Network Routing

Camy Ngo CS312

Array

My array implementation has time complexity of O(|V|^2) and space complexity is O(|V|)

Insert takes O(1) time. Because appending an item to the list is constant

Delete-min in my implementation is **O(|V|)** as it has to go through all items in the list

Decrease-key takes O(1) time. It simply swap the value by using insert function, hence why its O(1)

Heap

Insert takes **O(logn)** a worst-case scenario as it would need to descend to the lowest nodes of the binary tree, which has height of **logn**, to append a large value.

Delete-min in the heap is **O(logn).** Returning the minimum itself is only O(1) as the root of any minheap is the minimum value by definition, but then the tree has to be "build_heap" to maintain it's nature.
Function build_heap takes O(**logn)** if a value has to be shifted down the entire length of the tree because of this.

Decrease-key also takes **O(logn)** in a worst-case scenario if the value I want to change is at the bottom of the binary tree.

Space complexity analysis

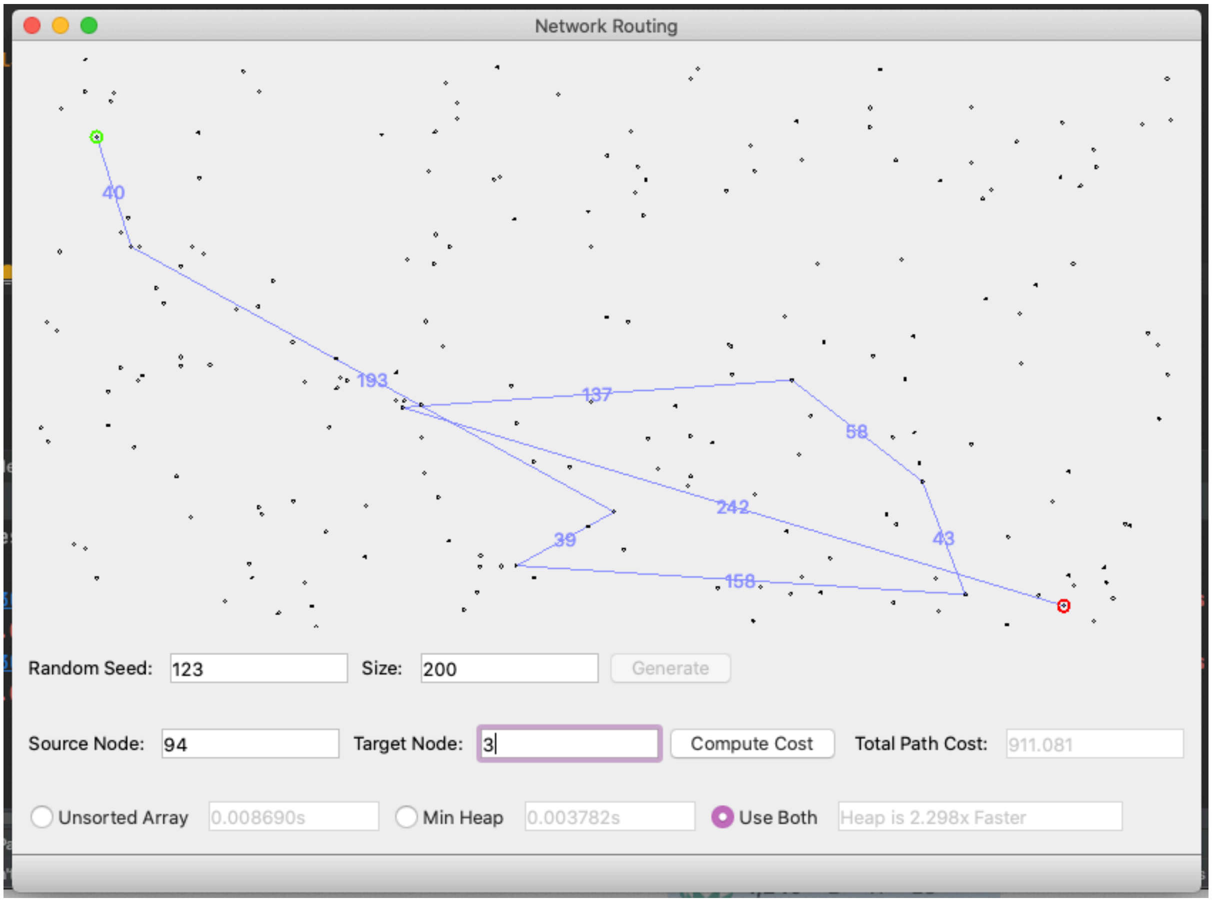
Dijkstra itself consists of two main data structures aside from the graph it analyses.

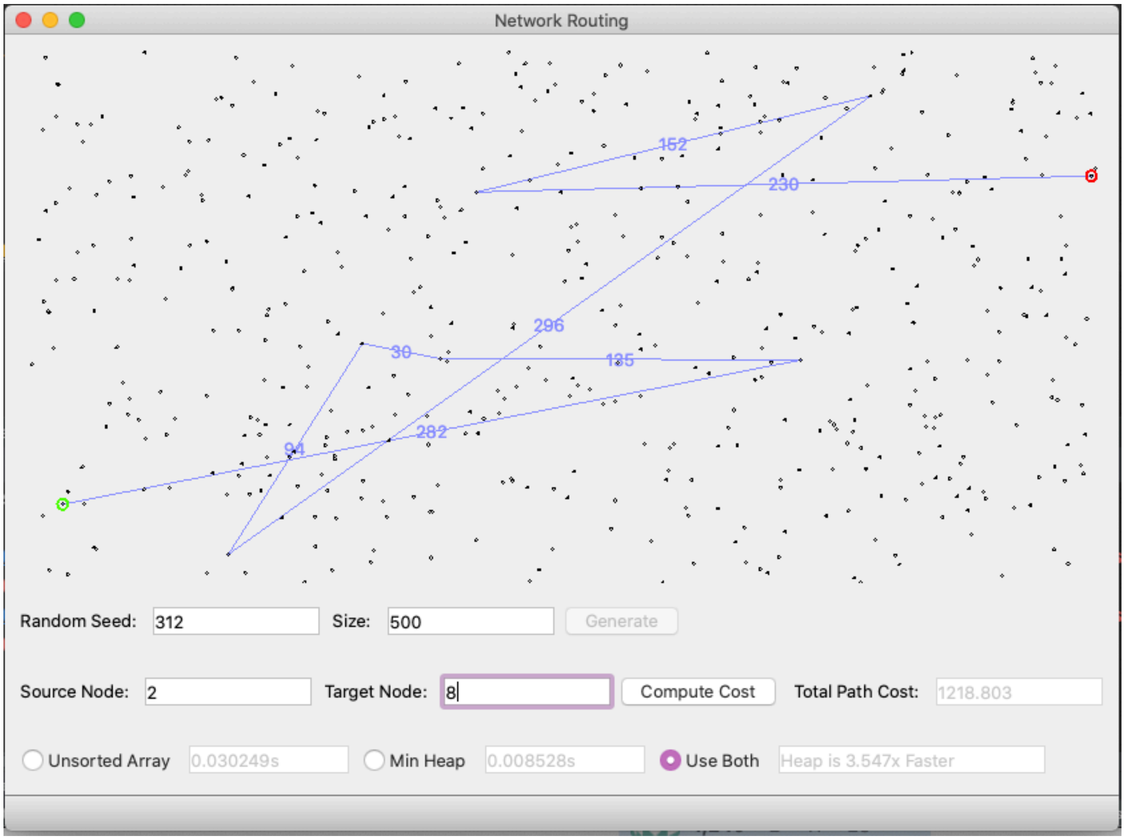
For first structure - the heap, space complexity **is O(|V| + |E|)** because it has to go through all nodes and edges. The compute path using heap has build_heap with complexity of O|V| and an inner loops that goes through all edges with complexity O(|E|)

The second structure is a priority queue, which is implemented as an unsorted list. The unsorted list is fairly straightforward, as there are only as many elements in the list as there are nodes, so the space complexity for the list is **O(3|V|) = O(|V|)** since it has to all through all V nodes, and insert + delete + decrease_key all has space complexity of O(|V|).

Pictures:


Network Routing
Random Seed: 42
Size: 20
Generate
Source Node: 7
Target Node: 1
Compute Cost
Total Path Cost: 0.000
Unsorted Array 0.000325s
Min Heap 0.000426s
Use Both
Heap is 0.763x Faster


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 0.008690s
Min Heap 0.003782s
Use Both
Heap is 2.298x Faster

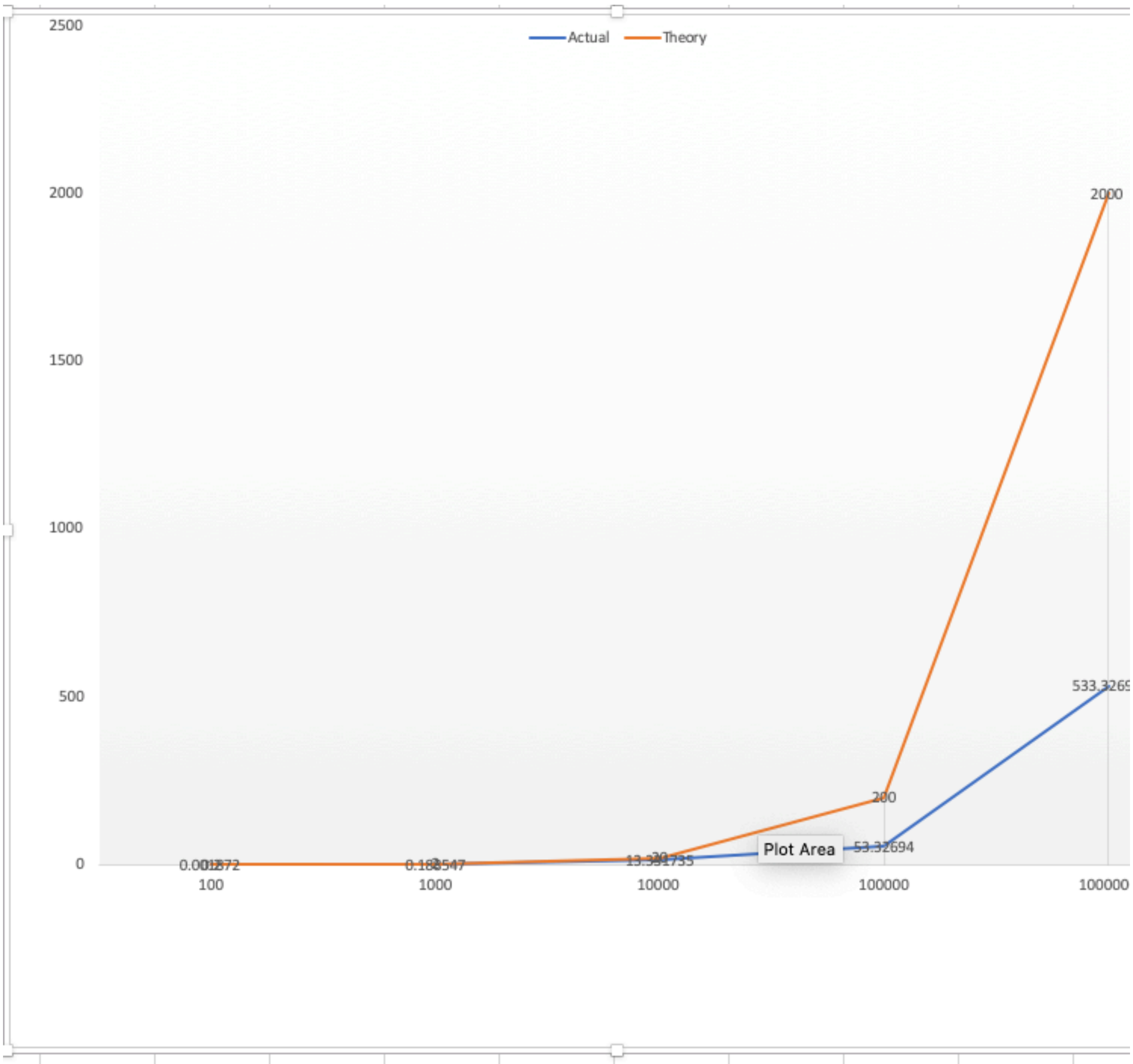


Empirical Analysis

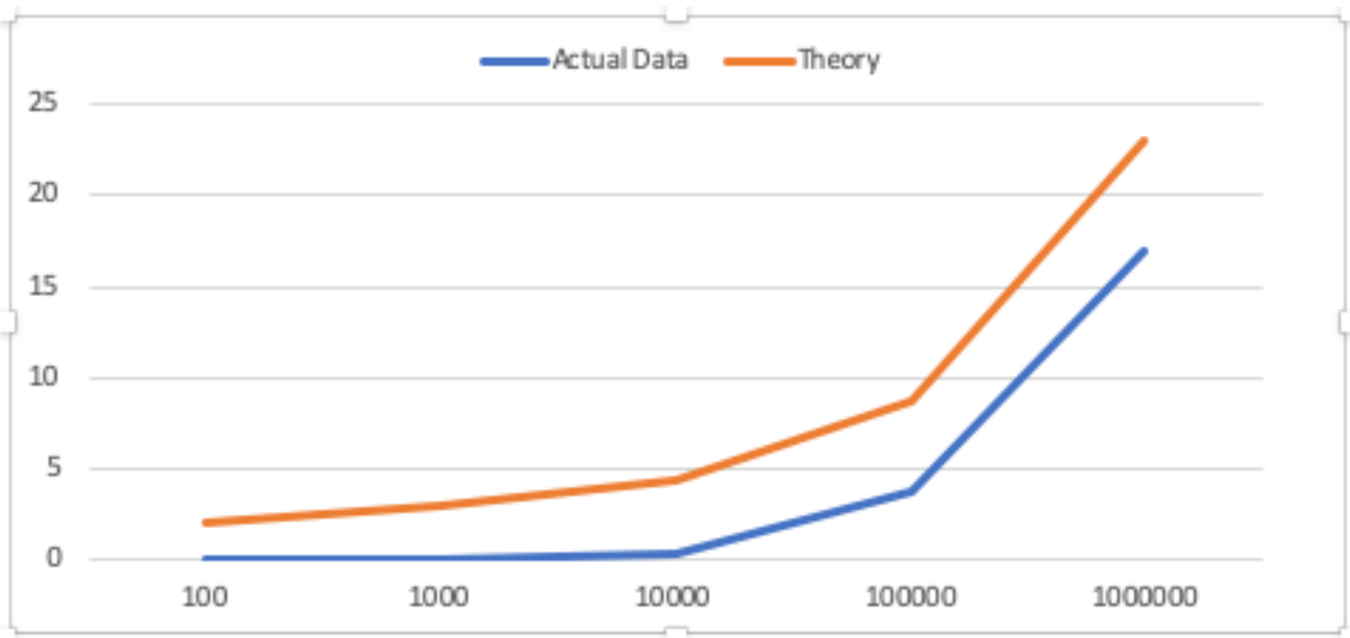
For the array and heap implementation empirical data

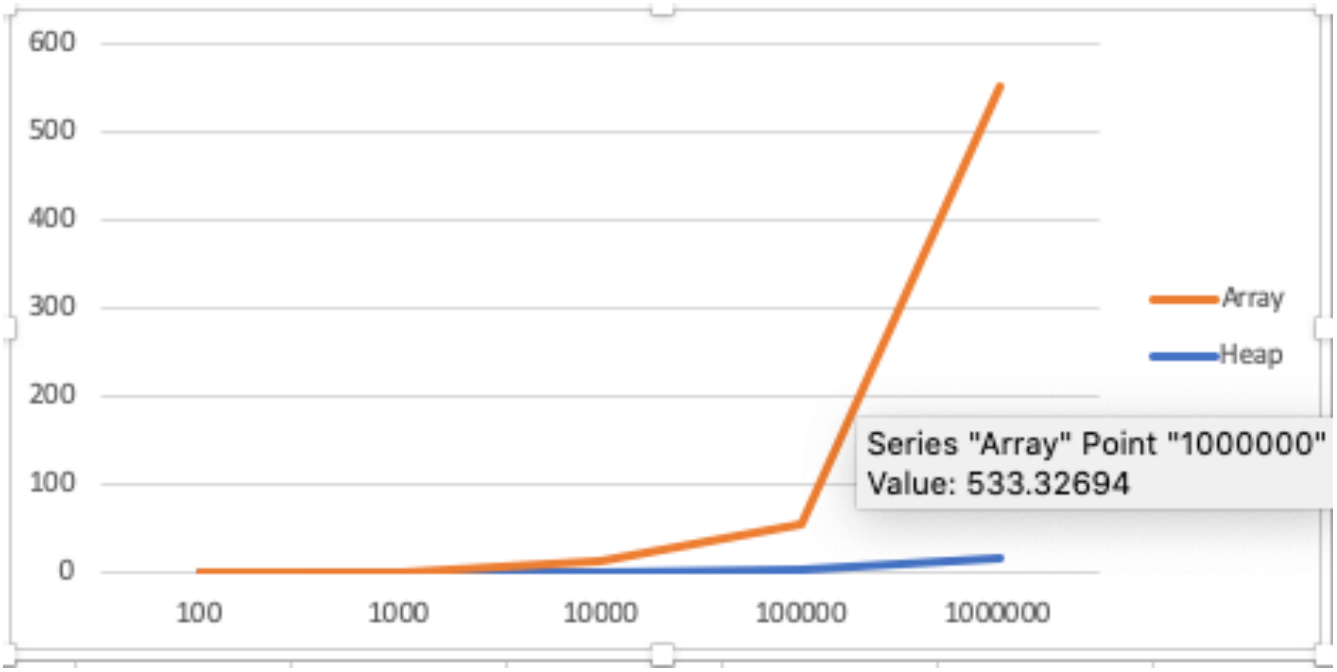
	N	Array time	Heap_time
0	100	0.001872	0.001575
1	1000	0.188547	0.026297
2	10000	13.331735	0.313420
3	100000	53.32694	3.76104
4	1000000	533.32694	16.92468

Array Queue with Dijkstra’s



Heap Queue with Dijkstra’s





The fact that the graphs don't match perfectly illustrate the nature of Big O notation, that it estimates the worst-case scenario. It is interesting how with smaller n-values, the array and heap is not giving much of a different. This is due to the scanning though arrays for small values run really fast. When the number > 1000, we start to see a huge different

Code:

Arrayqueue.py

```
import math


class ArrayQueue(object):
    def __init__(self):
        self.array_size = 0
        self.array_list = []

    # Time complexity: O(|V^2|)
    # Space complexity: O(|V|) simplifies from O(3|V|)
    def computePathsArray(self, srcIndex, nodes, destIndex):
        # Initialize array to keep track of distances
        dist = [math.inf] * len(nodes)
        shortest_Dist = [math.inf] * len(nodes)
        shortest_Dist[srcIndex] = 0
        dist[srcIndex] = 0
        self.array_size = len(nodes)
```

```
        prev = [-math.inf] * len(nodes)
        # insert nodes
        self.insert((srcIndex, 0))
        # run while loop until size of our priority queue is zero
        while self.array_size > 0:
            # call delete_min
            index, weight, pq_index = self.delete_min(self.array_list)

            # set distance of current node in our priority queue to -1, so we don't use
it again
            self.array_list[pq_index] = (index, -1)

            # visit neighbors of current node
            for edge in nodes[index].neighbors:
                curr_index = edge.dest.node_id
                curr_weight = edge.length + weight
                # if node hasn't been visited, or we found shorter path update distance
array
                if (dist[curr_index] == math.inf) or (dist[curr_index] > curr_weight):
                    shortest_Dist[curr_index] = edge.length
                    # the decrease_key for the array implementation
                    # Decrease_key is O(1) because we are appending it to the array
                    # Space complexity is O(|V|)
                    self.insert((curr_index, curr_weight))
                    dist[curr_index] = curr_weight
                    prev[curr_index] = index
        return shortest_Dist, prev

    # insert has time complexity of O(1)
    # Space complexity is O(|V|)
    def insert(self, node):
        self.array_list.append(node)

    # Time complexity: O(|V|) because we have to search through array
    # Space complexity: O(|v|) because we have to look through array
    def delete_min(self, array):
        index = 0
        array_min_index = 0
        min_weight = 0
        min_index = 0
        # search in priority queue for the next smallest value to remove from the queue
        for (x, y) in array:
            if min_weight == 0 and y != -1 or y < min_weight and y != -1:
                min_weight = y
                min_index = x
                array_min_index = index
            index = index + 1
        self.array_size -= 1
        return min_index, min_weight, array_min_index
```

Heapqueue.py

```
import math


class HeapQueue(object):
    def __init__(self):
        self.pointer = None
        self.heap = None
        self.heap_count = None

    # Time: O((|V| + |E|) log V)
    # Space complexity: O(|V| + |E|)
    def computePathsHeap(self, srcIndex, nodes, destIndex):
        # create distance and previous arrays
        distance = [math.inf] * len(nodes)
        prev_node = [-math.inf] * len(nodes)
        distance[srcIndex] = 0
        shortest_Dist = [math.inf] * len(nodes)
        self.heap_count = len(nodes) - 1
        # O(|V|) complexity
        self.heap, self.pointer = self.build_Heap(nodes, srcIndex)

        # run until there are no nodes left
        while self.heap_count > 0:
            index, weight = self.deleteMin_Heap()
            # explore current nodes neighbors
            for edge in nodes[index].neighbors:
                curr_index = edge.dest.node_id
                curr_weight = weight + edge.length
                # if current distance in distance[] array is larger, then we have found
a shorter path
                if distance[curr_index] > curr_weight:
                    distance[curr_index] = curr_weight
                    shortest_Dist[curr_index] = edge.length
                    prev_node[curr_index] = index

                    # call decrease_key
                    self.decreaseKey_Heap(curr_index, curr_weight)
        return shortest_Dist, prev_node

    # Time complexity: O(|V|)- loop over array of size |V| once to insert nodes to Heap
    # Space complexity: O(|V|)- creating heep and pointer arrays which are of size |V|
    def build_Heap(self, nodes, srcIndex):
        pointer = [-1] * len(nodes)
        heap = [] * len(nodes)
        # set pointer and heap for the source node
        pointer[srcIndex] = 0
        heap.append((srcIndex, 0))
        counter = 1
        for node in nodes:
            # append all nodes except for the source node
            if node.node_id != srcIndex:
                heap.append((node.node_id, math.inf))
                pointer[node.node_id] = counter
                counter += 1
        return heap, pointer
```

```python
    # Time complexity: O(log|V|) because we are doing a few order one operations then
calling siftDown() which is O(log|V|)
    # Space complexity: O(1)
    def deleteMin_Heap(self):
        # get index and weight of root node
        index, weight = self.heap[0]
        self.pointer[index] = - 1
        # set root node to value of last node in array
        self.heap[0] = self.heap[self.heap_count]
        newIndex, newWeight = self.heap[0]
        self.pointer[newIndex] = 0
        self.heap_count -= 1
        # call shift-down
        self.shift_down()
        return index, weight

    # Time complexity: O(log|V|) because we are doing two order one operation and
calling bubbleUp which has time
    # complexity of O(log|V|)
    # Space complexity: O(1) not allocating much memory
    def decreaseKey_Heap(self, index, dist):
        # get heap index from pointers to update the node value
        new_index = self.pointer[index]
        self.heap[new_index] = (index, dist)
        self.bubble_up(new_index)

    # Time complexity: O(log|V|) just traversing a tree which is logV complexity at each
layer of tree
    # Space complexity: O(1)
    def shift_down(self):
        # takes root node and see if it is larger than it's children
        is_bigger = False
        count = 0
        currNode = self.heap[0]
        currVal = currNode[1]
        isRightNone = False
        while not is_bigger:
            # get left and right children
            leftChild, rightChild = self.get_child(count)
            if rightChild is None:
                isRightNone = True
            if leftChild is not None:
                # if current value and it's children are infinity we are done
                if currVal == math.inf and not isRightNone:
                    if leftChild[1] == math.inf and rightChild[1] == math.inf:
                        break
                # check left child
                if not isRightNone and currVal > leftChild[1] and leftChild[1] <
rightChild[1]:
                    parent = self.heap[count]
                    self.heap[count] = leftChild
                    self.heap[self.pointer[leftChild[0]]] = parent
                    self.pointer[parent[0]] = self.pointer[leftChild[0]]
                    self.pointer[leftChild[0]] = count
                    count = self.pointer[parent[0]]
```

```
                # check right child
                elif not isRightNone and currVal > rightChild[1]:
                    parent = self.heap[count]
                    self.heap[count] = rightChild
                    self.heap[self.pointer[rightChild[0]]] = parent
                    self.pointer[parent[0]] = self.pointer[rightChild[0]]
                    self.pointer[rightChild[0]] = count
                    count = self.pointer[parent[0]]
                else:
                    is_bigger = True
            else:
                is_bigger = True
        return

    # Time Complexity: O(1)- doing a few order one operation
    # Space Complexity: O(1)- not allocating very much memory
    def get_child(self, index):
        # formula for index of left and right children
        left_index = (index * 2) + 1
        right_index = (index * 2) + 2
        left_child = None
        right_child = None
        # make sure we stay in bounds of our heap
        if left_index < self.heap_count:
            left_child = self.heap[left_index]
        if right_index < self.heap_count:
            right_child = self.heap[right_index]
        return left_child, right_child

    # Time complexity: O(log|V|) because we are bubbling up a tree which is log time at
each level
    # Space complexity: O(1) not really allocating any memory
    def bubble_up(self, index):
        is_greater = False
        child_index = index
        while not is_greater:
            # if we are at root break
            if child_index == 0:
                break
            parent_index = self.get_parent(child_index)
            child = self.heap[child_index]
            parent = self.heap[parent_index]
            # swap if child is less than parent
            if child[1] < parent[1]:
                self.heap[parent_index] = child
                self.heap[child_index] = parent
                # update the pointer array
                self.pointer[child[0]] = parent_index
                self.pointer[parent[0]] = child_index
                child_index = parent_index
            else:
                is_greater = True
        return

    # Time complexity: O(1)
    # Space complexity: O(1)
```

```
    def get_parent(self, index):
        return math.floor((index - 1) / 2)
```

Networkroutingsolver.py

```
#!/usr/bin/python3
from CS312Graph import *
import time
import math
from ArrayQueue import ArrayQueue
from HeapQueue import HeapQueue


class NetworkRoutingSolver:
    def __init__(self):
        self.prev_node = None
        self.distance = None
        self.destination_node = None
        self.network = None
        self.source = None

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

    def getShortestPath(self, destIndex):
        self.destination_node = destIndex
        # TODO: RETURN THE SHORTEST PATH FOR destIndex
        #       INSTEAD OF THE DUMMY SET OF EDGES BELOW
        #       IT'S JUST AN EXAMPLE OF THE FORMAT YOU'LL
        #       NEED TO USE

        path_edges = []
        total_length = 0

        source_node = self.network.nodes[self.source]
        source_id = source_node.node_id

        destination_node = self.network.nodes[destIndex]
        current_id = destination_node.node_id
        nodes = self.network.nodes

        if self.prev_node[current_id] == -math.inf:
            return {'cost': total_length, 'path': []}

        while current_id != source_id:
            prev_id = self.prev_node[current_id]
            if prev_id is None:
                return {'cost': total_length, 'path': []}
            node_dist = self.distance[current_id]
            path_edges.append((nodes[current_id].loc, nodes[prev_id].loc,
'{:.0f}'.format(node_dist)))
```

```python
            total_length += node_dist
            current_id = prev_id
        return {'cost': total_length, 'path': path_edges}

    # Time complexity: O((E + V)log(|V|)) because we call deleteMinHeap() |V| times 
worst case and we also call
    # decreaseKeyHeap() |E| times worst case. Both of these functions are O(log|V|)
    def computeShortestPaths(self, srcIndex, destIndex, use_heap=False):
        self.source = srcIndex
        t1 = time.time()
        # TODO: RUN DIJKSTRA'S TO DETERMINE SHORTEST PATHS.
        #       ALSO, STORE THE RESULTS FOR THE SUBSEQUENT
        #       CALL TO getShortestPath(dest_index)
        nodes = self.network.nodes

        if use_heap:
            Q = HeapQueue()
            dist, prev = Q.computePathsHeap(srcIndex, nodes, destIndex)
        else:
            Q = ArrayQueue()
            dist, prev = Q.computePathsArray(srcIndex, nodes, destIndex)

        self.distance = dist
        self.prev_node = prev
        t2 = time.time()
        return t2 - t1
```