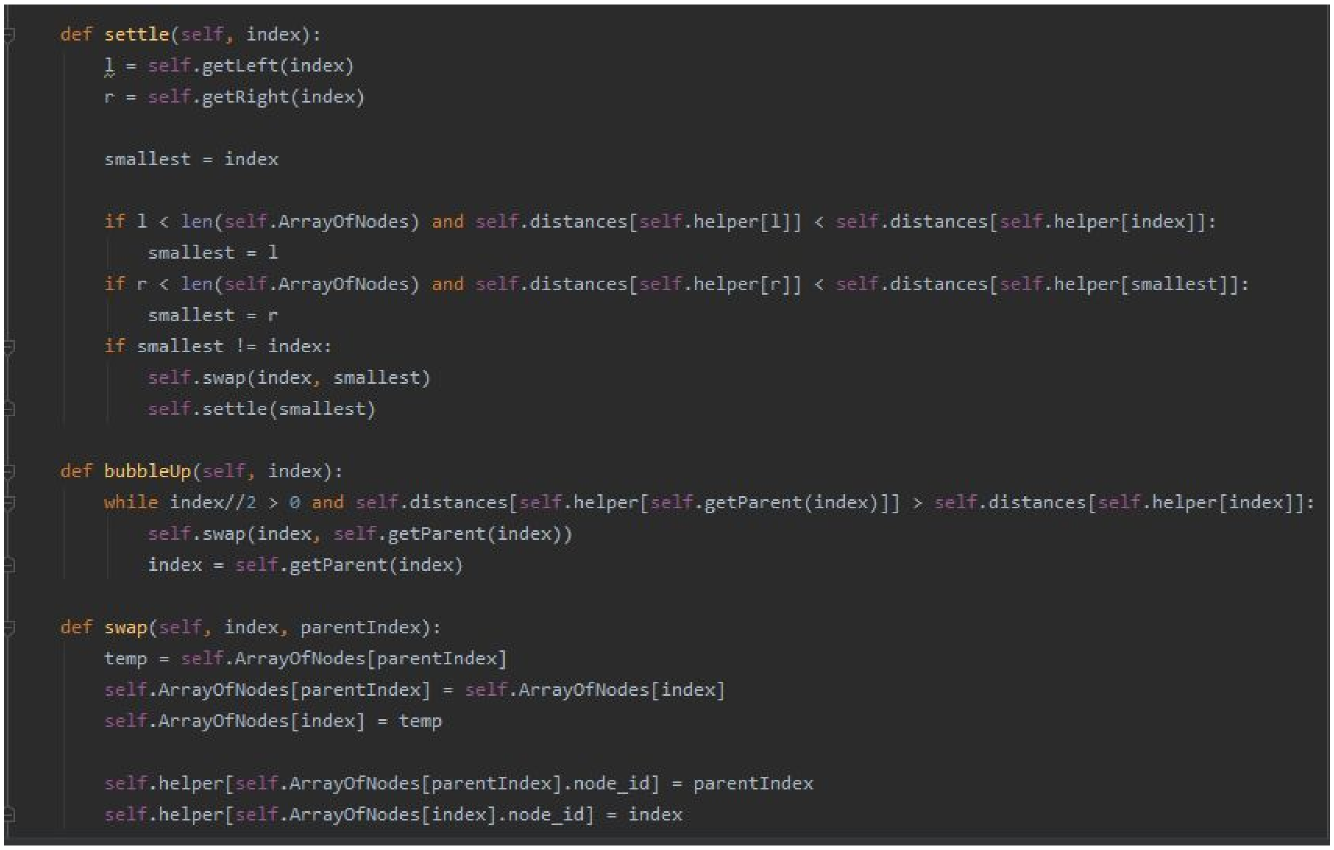
```
class HeapQueue:
    ArrayOfNodes = []
    helper = []
    distances = []
    previousNodes = []

    def insert(self, node):
        self.ArrayOfNodes.append(node)
        self.helper.append(len(self.ArrayOfNodes) - 1)
        self.bubbleUp(len(self.ArrayOfNodes) - 1)

    def deleteMin(self):

        ret_val = self.ArrayOfNodes[0]
        self.ArrayOfNodes[0] = self.ArrayOfNodes[len(self.ArrayOfNodes) - 1]
        self.ArrayOfNodes.pop()
        self.helper.pop(ret_val.node_id)
        self.settle(0)
        return ret_val

    def decreaseKey(self, node_id):
        self.bubbleUp(self.helper[node_id])
```



```
    def settle(self, index):
        l = self.getLeft(index)
        r = self.getRight(index)

        smallest = index

        if l < len(self.ArrayOfNodes) and self.distances[self.helper[l]] < self.distances[self.helper[index]]:
            smallest = l
        if r < len(self.ArrayOfNodes) and self.distances[self.helper[r]] < self.distances[self.helper[smallest]]:
            smallest = r
        if smallest != index:
            self.swap(index, smallest)
            self.settle(smallest)

    def bubbleUp(self, index):
        while index//2 > 0 and self.distances[self.helper[self.getParent(index)]] > self.distances[self.helper[index]]:
            self.swap(index, self.getParent(index))
            index = self.getParent(index)

    def swap(self, index, parentIndex):
        temp = self.ArrayOfNodes[parentIndex]
        self.ArrayOfNodes[parentIndex] = self.ArrayOfNodes[index]
        self.ArrayOfNodes[index] = temp

        self.helper[self.ArrayOfNodes[parentIndex].node_id] = parentIndex
        self.helper[self.ArrayOfNodes[index].node_id] = index
```

Insert:

O(log v) run time because the most number of swaps we will do during bubble up’s swap function is the height of the tree.

Decrease Key:

O(log v) run time because the most number of swaps we will do during bubble up’s swap is also the height of the tree.

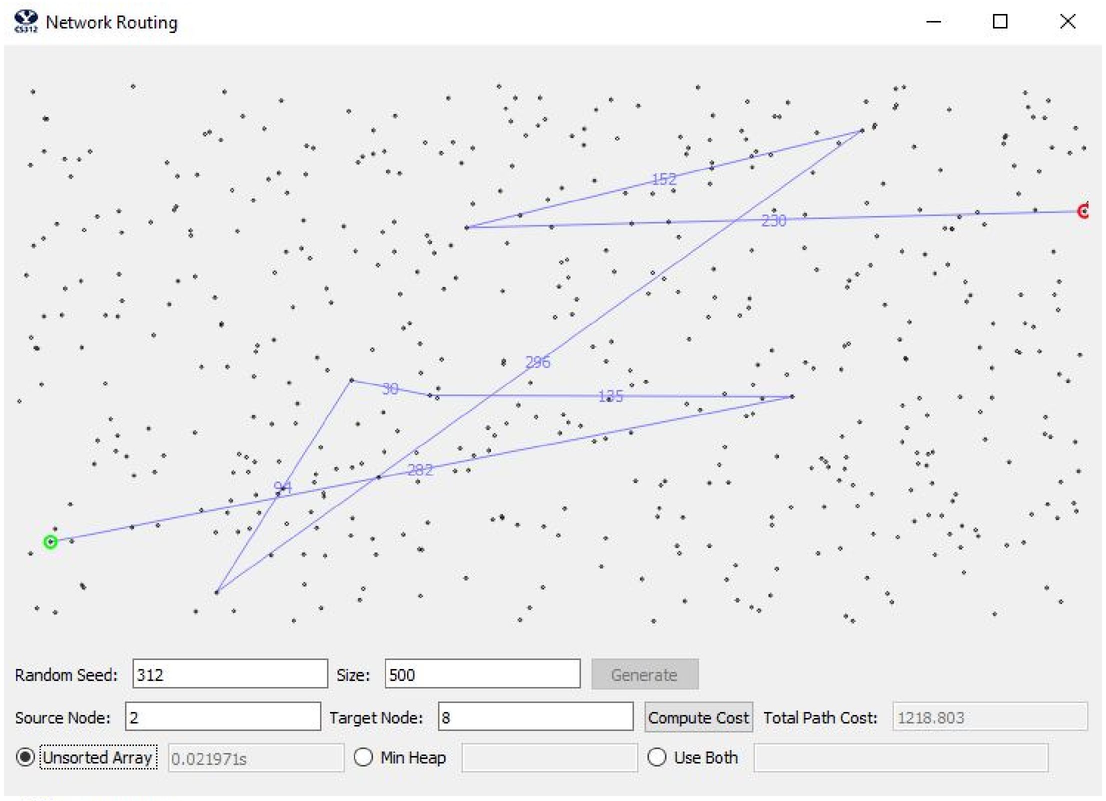
Delete Min:

O(log v) run time because we return the root value and then sift down the tree. The most number of swaps we do during sift down is also the height of the tree.

Overall Time and Space Complexity:

Time: O(E) for Dijkstra’s while loop, O(log v) from Inserting, O(log v) from Decrease Key, and O(log v) from Delete Min. Overall time complexity is O(E log V).

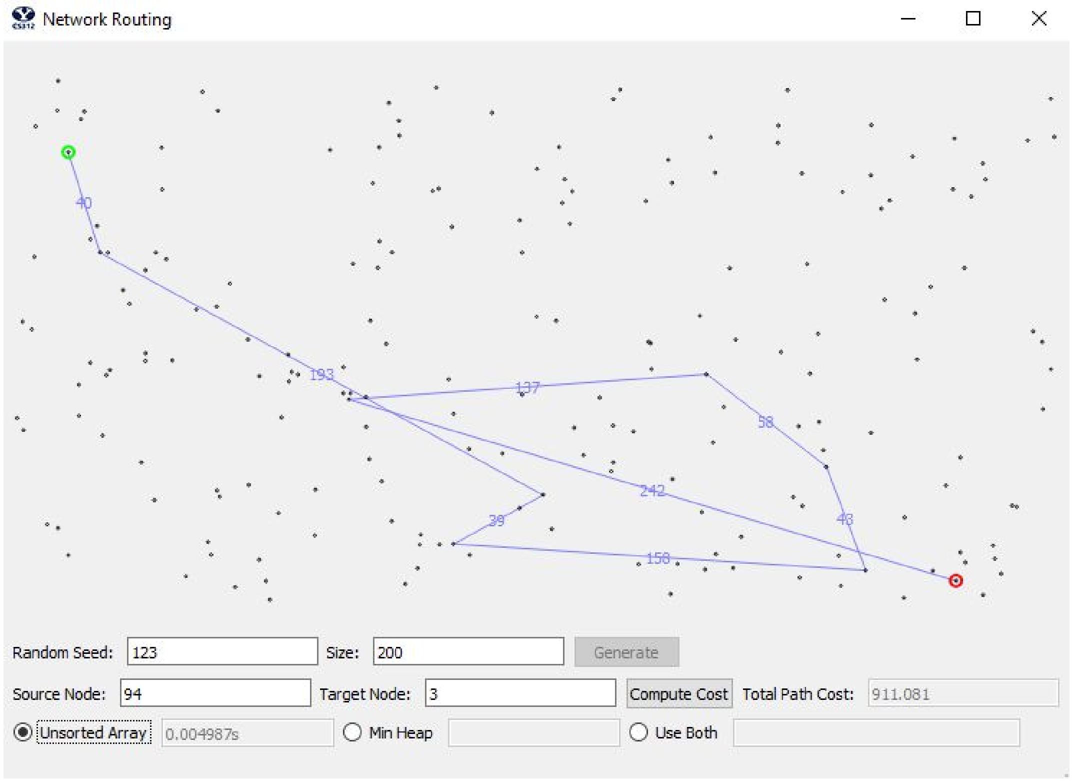
Space: O(V) from storing all nodes in an array, O(V) from storing all distances in an array, O(V) worst case for storing previous nodes, and O(V) for the helper array. Overall space complexity is O(V)



Random Seed: 312 Size: 500 Generate

Source Node: 2 Target Node: 8 Compute Cost Total Path Cost: 1218.803

(●) Unsorted Array 0.021971s ( ) Min Heap ( ) Use Both



Random Seed: 123 Size: 200 Generate

Source Node: 94 Target Node: 3 Compute Cost Total Path Cost: 911.081

(●) Unsorted Array 0.004987s ( ) Min Heap ( ) Use Both