CS 312 Network Routing Report

Cannon Farr

Part 1. Dijkstra's Algorithm and Code:

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


        path_edges = []
        #Set start node as the destination node
        node = self.network.nodes[destIndex]
        total_length = node.dist

        # Work backwards from the destination node to the source node
        while node is not None:
            prev = node.prev
            if prev is None: # Found the source node
                break
            for edge in prev.neighbors: # Find the edge that connects the
previous node to the current node
                if edge.dest.node_id == node.node_id:
                    path_edges.append( (edge.src.loc, edge.dest.loc,
'{:.0f}'.format(edge.length)) )
            node = node.prev # Move to the previous node
        return {'cost':total_length, 'path':path_edges}

    def computeShortestPaths( self, srcIndex, use_heap=False ):
        self.source = srcIndex
        t1 = time.time()
        if use_heap:
            q = PriorityQueueBinaryHeap(self.network.nodes)
        else:
            q = PriorityQueueArray(self.network.nodes)

        # Initialize all nodes with distance of infinity and no previous node
        for node in self.network.nodes:
            node.dist = float('inf')
            node.prev = None
        self.network.nodes[srcIndex].dist = 0

        # Build and populate queue object with all nodes
        q.make_queue(self.network.nodes)
```

```
        # Run Dijkstra's algorithm iteratively removing the node with the
smallest distance
        while len(q.queue) > 0:
            u = q.delete_min()
            # Find each neighbor of the current node and update its distance if
it is less than the current distance
            for edge in u.neighbors:
                dest = edge.dest
                if u.dist + edge.length < dest.dist:
                    dest.dist = u.dist + edge.length
                    dest.prev = u
                    # Decrease the key of the node in the queue
                    q.decrease_key(dest, dest.dist)

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

Part 2. Priority Queue Implementations

Unsorted Array:

```
class PriorityQueueArray:
    def __init__(self, nodes):
        self.queue = []
        self.nodes = nodes

    # Insert a node into the queue
    def insert(self, node):
        self.queue.append(node.node_id)

    # Build the queue with a list of nodes
    def make_queue(self, nodes):
        for node in nodes:
            self.insert(node)

    # Remove the node with the smallest distance from the queue
    def delete_min(self):
        minIndex = 0
        nodeIndex = 0
        for i in range(len(self.queue)): # Iterate through the queue to find the
node with the smallest distance
            if self.nodes[self.queue[i]].dist <
self.nodes[self.queue[minIndex]].dist:
                minIndex = i
                nodeIndex = self.queue[i]
```

```
        del self.queue[minIndex] # Remove the node with the smallest distance
from the queue
        return self.nodes[nodeIndex]

    # Has no funcionality for this implementation
    def decrease_key(self, value, priority):
        return
```

In the implementation above, the Priority queue is represented by an unsorted array of node IDs. We can analyze each of the operations and their complexity:

Insert: Insert simply appends a node_id to the end of the unsorted array. This is O(1).

Delete Min: delete min iterates through the length of the queue and finds the element that is the smallest and removes it. This iteration causes the operation to be O(|V|)

Decrease Key: In my array implementation, Decrease key doesn't have any functionality since I am storing the distances on the nodes themselves and changing them within dijkstras algorithm. We'll call this O(1).

Binary Heap:

```
class PriorityQueueBinaryHeap:
    def __init__(self, nodes):
        self.queue = []
        self.pointerDict = {}
        self.nodes = nodes

    # Returns the index of the parent node
    def parent(self, i):
        return (i-1)//2

    # Returns the index of the left child node
    def left_child(self, i):
        return 2*i + 1

    # Returns the index of the right child node
    def right_child(self, i):
        return 2*i + 2

    # Insert a node into the heap
    def insert(self, node):
        self.queue.append(node.node_id) # Add the node to the end of the queue
        self.pointerDict[node.node_id] = len(self.queue)-1
        self.__bubble_up(len(self.queue)-1) # Bubble up the node to its correct
position

    # Build the queue with a list of nodes
```

```
    def make_queue(self, nodes):
        for node in nodes:
            self.insert(node)

    # Remove the node with the smallest distance from the heap
    def delete_min(self):
        if len(self.queue) == 0:
            return None
        minNode = self.nodes[self.queue[0]] # The node with the smallest distance
is at the root of the heap
        self.queue[0] = self.queue[-1] # Replace the root with the last node in
the heap
        self.pointerDict[self.queue[0]] = 0
        del self.queue[-1]
        self.__bubble_down(0) # Bubble down the new root to its correct position
        return minNode

    # Decrease the distance of a node in the heap
    def decrease_key(self, value, priority):
        index = self.pointerDict[value.node_id]
        self.nodes[value.node_id].dist = priority # Update the distance of the
node
        self.__bubble_up(index) # Bubble up the node to its correct position
        return

    # Helper function to swap two nodes in the heap
    def __swap(self, i, j):
        if i >= len(self.queue) or j >= len(self.queue):
            return
        self.pointerDict[self.queue[j]] = i
        self.pointerDict[self.queue[i]] = j
        tmp = self.queue[i]
        self.queue[i] = self.queue[j]
        self.queue[j] = tmp

    # Helper function to bubble up a node to its correct position
    def __bubble_up(self, i):
        # Keep bubbling until the node is at the root or its parent has a smaller
distance
        while i > 0 and self.nodes[self.queue[self.parent(i)]].dist >
self.nodes[self.queue[i]].dist:
            self.__swap(i, self.parent(i))
            i = self.parent(i)

    # Helper function to bubble down a node to its correct position
```

```
    def __bubble_down(self, i):
        # Keep bubbling until the node is a leaf or its children have a larger
distance
        if i >= len(self.queue) or self.left_child(i) >= len(self.queue) or
self.right_child(i) >= len(self.queue):
            return
        minIndex = i
        l = self.left_child(i)
        r = self.right_child(i)

        # Find the child with the smallest distance
        minIndex = l if self.nodes[self.queue[minIndex]].dist >
self.nodes[self.queue[l]].dist else minIndex
        minIndex = r if self.nodes[self.queue[minIndex]].dist >
self.nodes[self.queue[r]].dist else minIndex

        if i != minIndex:
            self.__swap(i, minIndex) # Swap the node with the child with the
smallest distance
            self.__bubble_down(minIndex) # Recursively bubble down the nod
```

The Binary Heap implementation is still represented in an array data structure but nodes are sorted into a tree. Similarly to the array implementation, we will analyze each operation of the binary heap queue:

Insert: Inserting to the heap adds a node_id to the end of the queue and then “bubbles it up” until its distance is less than all of its children’s and greater than its parent’s. This bubbling up operation will take at most log|v| times because the worst case will be having to traverse to the very top of the tree. Therefore insertion is O(log |v|)

Delete Min: Deleting the minimum element of the heap is an O(1) operation because it is just the top element, but the process of rebalancing or “bubbling up” is an O(log|v|) operation. This is because the worst case is for the element to be bubbled up to the top of the heap which is log|v| levels.

Decrease Key: Similarly, the decrease key is also O(log|v|) for the same reason as deleting the minimum. The operation of actually updating the distance on the node is constant time, but the operation to rebalance the tree is the same process except we are comparing and pushing the element down the tree rather than up. This is still a worst case of O(log|v|).

Part 3. Time and space complexity

The Pseudocode of Dijkstras Algorithm is as follows:Tim

Dijkstra's Algorithm

```
Dijkstra(G, l, s)
    for all u ∈ V do
        dist(u) ← ∞
        prev(u) ← NULL
    dist(s) ← 0
    H.makequeue(V, dist)
    while H is not empty do
        u ← H.deletemin()
        for all edges (u, v) ∈ E do
            if dist(v) > dist(u) + l(u, v) then
                dist(v) ← dist(u) + l(u, v)
                prev(v) ← u
                H.decreasekey(v)
    return dist, prev
```

Time Complexity

Walking through Dijkstra's algorithm, we can analyze the time complexity of each step.

1. For all nodes u in Graph V, set the distances to infinity and the previous to null.
    a. Looping through each node in V gives us O(|V|)
2. The MakeQueue operation calls the method on the priority queue implementation. From the previous segment, we concluded that they have the following complexities:
    a. Unsorted Array – O(1)
    b. Binary Heap – O(log|V|)
3. While H is not empty do… For everything within H, we will run this O(|V|) times.
4. U.DeleteMin(). Again, these have different complexities based on the implementation as previously discussed:
    a. Unsorted Array – O(|V|)
    b. Binary Heaap – O(log|V|)
5. For all edges (u,v) from node E do… everything under this for loop will run O(|E|) times.
6. The only significant piece inside the for loop is decrease key, which we also know is different per implementation:
    a. Unsorted Array – O(|1|)
    b. Binary Heap – O(log|V|)

Adding and multiplying all of these complexities together, we get the following overall complexity:

$$O(MQ + DM \cdot |V| + DK \cdot |E|)$$

Then finally, we plug in the complexities for each implementation to get their final complexities:

Unsorted Array: $O(|V|^2)$

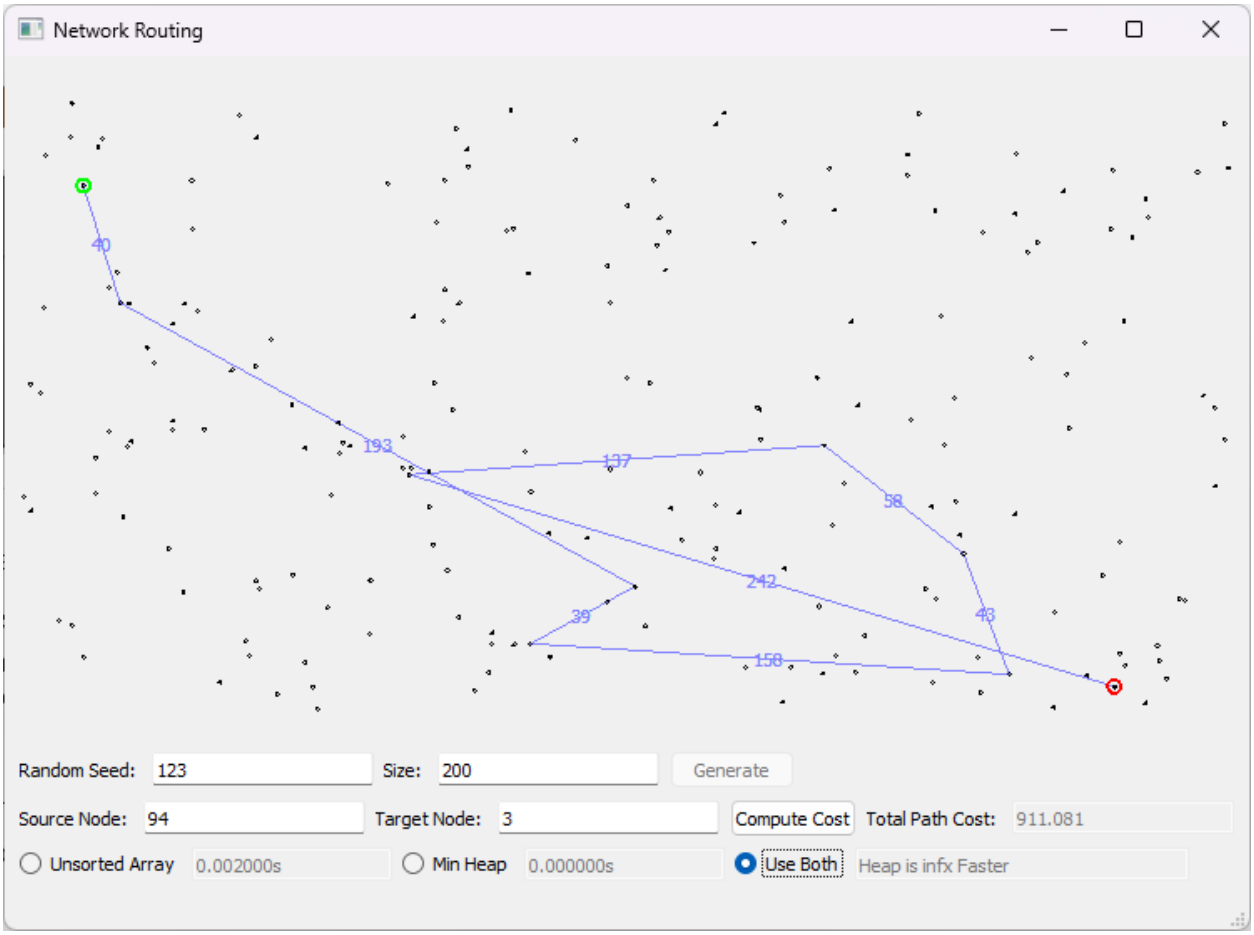
Binary Heap: $O\big((|V| + |E|)\log\ |V|\big)$

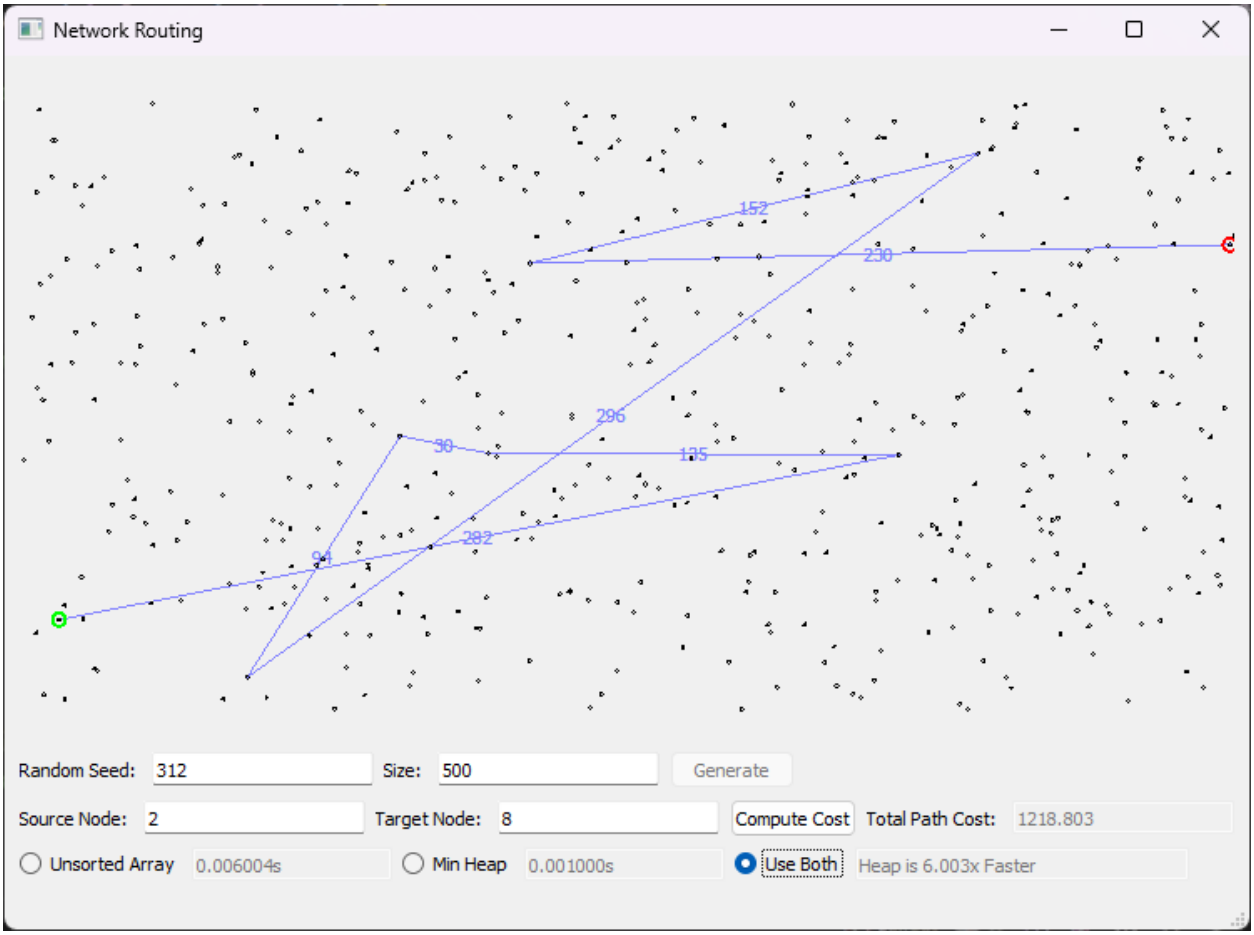
Space Complexity:

For each implementation, the space complexity is as follow:
Unsorted Array: The unsorted Array implementation uses an array of node_ids to represent the queue. This array is a space of O(|V|); one index for each node. Additionally, the distances and previous nodes need to be stored on the node objects themselves. This will be O(|V|) each. Giving us a final space complexity of 3|V| or O(|V|).

Binary Heap: The binary heap has a similar space complexity as the sorted array. The heap tree is also implemented in an array with |V| indexes for each node_id. Additionally, it also needs to store the distance and previous node on the node objects. We add another 2|V|. Finally, the binary heap has one additional data structure, and that is the pointer dictionary that is used to map the node_ids back to the indexes of their location in the heap. This also is one element for every node. Adding all of these gives a final space complexity of 4|V| or O(|V|)

Part 4. Test Screenshots


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
0.002000s
Min Heap
0.000000s
Use Both
Heap is infx Faster

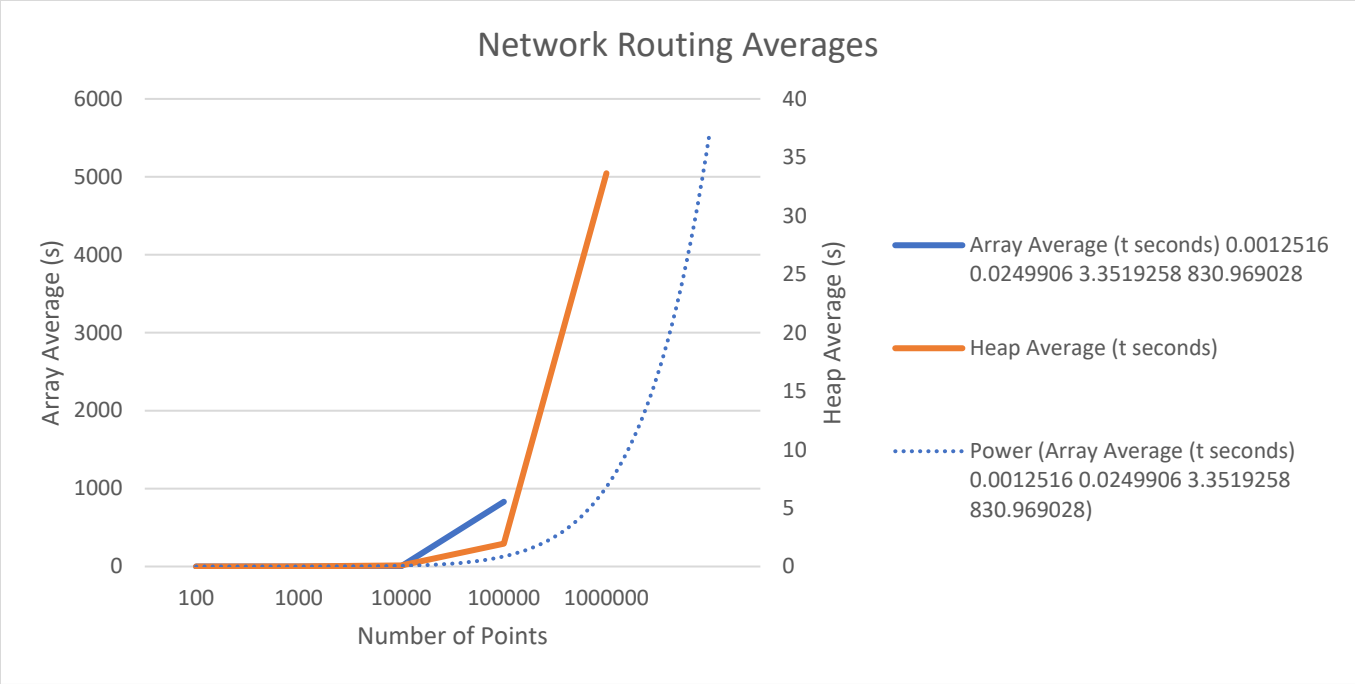


Part 5.

Empirical Time Complexities

n (# of points)	Array 1	Array 2	Array 3	Array 4	Array 5	**Array Average (t seconds)**
100	0.00503	0	0.001228	0	0	**0.0012516**
1000	0.024567	0.025223	0.02513	0.025522	0.024511	**0.0249906**
10000	3.183971	3.252566	3.448388	3.451603	3.423101	**3.3519258**
100000	743.0964	925.5911	844.5935	803.1641	838.4001	**830.969028**
1000000	-	-	-	-	-	**~ 5500**

n (# of points)	Heap 1	Heap 2	Heap 3	Heap 4	Heap 5	**Heap Average (t seconds)**
100	0	0.000504	0	0	0	**0.0001008**
1000	0.005506	0.005002	0.004651	0.004675	0.004791	**0.004925**
10000	0.083649	0.086151	0.092335	0.086497	0.086845	**0.08709548**
100000	1.953083	1.977966	1.953086	1.973199	1.93008	**1.9574828**
1000000	33.71596	33.34294	32.86028	35.64084	32.63626	**33.6392544**



The graph above shows that our complexity calculations were correct. The Heap average grows at an approximately nlogn scale. And the Array Average grows at about n squared. I estimated that given the trend of the Array Averages, it would take around 5500 seconds for the array implementation to calculate the path for 1 million points.