Advanced Algorithms (CS 5512)
Project #3
Submitted by: Shishir Khanal
Submitted to: Dr Paul Bodily

Problem Statement:

1. **Correctly implement Dijkstra's algorithm and the functionality discussed above. Include a copy of your (well-documented) code in your submission to the TA.**

```python
def dijkstra(self, Graph, source, PQ):
    self.Nodes = Graph.getNodes();
    self.prev = {}
    self.distance = {}
    self.pointdistance = {}

    for node in self.Nodes:
        self.distance[node.node_id] = sys.maxsize
        self.prev[node.node_id] = None
        print(node.node_id)

    self.distance[source] = 0
    PQ.decrease_key(source,0)

    while PQ.getsize() != 0:

        u = PQ.delete_min()
        for edge in u.neighbors:
            if self.distance[edge.dest.node_id] > self.distance[u.node_id] + edge.length:
                self.distance[edge.dest.node_id] = self.distance[u.node_id] + edge.length
                self.prev[edge.dest.node_id] = u.node_id
                PQ.decrease_key(edge.dest.node_id, self.distance[edge.dest.node_id])
                self.pointdistance[edge.dest.node_id] = edge.length
```

2. **Correctly implement both versions of a priority queue, one using an array with worst case O(1), O(1) and O(|*V*|) operations and one using a heap with worst case O(log|*V*|) operations. For each operation (*insert*, *delete-min*, and *decrease-key*) convince us (refer to your included code) that the complexity is what is required here.**

Array Implementation:

```
class priorityqueuearray:
    def __init__( self, graph_nodes):
        self.pq_array = [[node, sys.maxsize] for node in graph_nodes]
        self.size = len(graph_nodes)
        self.index = [0]*self.size
        self. insert(graph_nodes, self.index)

    def insert(self, graph_nodes, index):
        for ind, node in enumerate(graph_nodes):
            self.index[node.node_id] = ind

    def getsize(self,):
            return len(self.pq_array)

    def distanceweight(self, i):
        return self.pq_array[i][1]

    def swap(self, index1, index2):
        self.pq_array[index1], self.pq_array[index2] = self.pq_array[index2], self.pq_array[index1]
        temp1 = self.pq_array[index1][0].node_id
        temp2 = self.pq_array[index2][0].node_id
        self.index[temp1], self.index[temp2] = self.index[temp2], self.index[temp1]


    def decrease_key(self, ind, distance):
        temp = self.index[ind]
        if temp >= self.size:
            return "unity index"
        self.pq_array[temp][1] = distance

    def delete_min(self):
        if self.size ==0:
            return "no array"
        i = min(range(self.size), key = self.distanceweight)
        self.swap(i, self.size-1)
        minimum, pos = self.pq_array.pop()
        self.size -= 1
        return minimum
```

Binary Heap Implementation:

```python
class priorityqueuebinaryheap:
    def __init__(self, graph_nodes):
        self.pq_array = [[node, sys.maxsize] for node in graph_nodes]
        self.size = len(graph_nodes)
        self.index = [0]*self.size
        self.insert(graph_nodes, self.index)
        self.heapify()

    def getsize(self):
        return len(self.pq_array)

    def insert(self, graph_nodes, index):
        for ind, node in enumerate(graph_nodes):
            self.index[node.node_id] = ind

    def distanceweight(self, ind):
        return self.pq_array[ind][1]


    def getparentindex(self, ind):
        return (ind-1)//2

        #l_r =>  1 or 2 for left child or right child
    def getchildindex(self, ind, l_r):
        return 2*ind + l_r

    def isChild(self, ind):
        return 0 <= ind  and ind < self.size

    def isleafnode(self, ind):
        return self.isChild(ind) and not self.isChild(self.getchildindex(ind, 1))


    def heapify(self):
        for i in reversed(range(self.size)):
            if self.distanceweight(i) < self.distanceweight(self.getparentindex(i)):
                self.swap(self, i, self.getparentindex(i))

    def swap(self, index1, index2):
        self.pq_array[index1], self.pq_array[index2] = self.pq_array[index2], self.pq_array[index1]
        temp1 = self.pq_array[index1][0].node_id
        temp2 = self.pq_array[index2][0].node_id
        self.index[temp1], self.index[temp2] = self.index[temp2], self.index[temp1]

    def bubbleup(self, ind):
        while ind != 0 :
            if self.distanceweight(ind) >= self.distanceweight(self.getparentindex(ind)):
                break
            self.swap(ind, self.getparentindex(ind))
            ind = self.getparentindex(ind)
```

```python
def delete_min(self):
    if self.size == 0:
        return "Empty heap"
    self.swap(0, self.size-1)
    minimum, pos = self.pq_array.pop()
    self.size -= 1
    self.bubbledown(0)
    return minimum

def bubbledown(self, ind):

    while self.isChild(ind) and not self.isleafnode(ind):
        lowestindex = ind
        if self.distanceweight(self.getchildindex(ind, 1)) < self.distanceweight(lowestindex):
            lowestindex = self.getchildindex(ind, 1)


        if self.isChild(self.getchildindex(ind, 2)) and self.distanceweight(self.getchildindex(ind, 2)) < self.distanceweight(lowestindex):
            lowestindex = self.getchildindex(ind,2)
        if lowestindex == ind:
            return
        self.swap(ind, lowestindex)
        ind = lowestindex


def decrease_key(self, ind, distance):
    ii = self.index[ind]
    if ii >= self.size:
        return "unity index"
    self.pq_array[ii][1] = distance
    self.bubbleup(ii)
```

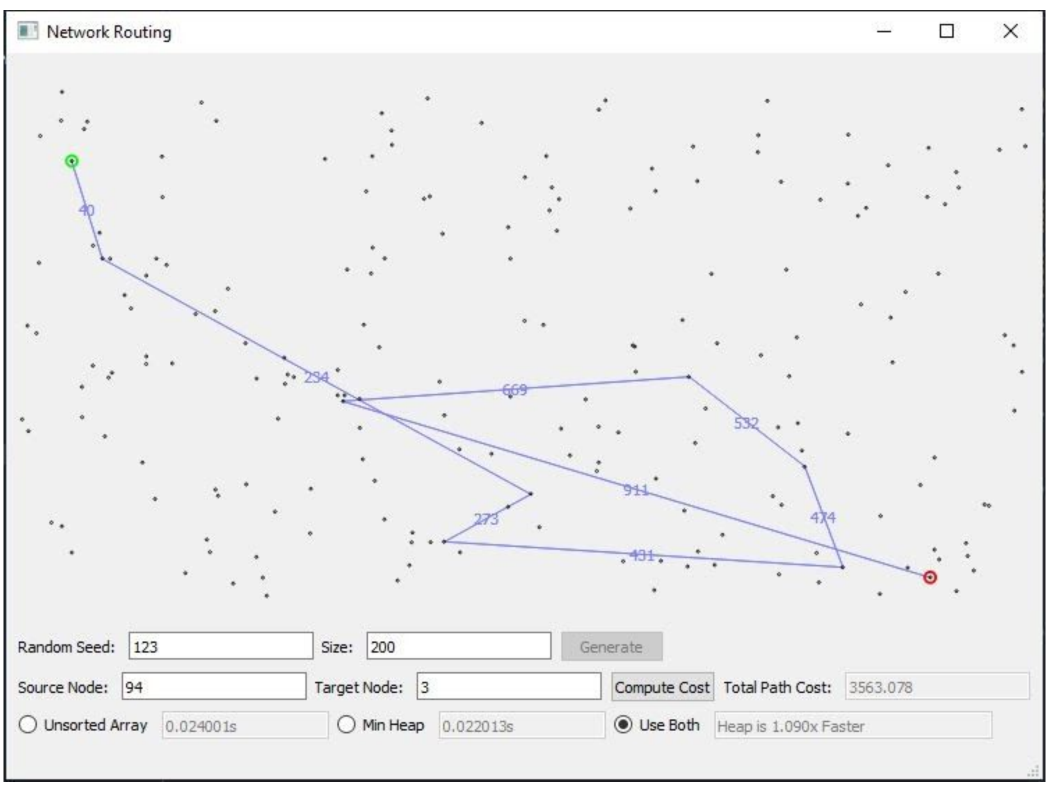
3. **Explain the time and space complexity of both implementations of the algorithm by showing and summing up the complexity of each subsection of your code.**

	Subsections:	**Time Complexity**	**Space Complexity**
dijkstra	**Using Array**	O(|V|^2)	O(|V|)
	Using Binary Heap	O(|V|log|V|)	O(|V|)
priorityqueuebinaryheap	**getsize**	O(V)	O(V)
	insert	O(log|V|)	O(V)
	distanceweight	O(1)	O(1)
	getparentindex	O(1)	O(1)
	getchildindex	O(1)	O(1)
	isChild	O(1)	O(1)

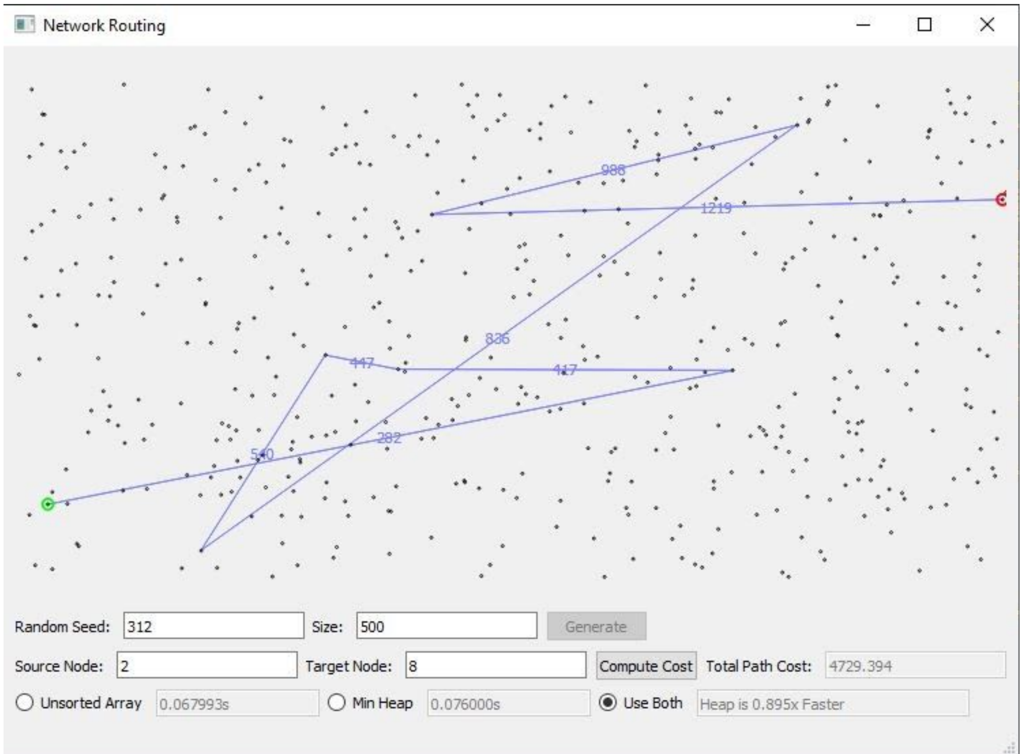
	isleafnode	O(1)	O(1)
	heapify	O(V)	O(V)
	swap	O(n)	O(1)
	bubbleup	O(1)	O(1)
	delete_min	O(log|V|)	O(1)
	bubbledown	O(1)	O(1)
	decrease_key	O(log|V|)	O(1)
priorityqueuearray	**insert**	O(1)	O(V)
	getsize	O(V)	O(V)
	distanceweight	O(1)	O(1)
	swap	O(V)	O(1)
	decrease_key	O(1)	O(1)
	delete_min	O(V)	O(V)

4. **For Random seed 42 - Size 20, Random Seed 123 - Size 200 and Random Seed 312 - Size 500, submit a screenshot showing the shortest path (if one exists) for each of the three source-destination pairs, as shown in the images below.**
    1. **For Random seed 42 - Size 20, use node 7 (the left-most node) as the source and node 1 (on the bottom toward the right) as the destination, as in the first image below.**

2. **For Random seed 123 - Size 200, use node 94 (near the upper left) as the source and node 3 (near the lower right) as the destination, as in the second image below.**



3. **For Random seed 312 - Size 500, use node 2 (near the lower left) as the source and node 8 (near the upper right) as the destination, as in the third image below.**

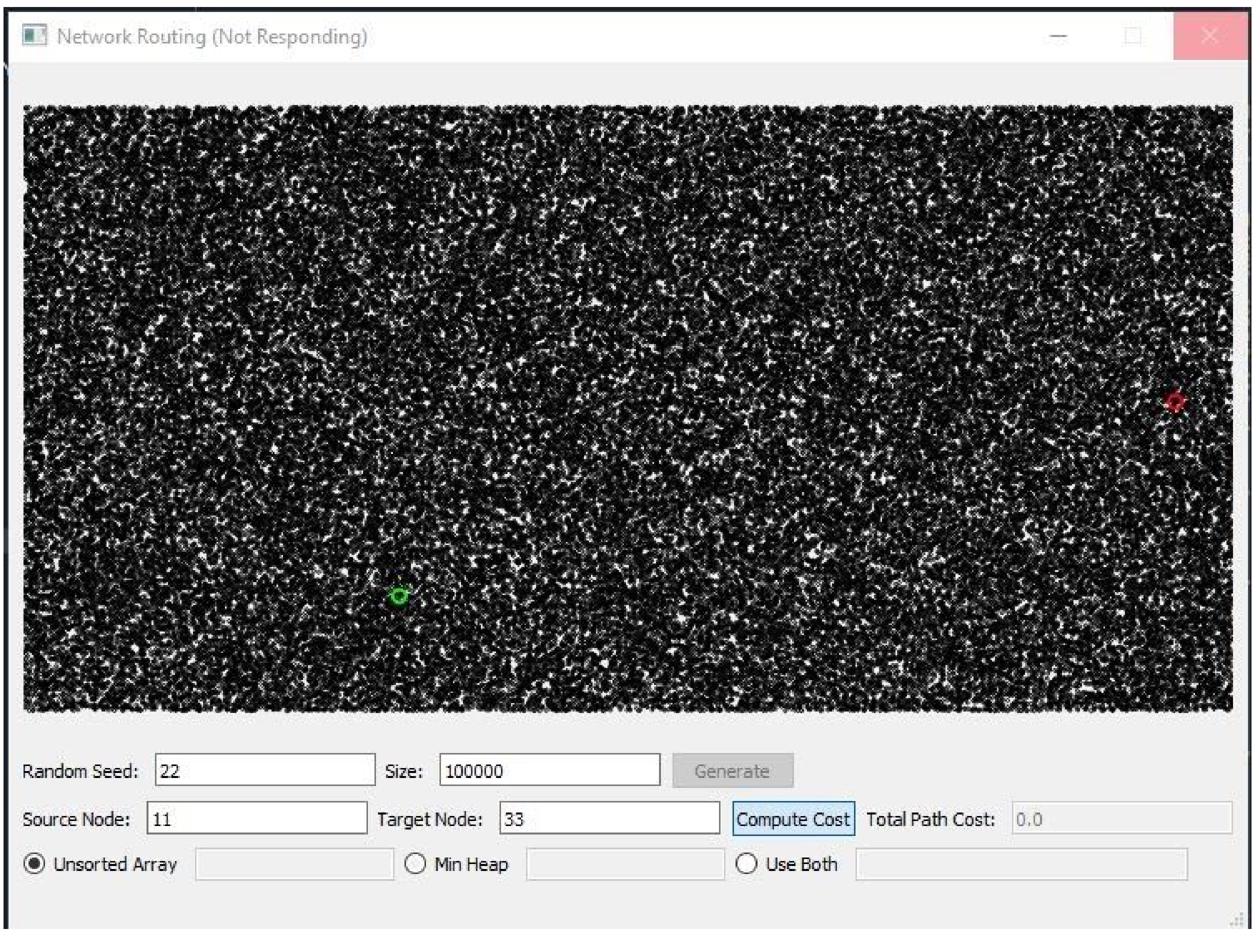


5. **For different numbers of nodes (100, 1000, 10000, 100000, 1000000), compare the empirical time complexity for Array vs. Heap, and give your best estimate of the difference (for 1000000 nodes, run only the heap version and then estimate how long you might expect your array version to run based on your other results). For each number of nodes do at least 5 tests with different random seeds, and average the results. Redo any case where the destination is unreachable. Each time, start with nodes approximately in opposite corners of the network. Graph your results and also give a table of your raw data (data for each of the runs); in both graph and table, include your one estimated runtime (array implementation for 1000000 points). Discuss the results and give your best explanations of why they turned out as they did.**

Seed: 22
Source Node: 11
Target Node: 33

n	Array(s)					Heap(s)				
100	0.011997	0.015999	0.014986	0.008551	0.015999	0.011934	0.014998	0.015999	0.016998	0.016000
1000	0.210003	0.200998	0.219004	0.213586	0.202532	0.154001	0.144997	0.148001	0.149002	0.142000
100000	UI Becomes Unresponsive					15.937809	15.378215	15.583414	14.810755	15.355181
1000000	UI Becomes Unresponsive					UI Becomes Unresponsive				

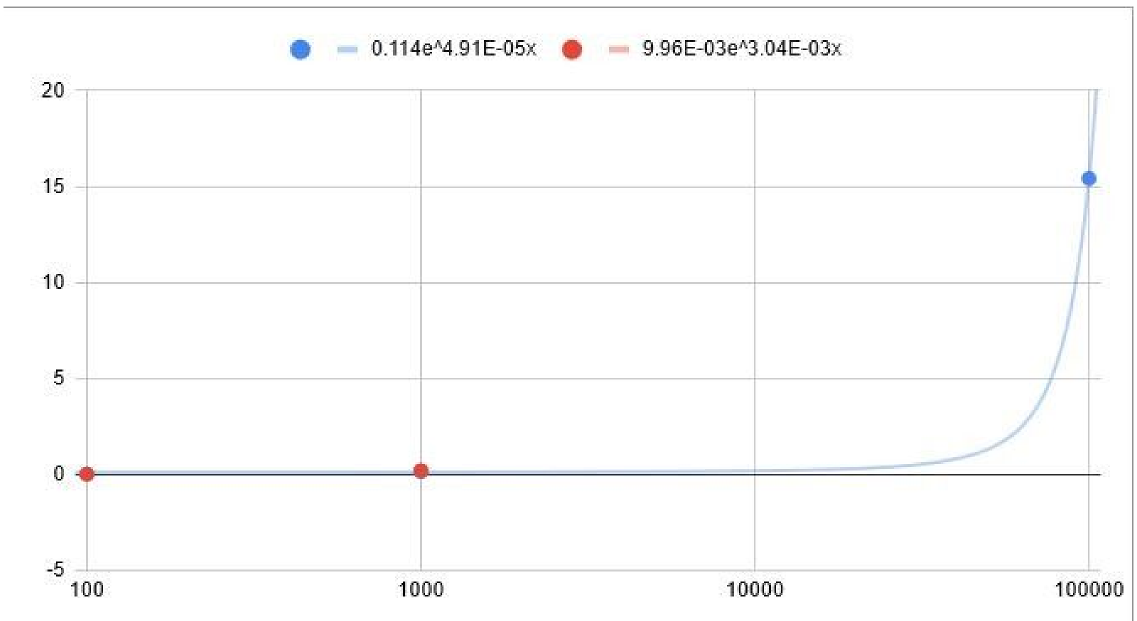
N = 100000:



Average Values

n	Array	Heap
100	0.0135064	0.0151858
1000	0.2092246	0.1476002
100000	-	15.4130748
1000000	-	-

Curve Fit (Log Plot of y-axis):



For Array: $time = (9.96 * 10^{-3})e^{(3.04*10^{-3})*nodes}$

For Heap: $time = (0.114 * 10^{-3})e^{(4.91*10^{-5})*nodes}$

Average Values (Completed with estimation)

n	Array	Heap
100	0.0135064	0.0151858
1000	0.2092246	0.1476002
100000	1.06E130	15.4130748
1000000	inf	2.40E17