```python
    def getShortestPath( self, destIndex ):
        self.dest = destIndex
        path_edges = []
        total_length = 0
        src_node = self.network.nodes[self.source]
        dest_node = self.network.nodes[self.dest]
        prev_node = dest_node
        while prev_node.prev is not None:
            edge = prev_node.prev
            path_edges.append( (edge.src.loc, edge.dest.loc, '{:.0f}'.format(edge.length)) )
            total_length += edge.length
            prev_node = edge.src
        if prev_node != src_node:
            return {'cost':float("inf"), 'path':[]}

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

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

        if (use_heap):
            self.heap_dijkstra(srcIndex)
        else:
            self.array_dijkstra(srcIndex)

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

    def heap_dijkstra(self, srcIndex):
        node_indices = {}
        queue = []
        self.network.nodes[srcIndex].dist = 0
        self.make_heap_queue(queue, node_indices)
        while len(queue) > 0:
            node = self.heap_delete_min(queue, node_indices)
            for edge in node.neighbors:
                if edge.dest.dist > edge.src.dist + edge.length:
                    edge.dest.dist = edge.src.dist + edge.length
                    edge.dest.prev = edge
                    self.decreaseKey(queue, node_indices, edge.dest)

```

```python
def array_dijkstra(self, srcIndex):
    queue = self.make_array_queue()
    queue[srcIndex].dist = 0
    while len(queue) > 0:
        node = self.array_delete_min(queue)
        for edge in node.neighbors:
            if edge.dest.dist > edge.src.dist + edge.length:
                edge.dest.dist = edge.src.dist + edge.length
                edge.dest.prev = edge


def make_heap_queue(self, queue, node_indices):
    for node in self.network.nodes:
        queue.append(node)
        self.bubble_up(queue, len(queue) - 1, node_indices)

def make_array_queue(self):
    return self.network.nodes.copy()

def heap_delete_min(self, queue, node_indices):
    if len(queue) > 1:
        min_node = queue[0]
        queue[0] = queue.pop()
    else:
        return queue.pop()
    self.sift_down(queue, 0, node_indices)
    return min_node

def array_delete_min(self, queue):
    min_node = None
    min_index = -1
    for i in range(len(queue)):
        if min_node is None or min_node.dist > queue[i].dist:
            min_node = queue[i]
            min_index = i
    queue.pop(min_index)
    return min_node

def decreaseKey(self, queue, node_indices, node):
    self.bubble_up(queue, node_indices[node], node_indices)
```

```python
def bubble_up(self, queue, index, node_indices):
    parent_index = (index - 1) // 2
    node_indices[queue[index]] = index
    while index != 0 and queue[index].dist < queue[parent_index].dist:
        node_indices[queue[index]], node_indices[queue[parent_index]] = (parent_index, index)
        queue[index], queue[parent_index] = (queue[parent_index], queue[index])
        index = parent_index
        parent_index = (index - 1) // 2

def sift_down(self, queue, index, node_indices):
    min_child_index = self.min_child(queue, index)
    node_indices[queue[index]] = index
    while min_child_index != 0 and queue[index].dist > queue[min_child_index].dist:
        node_indices[queue[index]], node_indices[queue[min_child_index]] = (min_child_index, index)
        queue[index], queue[min_child_index] = (queue[min_child_index], queue[index])
        index = min_child_index
        min_child_index = self.min_child(queue, index)

def min_child(self, queue, i):
    if (i + 1) * 2 > len(queue):
        return 0
    elif (i + 1) * 2 == len(queue):
        return ((i + 1) * 2) - 1
    else:
        a = ((i + 1) * 2) - 1
        b = (i + 1) * 2
        return a if queue[a].dist < queue[b].dist else b
```

For a heap, insertion is O(log(v)), since it is simply appending to the queue(which is O(1)) and then bubbling up, which is log(v) time.

For the unsorted array, insertion is O(1), as it would simply append to the queue; in this case I just returned a copy of the array in make_array, which is O(v), but there would be v insertions anyway, so it is the same time complexity.

Bubble_up is log(v) because it will at worst iterate a number of times equal to the height of the tree.

Heap_delete_min is log(v) because pop() is an O(1) operation and sift_down is log(v) (for the same reason bubble_up is log(v))

Array_delete_min is O(n) because it has to iterate though the list to find the correct index, then pop(index) is O(n)

The array implementation does not have a decreaseKey function, as it would be pointless. Since the array is unsorted, there is no need to notify anything but the node itself when a distance is updated.

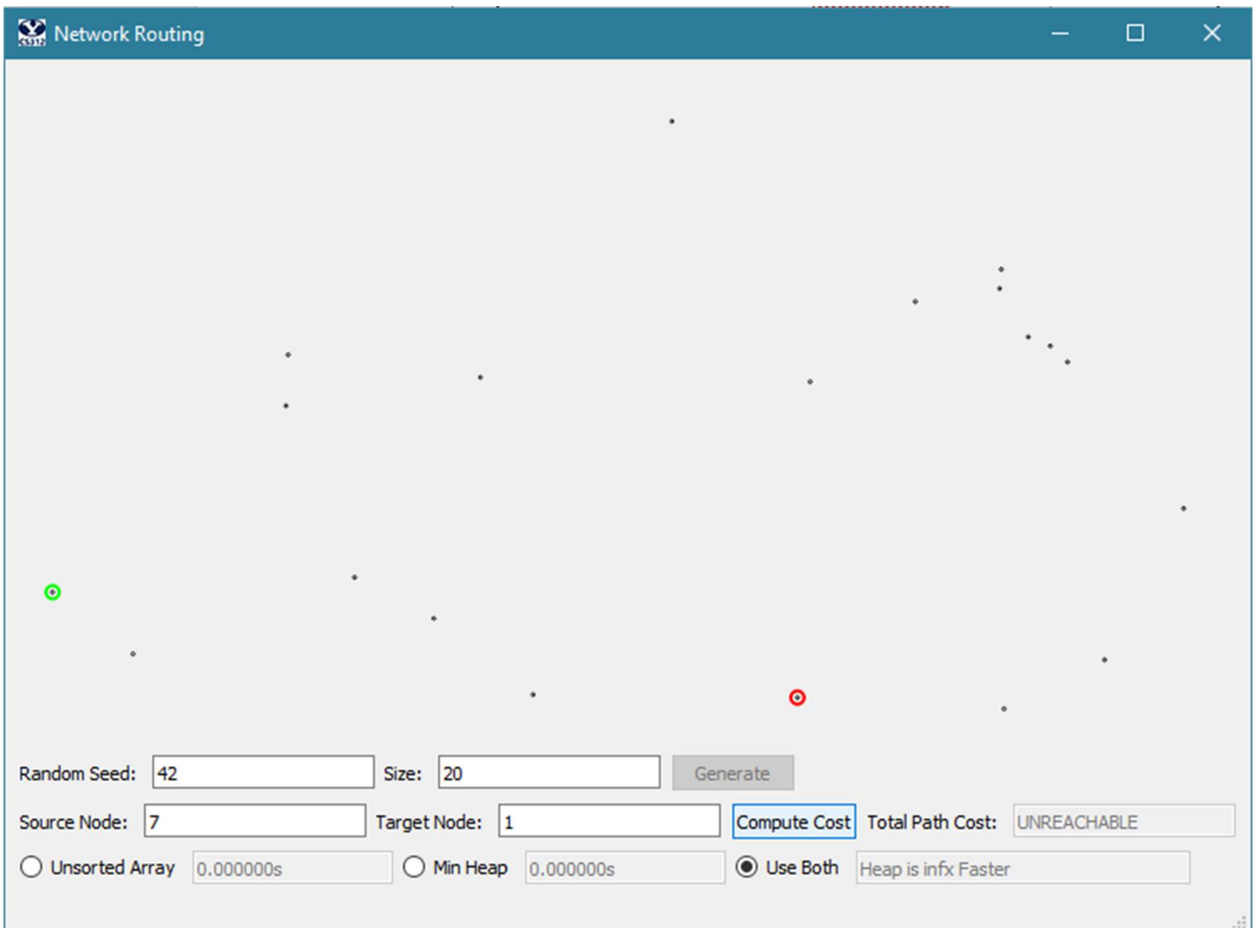
Decrease_key is also log(v) since all it does is call bubble_up; the node value in the queue has already been updated on the line previous to the call to decrease_key

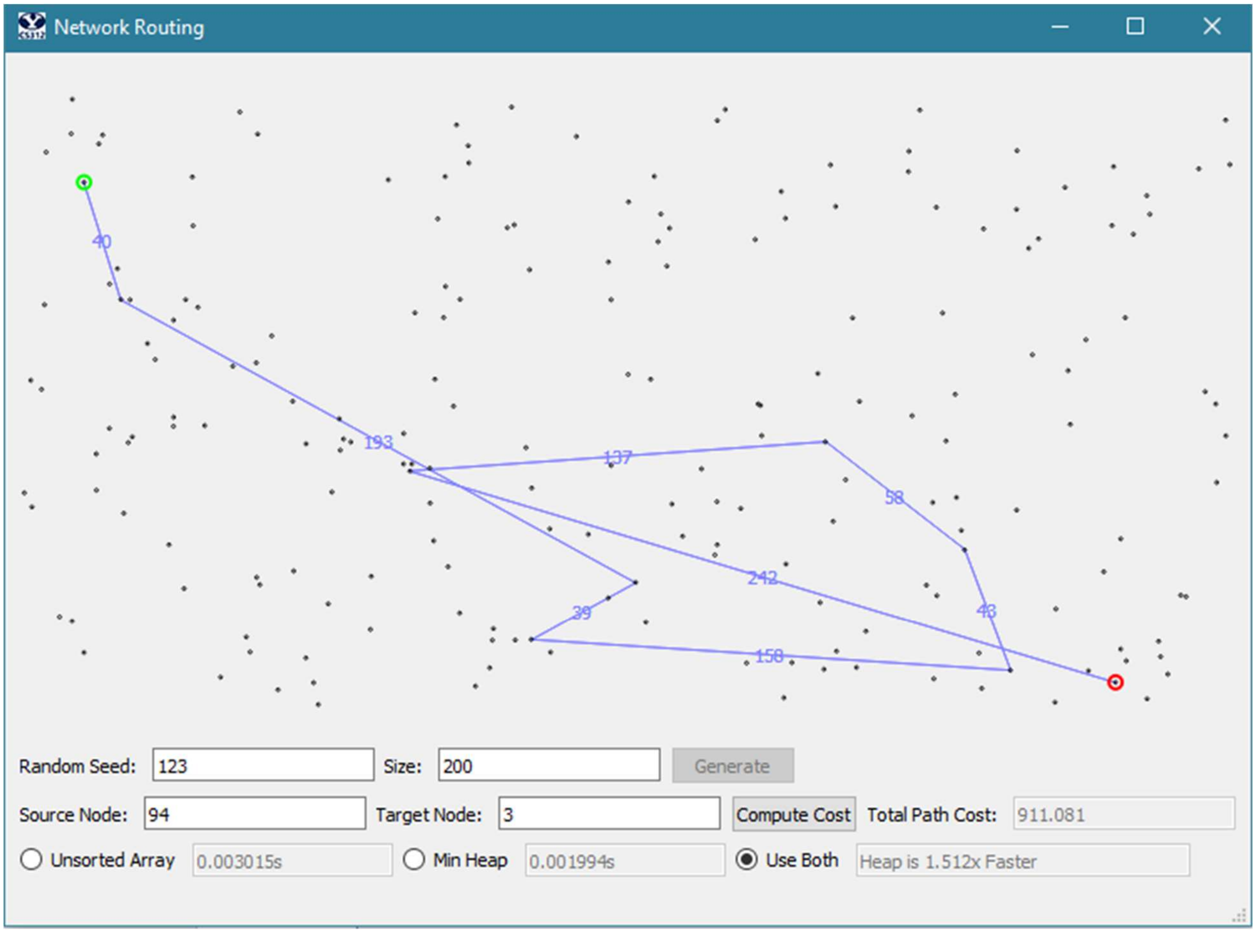
The space complexity of the unsorted array implementation is simply O(v), as it only ever keeps track of the single array

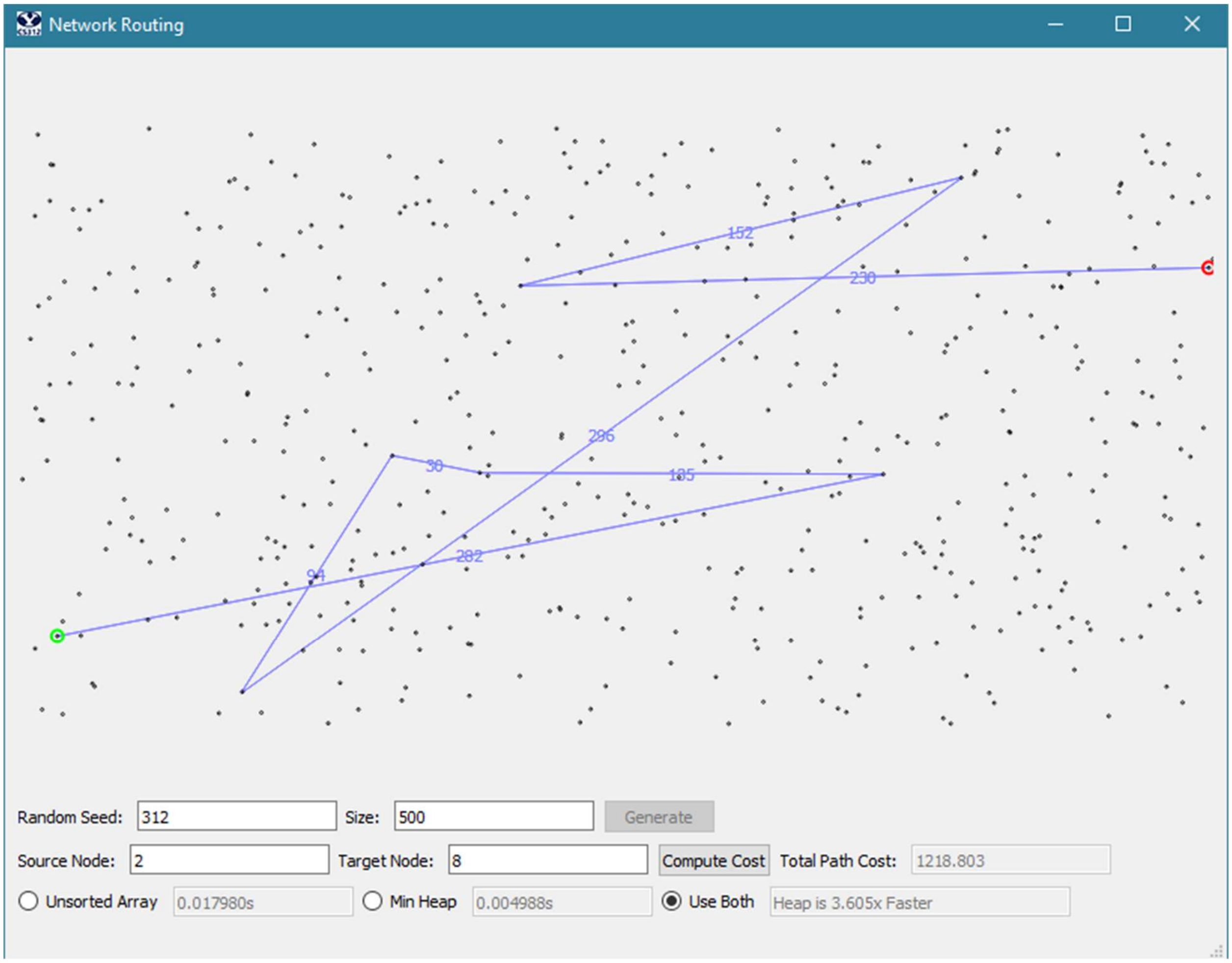
The space complexity of the heap implementation is O(2v) = O(v), as it keeps track of the array as well as a dictionary of node_indices which is size v.

The overall time complexity for the unsorted array implementation is O(v^2), as it will go through each node (O(v)) and for each node, call delete_min, which adds another factor of v.

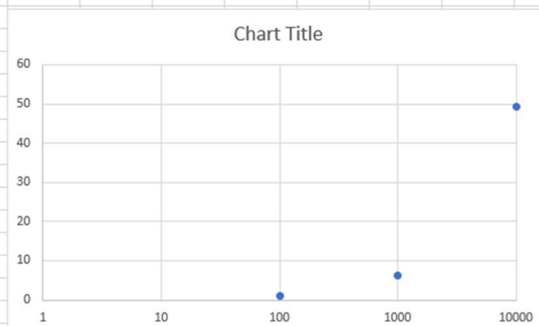
The overall time complexity for the heap implementation is O(vlog(v)), as it will go through each node (again, O(v)) and for each node, call each of insert, delete_min, and decrease_key, adding up to O(log(V) asymptotically, resulting in O(vlog(v)) overall.




Network Routing
40
193
137
58
242
39
43
158
Random Seed: 123
Size: 200
Generate
Source Node: 94
Target Node: 3
Compute Cost
Total Path Cost: 911.081
Unsorted Array 0.003015s
Min Heap 0.001994s
Use Both
Heap is 1.512x Faster



	A	B	F	G
1	100	0.999	100	0.9984
2	100	1.02	1000	6.2686
3	100	1.003	10000	49.2888
4	100	0.991		
5	100	0.979		
6	1000	6.258		
7	1000	6.27		
8	1000	6.363		
9	1000	6.181		
10	1000	6.271		
11	10000	59.405		
12	10000	46.246		
13	10000	48.63		
14	10000	45.48		
15	10000	46.683		
16	100000			
17	100000			
18	100000			
19	100000			
20	100000			
21	1000000			
22	1000000			
23	1000000			
24	1000000			
25	1000000			
26				



Unfortunately I ran out of time to compare array to heap implementation on the 100000 tests, but my heap seemed to get about another 10x faster the more nodes it had to go through, so I would estimate that my heap would be about 5000x faster than the array implementation for 1000000 nodes.