# Network Routing Report

## 1. Correct Implementation of Dijkstra's. See appendix for full code

## 2. Correct Implementation of Priority Queues. See appendix for code

1. Binary Heap Operations
    a. **Insert O(1)**
       Simple python set.add call. Is O(1) time
       See code line 14
    b. **deleteMin O(V)**
       Loops over each element in the set O(V) and saves the location of the minimum element according to the distance values which are passed in. This element is then removed from the set using pop O(1) and returned. Since each element in the set is visited, it is O(V) time.
       See code line 18-25
    c. **decreaseKey O(1)**
       Passes as distance values are not stored internally. Thus is O(1) time
       See code line 29
2. Unsorted Array Operations
    a. **Insert O(1) / O(logV)**
       Inserts the new node index with inf distance into the tree at the first empty spot. Tree is implemented as an array so adding to it is append with O(1) time. Bubbling up is not necessary here as inserts are all called on nodes with infinite distance so they will already be in the correct position. As it is a simple append, this is O(1). It would be worst case O(logV) if bubbling up was necessary.
       See code lines 44-50
    b. **deleteMin O(logV)**
       Removes the first element in the tree which is guaranteed to be the min and returns it. This is O(1) time. The last element is then moved into the first position. The pointer for the last element is updated and bubble down is called on the first index to correct the ordering of the tree. Bubble down could potentially move the new top node all the way back down to the bottom which is a worse case of O(logV) as there are logV levels in the tree. This means the overall worst case for deleteMin is O(logV).
       See code lines 52-68 and 98-134 (bubble down)
    c. **decreaseKey O(logV)**
       The location of the given node is found in the pointers array O(1) and used to change the value of the node in the tree at that location to the new value passed in. Bubble up is then called on that node (as the value of the Node will only be less than it was before, never more). Bubbling up is a worst case of O(logV) time because it could in the worst case move up every layer in the tree which is logV layers. This is

done by comparing the value of the node with the value of it's parent and swapping if it is less. This makes the overall worst case of decreaseKey O(logV) time.
See code lines 71-77 and 80-95 (bubble up)

## 3. Time and Space of both implementations

**Pseudo Code for Dijkstra's algorithm:**

| | |
|---|---|
| Create priority queue | O(1) |
| For each node in network: | O(V) |
|     Set distance to node to infinity | |
|     Set previous node to null | |
|     Insert node into queue | |
| decreaseKey for start node to 0 | set O(1) / heap O(logV) |
| while queue is not empty: | O(V) |
|     min = deleteMin from queue | set O(V) / heap O(logV) |
|     if distance to min == inf: break | |
|     for edge (u,v) from min: | O(3) because 3 edges from each vertex |
|         if distance to v is less: | |
|             decreaseKey for v | set O(1) / heap O(logV) |

**Time Complexity:**
From this pseudo code we can see that regardless of the priority queue implementation, the time complexity of Dijkstra's algorithm is at least O(V) because it loops over each node twice. The number of edges in the graphs we were using was fixed at 3 edges per vertex so this constant factor is ignored. In different graphs, this value might be included as a time complexity of O(V + E).

Using the unsorted array (set) implementation, insert and decrease key are both O(1) and so don't contribute to the overall time complexity. The deleteMin operation is, however, O(V) time so when using an unsorted array implementation of the priority queue, the overall algorithm is O(V^2) time complexity.

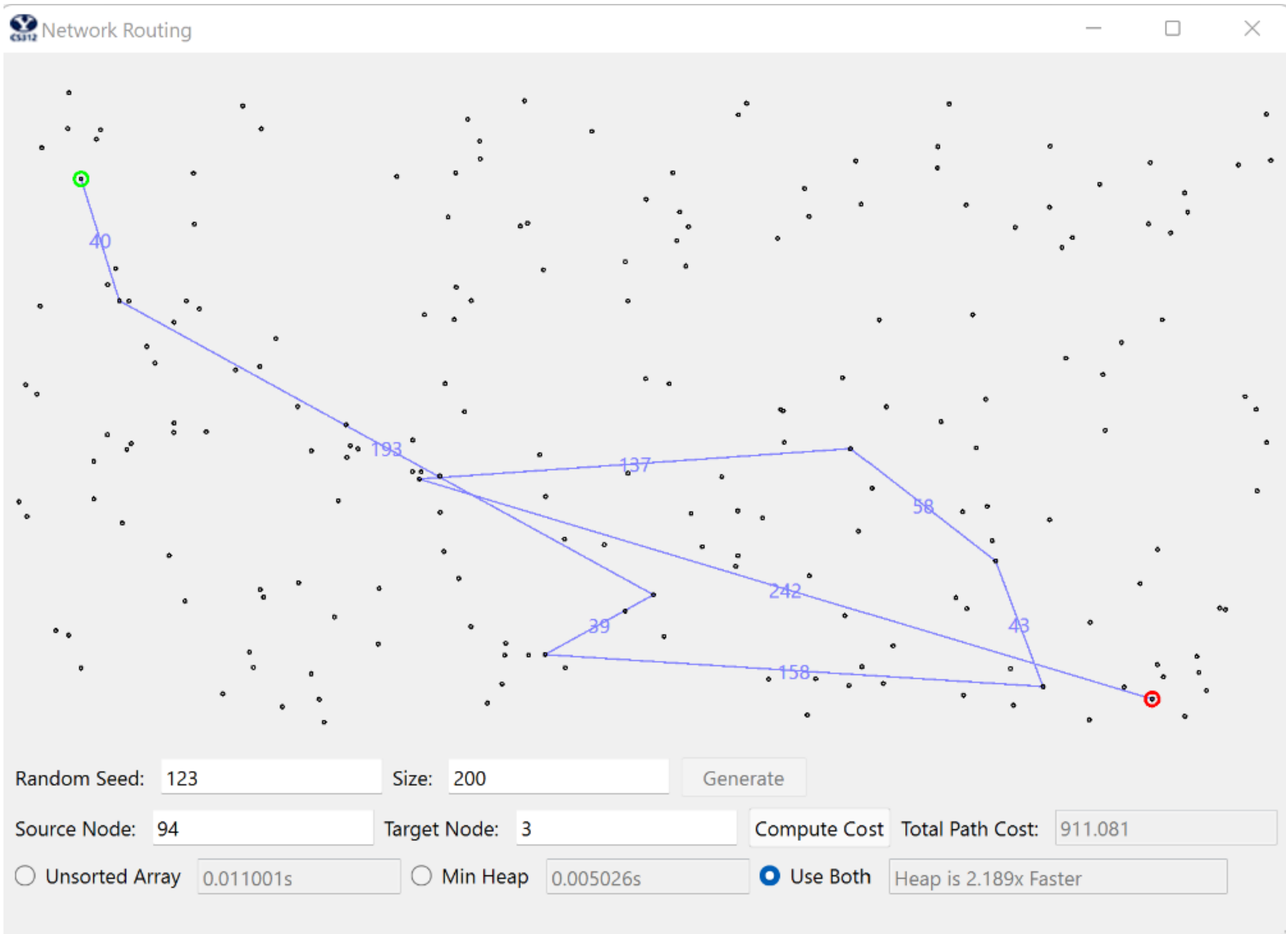
Using the binary heap implementation, insert, decrease, and deleteMin operations all have a worst case of O(logV). While this is slower than the O(1) time for insert and decrease, it is faster for deleteMin. The time complexity after dropping constant factors is dominated by the V calls to delete min giving a overall time complexity of O(VlogV) or O((V+E)logV) if the graph did not have a fixed number of edges per vertex.

**Space Complexity:**
Space complexity for the two algorithms is the same despite implementation of the priority queue. Each node is stored just once along with it's distance value, previous pointer, and outgoing edges. This complete representation is of size V or V+E when edge number is not fixed. Since the algorithm does not create an additional memory, the space complexity is O(V).

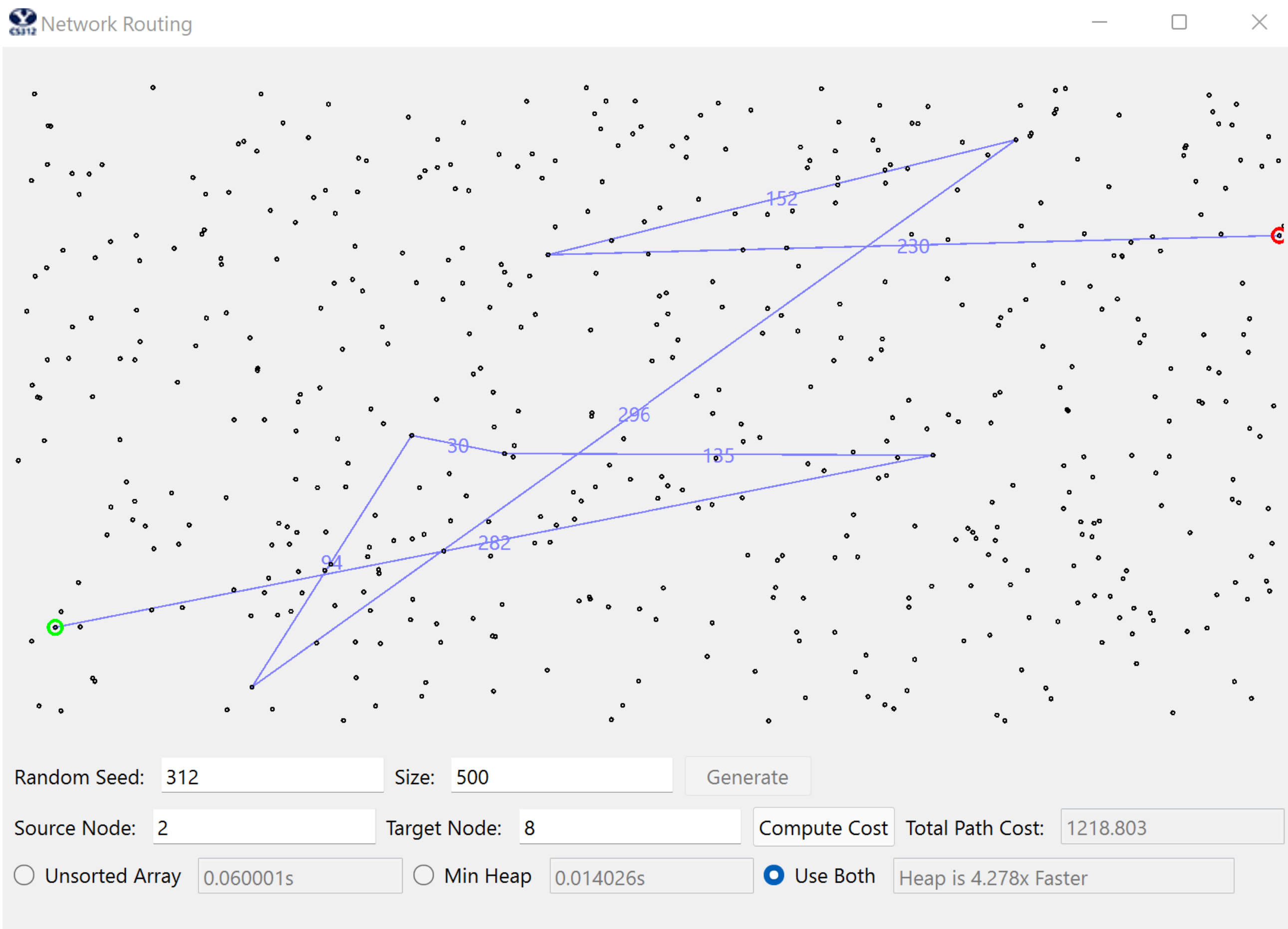
# 4. Example Screenshots

a) Seed: 42, Size 20

b) Seed: 123, Size 200



c) Seed: 312, Size: 500



## 5. Empirical Analysis

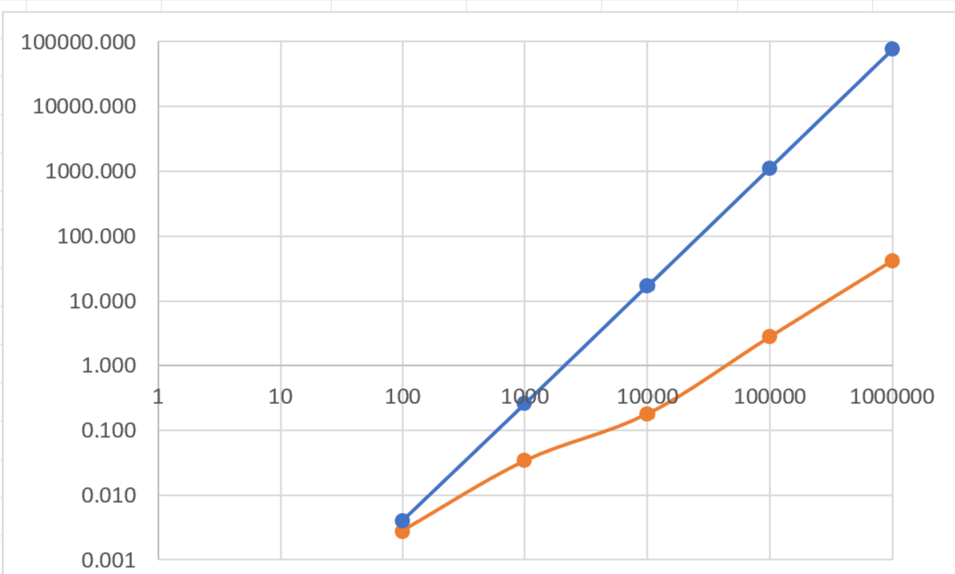
| Array | | | | | | |
|---|---|---|---|---|---|---|
| # points | test 1 | test 2 | test 3 | test 4 | test 5 | avg time |
| 100 | 0.005 | 0.004 | 0.00301 | 0.004 | 0.004 | 0.00400 |
| 1000 | 0.238 | 0.254 | 0.23699 | 0.229 | 0.314 | 0.25440 |
| 10000 | 15.8 | 16.57 | 18.095 | 17.28 | 16.45 | 16.83900 |
| 100000 | 1125 | N/A | N/A | N/A | N/A | 1125.00000 |
| 1000000 | 75375 | N/A | N/A | N/A | N/A | 75375.00000 |

| Heap | | | | | | |
|---|---|---|---|---|---|---|
| # points | test 1 | test 2 | test 3 | test 4 | test 5 | avg time |
| 100 | 0.00302 | 0.002 | 0.003 | 0.003 | 0.003 | 0.00280 |
| 1000 | 0.03498 | 0.036014 | 0.031003 | 0.03421 | 0.03245 | 0.03373 |
| 10000 | 0.18 | 0.19 | 0.17 | 0.17543 | 0.18231 | 0.17955 |
| 100000 | 2.768 | 2.804 | 2.912 | 2.824 | 2.745 | 2.81060 |
| 1000000 | 42.15 | N/A | N/A | N/A | N/A | 42.15000 |

This shows the number of times faster the Heap is at that level than the unsorted array implementation

| Factor Difference |
|---|
| |
| 1.427247 |
| 7.541875 |
| 93.78551 |
| 400.2704 |
| 1788.256 |

Here is a graph of the results for both implementations with the heap in orange.



I was unable to run the algorithm for 1000000 points because my laptop could not handle even generating that number of points. I estimated values for the unsorted array at 100,000 points and 1,000,000 points based on the observed rate of increase in the previous values (about 67x). In the heap, times appeared to be increasing by about 15x each time so I used this estimate for the 1,000,000 points as I couldn't run it.

As can be seen, the times for both implementations are clearly linear in the logarithmic scale meaning they are showing exponential growth as we expected. The heap implementation is significantly faster especially at large values where the nlogn term is significantly smaller than the n^2 term. At smaller values, the speeds are more similar as the overhead of running the algorithm and the constant factors have a larger impact.

These results show obvious n^2 and nlogn growth in runtimes which is what we expected from the theoretical analysis. The constant factors of course have an impact but especially with the larger values, the trend is more obvious.

## Appendix: Full Code

```
#!/usr/bin/python3


from CS312Graph import *
import time

# Unsorted Array implementation of a priority queue
class PriorityQueueArray:
    def __init__(self):
        self.set = set()

    # Insert a node into the array
    def insertNode(self, index):
        self.set.add(index)

    # return node with the smallest distance, remove from list
    def deleteMin(self, dist):
        minIndex = next(iter(self.set)) # gets first element from set
```

```
        for num in self.set:
            if dist[num] == float('inf'):
                continue
            elif dist[minIndex] is float('inf') or dist[num] < dist[minIndex]:
                minIndex = num
        self.set.remove(minIndex)
        return minIndex

    # decrease the dist value at that index
    def decreaseKey(self, index, newDist):
        pass

    # Check if tree has more items
    def isEmpty(self):
        if len(self.set) == 0:
            return True
        else:
            return False


class PriorityQueueHeap:
    def __init__(self):
        self.tree = []
        self.pointers = []

    # Insert a new node
    def insertNode(self, node_id):
        # Put new node into next stop
        loc = len(self.tree)
```

```
        self.pointers.append(loc)
        self.tree.append((node_id, float('inf')))
        # Bubble up not necessary here as all inserts will have inf distance

    # return and remove the minimum node
    def deleteMin(self, dist):
        # Catch error case where only 1 item in tree
        if len(self.tree) == 1:
            topNode_id, topNode_dist = self.tree.pop()
            return topNode_id
        # Get top node and remove from tree
        topNode_id, topNode_dist = self.tree[0]
        self.pointers[topNode_id] = None
        # move last node to top
        self.tree[0] = self.tree[-1]
        self.tree.pop()
        # Bubble top node down
        bottomNode_id, bottomNode_dist = self.tree[0]
        self.pointers[bottomNode_id] = 0
        self.bubbleDown(bottomNode_id)
        return topNode_id

    # lower the value of a node
    def decreaseKey(self, node_id, newDist):
        # get location from pointers
        loc = self.pointers[node_id]
        # Change value at that location
        self.tree[loc] = (node_id, newDist)
        # Bubble up
```

```
    self.bubbleUp(node_id)

# Bubble a value up in the tree
def bubbleUp(self, node_id):
    cur_id = node_id
    while True:
        cur_loc = self.pointers[cur_id]
        if cur_loc == 0:  # Check for top
            break
        parent_loc = (cur_loc - 1) // 2
        cur_id, cur_value = self.tree[cur_loc]
        parent_id, parent_value = self.tree[parent_loc]
        if cur_value < parent_value:  # swap child with parent if less
            self.tree[parent_loc] = self.tree[cur_loc]
            self.pointers[cur_id] = parent_loc
            self.tree[cur_loc] = (parent_id, parent_value)
            self.pointers[parent_id] = cur_loc
        else:
            break

# Bubble a value down in the tree
def bubbleDown(self, node_id):
    cur_id = node_id
    while True:
        cur_loc = self.pointers[cur_id]
        first_child_loc = round((cur_loc + 0.5) * 2)
        second_child_loc = (cur_loc + 1) * 2
        cur_id, cur_value = self.tree[cur_loc]
        # Check if children are valid
```

```
maxLoc = len(self.tree) - 1
if first_child_loc > maxLoc and second_child_loc > maxLoc:
    break
else:
    if first_child_loc > maxLoc:
        first_child_value = float('inf')
        second_child_id, second_child_value = self.tree[second_child_loc]
    elif second_child_loc > maxLoc:
        first_child_id, first_child_value = self.tree[first_child_loc]
        second_child_value = float('inf')
    else:
        first_child_id, first_child_value = self.tree[first_child_loc]
        second_child_id, second_child_value = self.tree[second_child_loc]

if first_child_value <= second_child_value:
    child_id = first_child_id
    child_value = first_child_value
    child_loc = first_child_loc
else:
    child_id = second_child_id
    child_value = second_child_value
    child_loc = second_child_loc
if cur_value > child_value:  # swap child with parent if less
    self.tree[child_loc] = self.tree[cur_loc]
    self.pointers[cur_id] = child_loc
    self.tree[cur_loc] = (child_id, child_value)
    self.pointers[child_id] = cur_loc
else:
    break
```

```
    # Check if tree has more items
    def isEmpty(self):
        if len(self.tree) == 0:
            return True
        else:
            return False


class NetworkRoutingSolver:
    def __init__(self):
        self.dist = None
        self.prev = None
        self.source = None
        self.dest = None
        self.network = None

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

    def getShortestPath(self, destIndex):
        self.dest = destIndex
        path_edges = []
        total_length = self.dist[self.dest]
        index = self.dest

        # Check for unreachable
        if self.prev[destIndex] is None:
```

```
        return {'cost': float('inf'),
                'path': path_edges}

    # search backwards using prev pointers to find all edges used
    while self.prev[index] is not None:
        previous = self.prev[index]
        prevNode = self.network.nodes[previous]
        for edge in prevNode.neighbors:
            if edge.dest.node_id == index:
                path_edges.append((edge.src.loc, edge.dest.loc, '{:.0f}'.format(edge.length)))
        index = previous

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

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

    # Choose which heap implementation to use
    if use_heap:
        print("Using heap implementation")
        Q = PriorityQueueHeap()
    else:
        print("Using array implementation")
        Q = PriorityQueueArray()

    dist = []
    prev = []
    # load the queue with all the points
```

```
for i in range(len(self.network.nodes)):
    dist.insert(i, float('inf'))
    prev.insert(i, None)
    Q.insertNode(i)
dist[srcIndex] = 0
Q.decreaseKey(srcIndex, 0)

# loop until queue is empty
while not Q.isEmpty():
    # get minimum node from queue
    uInd = Q.deleteMin(dist)

    # check if queue is still returning reachable nodes
    if dist[uInd] == float('inf'):
        print("All reachable nodes searched")
        break

    # update values for each neighboring node
    for edge in self.network.nodes[uInd].neighbors:
        vInd = edge.dest.node_id
        newDist = dist[uInd] + edge.length
        if dist[vInd] == float('inf') or newDist < dist[vInd]:
            dist[vInd] = newDist
            prev[vInd] = uInd
            Q.decreaseKey(vInd, newDist)

# set values for use in getPath function
self.dist = dist
self.prev = prev
```

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