1.

```
#!/usr/bin/python3
import math

from CS312Graph import *
import time

# Class that keeps track of Node dist and prev and index in heap for fast look up
# All functions are an amortized time complexity of O(1)
# However space is O(|V|) since there are three different dictionaries of length |V| being made
class Dictionaries(object):
    def     init    (self):
        self.dist = {}
        self.prev = {}
        self.indexDict = {}

    def getDistance(self):  #Return whole distance dictionary
        return self.dist

    def setDistance(self, node, distance):  # set node(key) new value(distance)
        self.dist[node] = distance

    def getPrev(self):  #Return whole previous dictionary
        return self.prev

    def setPrev(self, node, prev):  # Set node(key) to new value(previous)
        self.prev[node] = prev

    def getNodeDist(self, node): # get the distance for key node
        return self.dist[node]

    def getNodePrev(self, node): # get the previous for key node
        return self.prev[node]

    def getindexDict(self):  # Return whole index dictionary
        return self.indexDict

    def getNodeIndex(self, node):  # get the index for key node
        return self.indexDict[node]

    def setindexDict(self, node, index):  # set node(key) to new value(index)
        self.indexDict[node] = index


class ArrayQueue:
    def __init__(self, network, dicts):
        self.network = network
        self.dicts = dicts
        self.H = []
        self.makeQueue()

    def decreaseKey(self, index):
        # O(1) for time and space since it is not doing or storing anything
        # Don't need since deletemin finds smallest key
        pass

    def makeQueue(self):
        # O(|V|) for time and space cause it is called |V| times and makes an array of size |V|
        nodes = list(self.network.nodes)
        for x in nodes:
            self.insert(x)
```

```
    def insert(self, node):
        # O(1) for time and O(|V|) space; it eventually adds |V| things to the array and does it does it in constant time
        self.H.append(node)

    def deleteMin(self):
        # O(|V|) it looks at the distance dictionary and grabs the smallest value.
        # it can do this at most the size of H which is |V|
        # Space complexity here is O(1), since we aren't storing any new information
        d = self.dicts.getNodeDist(self.H[0]) # O(1)
        index = 0
        for i in range(len(self.H)):
            x = self.dicts.getNodeDist(self.H[i]) # O(1)
            if x < d:
                d = x
                index = i
        return self.H.pop(index) # O(1)

    def getQueue(self):
        # Time and space both O(1) since it is just returning the queue and not adding to it
        return self.H


class HeapQueue:
    def __init__(self, network, dicts):
        self.network = network
        self.dicts = dicts
        self.H = []
        self.makeQueue()

    def decreaseKey(self, node):
        # O(log|V|) is time complexity cause the binary tree's height is at most log|V| which means that bubble up can take
            # this long.
        # Space complexity is O(1) since nothing is being added to the array
        index = self.dicts.getNodeIndex(node) # O(1) time and space
        return self.bubbleUp(index)  # O(log|V|) for time

    def makeQueue(self):
        # Time complexity is O(|V|log|V|) since it runs V times and inserting into a tree is log|V|
        # The space complexity is O(|V|) since we are making a new array of size |V|
        nodes = list(self.network.nodes)  # O(|V|)
        for x in nodes:  # O(|V|)
            self.insert(x)  # O(log|V|)

    def insert(self, node):
        # O(log|V|) is time complexity cause inserting a node into a binary tree can take at most the height of the tree
            # log|V|. this means the newest addition goes from the bottom all the way to the top
        # Space complexity is O(|V|) since that is the size of the heap array made
        childIndex = len(self.H)  # O(1) grab new node index
        self.H.append(node)  # O(1) to append
        self.dicts.setindexDict(node, childIndex)  # O(1) to set new key-value pair
        self.bubbleUp(childIndex)  # O(log|V|)

    def deleteMin(self):
        # Time complexity is O(log|V|) since the furthest a node can bubble down is from the top to the bottom
            # and the height of a binary tree is log|V|
        # Space complexity here is O(1), since we aren't storing any new information
        u = self.H[0]  # Take from the front cause it is highest priority O(1)
        z = self.H[len(self.H) - 1]  #Take the right most node and put it at the top O(1)
        self.H[0] = z  # O(1)
        self.dicts.setindexDict(self.H[0], 0)  #O(1)
        self.H.pop(len(self.H) - 1)  # Pop off the rightmost node since it is a duplicate now O(1)
        self.bubbleDown(0) # O(log|V|)
```

```
        return u

    def getQueue(self):
        # Time and space both O(1) since it is just returning the queue and not adding to it
        return self.H

    def bubbleUp(self, childIndex):
        # Time complexity here is O(log|V|) since the longest time bubble up a node can be from the bottom
            # to the top of the heap which has a height of log|V|
        # Space complexity is O(1) since we aren't adding or taking away from the array
        if childIndex == 0:  # Base case we are the root node now
            return
        parentIndex = ((childIndex - 1) // 2)  # O(1)

        x = self.dicts.getNodeDist(self.H[parentIndex])  # O(1)
        y = self.dicts.getNodeDist(self.H[childIndex]) # O(1)
        if x > y:
            self.switch(parentIndex, childIndex) # O(1)
            return self.bubbleUp(parentIndex)  # O(log|V|)
        else:
            return

    def switch(self, parentIndex, childIndex):
        # Time and space complexities are O(1) since switches happen in constant time and no new memory taken up
        temp = self.H[parentIndex]  # O(1)
        self.H[parentIndex] = self.H[childIndex]  # O(1)
        self.H[childIndex] = temp  # O(1)
        self.dicts.setindexDict(self.H[parentIndex], parentIndex)  # O(1)
        self.dicts.setindexDict(self.H[childIndex], childIndex)  # O(1)
        return

    def bubbleDown(self, parentIndex):
        # Time complexity is O(log|V|) since the longest time to bubble down a node can be from the top
            # to the bottom of the heap which has a height of log|V|
        # Space complexity is O(1) since we aren't adding or taking away from the array
        if parentIndex == (len(self.H) - 1):  # Base case shifted the node all the way to the end  O(1)
            return
        leftChildIndex = (2 * parentIndex) + 1  # O(1)
        rightChildIndex = (2 * parentIndex) + 2 # O(1)

        if leftChildIndex < len(self.H): # This means it has a left child # O(1)
            if self.dicts.getNodeDist(self.H[parentIndex]) > self.dicts.getNodeDist(self.H[leftChildIndex]): # O(1)
                if rightChildIndex < len(self.H): # Means it has a right child and need to switch with smallest distance # O(1)
                    if self.dicts.getNodeDist(self.H[leftChildIndex]) <= self.dicts.getNodeDist(self.H[rightChildIndex]):
                        # This means the left child is smaller or equal to right so just switch with left kid
                        self.switch(parentIndex, leftChildIndex) # O(1)
                        return self.bubbleDown(leftChildIndex) # O(log|V|)
                    else: # Right kid is smaller
                        self.switch(parentIndex, rightChildIndex) # O(1)
                        return self.bubbleDown(rightChildIndex) # O(log|V|)

        # no left child check if there is a right one
        if rightChildIndex < len(self.H): # O(1)
            if self.dicts.getNodeDist(self.H[parentIndex]) > self.dicts.getNodeDist(self.H[rightChildIndex]): # O(1)
                self.switch(parentIndex, rightChildIndex) # O(1)
                return self.bubbleDown(rightChildIndex) # O(log|V|)

class NetworkRoutingSolver:
    def __init__(self):
        self.dicts = Dictionaries()
        pass
```

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

    # Time complexity here is O(|V|) since the for loop isn't always gone
        #  through in the while loop cause of the sparsity of the graph so we can add |E| and |V| together as seen below
    # Space complexity is O(|V|) since we are making an array of path_edges and worst case would be to have all the
nodes
        # a part of the path
    def getShortestPath( self, destIndex ):
        self.dest = destIndex
        path_edges = []
        nodes = self.network.getNodes() # O(|V|)
        endNode = nodes[self.dest]  # O(1)
        total_length = self.dicts.getNodeDist(endNode)  # O(1)
        currentNode = endNode  # O(1)
        # Overall time here is O(|V| + |E|) = O(|V|)
        while self.dicts.getPrev().get(currentNode) is not None:  # O(|V|)
            nextNode = self.dicts.getPrev().get(currentNode)  # O(1)
            for edge in nextNode.neighbors:  # O(|E|) = O(|V|)
                if (edge.dest == currentNode):  # Only add edge if it is one we want O(1)
                    path_edges.append( (nextNode.loc, currentNode.loc, '{:.0f}'.format(edge.length)) )  #O(1) and space
O(|V|)
            currentNode = nextNode # O(1)
        return {'cost':total_length, 'path':path_edges}

    # Compute shortest paths has a time complexity of
        # O(|V^2|) if priority queue is an array and
        # O(|V|log|V|) if the priority queue is a heap and the graph is sparse
            # otherwise the O(|V^2|log|V|) since |E| becomes equal to |V^2| not |V| anymore
    # Space complexity is O(|V| + 3|V|) = O(|V|) since this is the size of the queue(|V|) and dictionaries(3|V|) made
    def computeShortestPaths(self, srcIndex, use_heap=False ):
        self.source = srcIndex
        t1 = time.time()
        # Run Dijkstra's
            # goes through the all the nodes and sets dist for a node to be inf
            # and prev to be nil O(|V|) for time O(|V^2|) for space cause two dictionaries are being stored
        for x in self.network.nodes:  # O(|V|)
            self.dicts.setDistance(x, math.inf)  # O(1)
            self.dicts.setPrev(x, None)  # O(1)
        nodes = self.network.getNodes()  # O(1)
        self.dicts.setDistance(nodes[self.source], 0)  # O(1)

        # This makeQueue runs in O(|V|) if it is an array and O(|V|log|V|) if it is a heap
        if use_heap:
            H = HeapQueue(self.network, self.dicts)  # O(|V|log|V|)
        else:
            H = ArrayQueue(self.network, self.dicts)  # O(|V|)
        # O(|E|) = O(|V|) cause of the predefined maxiumum out-degree then:
        # If queue is an array this runs O(|V^2| + |V|) = O(|V^2|)
            # If queue is a heap runs in O(|V|log|V| + |V|log|V|) which is just O(|V|log|V|)
            # no new memory taken up so space is O(1)
        while len(H.getQueue()) > 0:  # O(|V|) for both
            u = H.deleteMin()  # O(|V|) for array O(log|V|)
            for n in u.neighbors: # O(|V|) for array O(|V|log|V|) for heap since |E| = |V|
                v = n.dest  # u and v are neighbors, they have an edge together
                if self.dicts.getNodeDist(v) > (self.dicts.getNodeDist(u) + n.length):  # O(1)
                    self.dicts.setDistance(v, (self.dicts.getNodeDist(u) + n.length))  # O(1)
                    self.dicts.setPrev(v, u)  # O(1)
                    H.decreaseKey(v)  # O(1) for array O(log|V|) for heap
        t2 = time.time()
        return (t2-t1)
```

2.

From now on in the write up |V| is equal to the number of nodes in the network.

Array Priority Queue Implementation:

Insert: The time complexity for my insert is O(1). This is because it only calls self.H.append(node) and when appending to an array it is constant time.

Delete-Min: The time complexity for my array delete-min is O(|V|) this is because it iterates over the entire array (for i in range(len(self.H)) using a for loop and then checks each node's (which is a key to the distance dictionary) distance (stored in x) to the minimum distance found so far (stored in d). It compares distances in constant time (stores the smallest values index in the array in index) and then pops off the smallest value on the array by calling pop, which is also constant.

Decrease-Key: The time complexity for my array decrease-key is O(1) because it does nothing by calling "pass" in the code which is constant.

Heap Priority Queue Implementation:

Insert: The time complexity for my insert is O(log|V|). First it finds the length of the queue (len(self.H)) which is used to see where the new node will be inserted (childIndex). Then it appends the new node to the array by calling append which is an O(1) and sets the nodes key-value pair in the dictionary which is also O(1). Then it finally calls bubbleUp. This moves the node from its current position up the array if its distance is less than its parent (parent node is found at childIndex – 1// 2). If the child's distance is less than the parent's it calls switch. Which switches in constant time using a temp variable (temp) to hold the parent node, and then switches the two and updates the two's key-value pairs in the dictionary to reflect the nodes new index. Bubble up can be called recursively up to log|V| times. This is because a balanced binary tree (which is what a heap is) has a height of log|V| and if the new node's distance is less than all of the nodes already in the queue it will travel up the entire height of the tree which is log|V| as said before. Thus, since insert calls bubble up, and bubble up can take O(log|V|) time insert takes this much time as well.

Delete-Min: The time complexity for my delete-min is O(log|V|). First it takes the first element of the array (self.H[0]) and is O(1) for look-up. Then it takes the last element of the array which is the rightmost node in the binary tree (self.H[len(self.H) -1) and sets it as the first element both are O(1) operations. It updates the new first node's index value as 0 in the dictionary(self.dicts.setindexDict(self.H[0], 0) also an O(1) operation. It then pops off the last array element since it is now at the beginning of the array by calling pop. Pop is also an O(1) operation. Then it calls bubble down. This moves the top node to its new position in the array. It grabs the nodes left and right child indices by doing (2 * parentIndex) + 1 and (2 * parentIndex) +2 respectively. It then checks to make sure the leftChildIndex is still in the array (leftChildIndex < len(self.H)) O(1). If it is it compares the parent's distance to the child's distance

(self.dicts.getNodeDist(self.H[parentIndex]) > self.dicts.getNodeDist(self.H[leftChildIndex]) O(1). If parent's distance is more than it, it then checks to see if there is a right child (rightChildIndex < len(self.H)) O(1). If there is a right child it compares the left and right child's distance to see whose is lower (self.dicts.getNodeDist(self.H[leftChildIndex]) <= self.dicts.getNodeDist(self.H[rightChildIndex]) O(1). If the left is lower than the right, switch is called on the parent and the left child (self.switch(parentIndex, leftChildIndex)) O(1) and then bubble down is called on the leftChild because this is now the parent. If right is smaller than the left then switch is called on parent and the right child (self.switch(parentIndex, rightChildIndex)) O(1) and then bubble down is called on the rightChild because it is now the parent. If the leftChildIndex was out of the array check to see if the rightChild is in the array and check if the parent is bigger than the right child's distance (self.dicts.getNodeDist(self.H[parentIndex]) > self.dicts.getNodeDist(self.H[rightChildIndex]) O(1) and if it is switch the parent and the right child (self.switch(parentIndex, rightChildIndex)) O(1). Then bubble down is called again. Because the array is a balanced binary tree (which is what a heap is) the height is log|V|. Thus the most recursive calls bubble down can have is log|V| because the node's distance could be larger than all of the other node's distances. This would cause the node to go from the top of the heap all the way to the bottom of the heap which takes as long as the heap's height which is log|V|. Thus, since delete-min calls bubble down and bubble down can take O(log|V|) time delete-min takes this much time as well.

Decrease-Key: The time complexity of my decrease-key is O(log|V|). First it grabs the index of the node whose distance changed (index = self.dicts.getNodeIndex(node)) O(1) and then calls bubble up on that node. This moves the node from its current position up the array if its distance is less than its parent (parent node is found at childIndex – 1// 2). If the child's distance is less than the parent's it calls switch. Which switches in constant time using a temp variable (temp) to hold the parent node and then switches the two and updates the two's key-value pairs in the dictionary to reflect the nodes new index. Bubble up can be called recursively up to log|V| times. This is because it is a balanced binary tree (which is what a heap is), and if the new node's distance is less than all of the nodes already in the queue it will travel up the entire height of the tree which is log|V| as said before. Thus, since decrease-key calls bubble up and bubble up can take O(log|V|) time decrease-key takes this much time as well.

3.

Time and Space Complexity for Array implementation:

computeShortestPaths: This has a time complexity of O(|V|^2) because the while loop runs the size of the H which is |V| and then it calls deleteMin() which iterates through the entire array to find the node with the smallest distance this runs the size of the array again, |V|. Thus, because deleteMin is inside the for loop it runs O(|V|^2). The loop that looks at all of the neighbors of the deletedMin node can

have at most 3 neighbors. Since the graph is thus sparse this for loop isn't ran for long and can just be added to the overall complexity, and since |E| = |V| cause of the predefined out-degree of each node the complexity of the while loop becomes O(|V^2| + |V|) but the |V| is dropped and the over all complexity of computeShortestPaths for an array queue is O(|V^2|). Space complexity is O(|V| + 3|V|). This is because an array queue of size |V| is made and 3 different dictionaries of size |V| are made. However addition can be ignored here and thus the total space complexity here is O(|V|).

getShortestPath: The time complexity here is O(|V|). This is because of the while loop that iterates over the length of the previous dictionary which can be as long as |V|. The loop that looks at all of the neighbors of the nextNode can have at most 3 neighbors. Since the graph is thus sparse this for loop isn't ran for long and can just be added to the overall complexity, and since |E| = |V| cause of the predefined out-degree of each node the complexity of the while loop becomes O(|V| + |V|) but the |V| is dropped and the overall complexity is thus just O(|V|). The space complexity is O(|V|) because we are making an array of path edges between nodes and the worst case would be to have all of the nodes a part of the path.

Class Dictionaries: All of the functions here are time complexity of an amortized O(1) since all inserts into a dictionary, setting a key-value in a dictionary, and getting the whole dictionary are all time averaged out to be O(1) for any kind of python dictionary. The space here though is O(3|V|) since there are three dictionaries of size |V|, the multiplicative constant can be dropped though to just give a space complexity of O(|V|).

DecreaseKey: The time complexity for my array decrease-key is O(1) because it does nothing by calling "pass" in the code which is constant. Space complexity is also O(1) since it is not storing anything.

MakeQueue: The time complexity is O(|V|) because it iterates through all of the nodes in the network which is size |V|. The space complexity is also O(|V|) since it eventually makes a new list of size |V| by calling insert |V| times.

Insert: The time complexity for my insert is O(1). This is because it only calls self.H.append(node) and when appending to an array it is constant time. Space complexity is O(|V|) because it eventually appends |V| things to the array.

DeleteMin: The time complexity for my array delete-min is O(|V|) this is because it iterates over the entire array (for i in range(len(self.H)) using a for loop and then checks each node's (which is a key to the distance dictionary) distance (stored in x) to the minimum distance found so far (stored in d). It compares distances in constant time (stores the smallest values index in the array in index) and then pops off the smallest value on the array by calling pop, which is also constant. The space complexity here is O(1) since we aren't storing any new information here.

getQueue: Time and space complexity are both O(1) here since it is just returning the queue and not adding to it.

Thus the overall time complexity when using an array is O(|V^2|) and has a space complexity of O(|V|) where V is the number of nodes in the network.

Time and Space Complexity for Heap implementation:

computeShortestPaths: This has a time complexity of O(|V|log|V|) because the while loop runs the size of the H which is |V| and then it calls deleteMin() which goes at most through the entire balanced binary tree which is the height of the tree, log|V|. It does this to find the node with the smallest distance this can run the height of the heap which is log|V|. Thus, because deleteMin is inside the for loop it runs O(|V|log|V|). The loop that looks at all of the neighbors of the deletedMin node can have at most 3 neighbors. Since the graph is thus sparse this for loop isn't ran for long and can just be added to the overall complexity, and since |E| = |V| cause of the predefined out-degree of each node the complexity of the while loop becomes O(|V|log|V| + |V|log|V|). The second |V|log|V| comes from the decreaseKey function which runs bubbleUp which goes at most through the entire balanced binary tree which is the height of the tree, log|V|. But it is dropped and the over all complexity of computeShortestPaths for a heap queue is O(|V|log|V|). Space complexity is O(|V| + 3|V|). This is because a heap queue of size |V| is made and 3 different dictionaries of size |V| are made. However addition can be ignored here and thus the total space complexity here is O(|V|).

getShortestPath: The time complexity here is O(|V|). This is because of the while loop that iterates over the length of the previous dictionary which can be as long as |V|. The loop that looks at all of the neighbors of the nextNode can have at most 3 neighbors. Since the graph is thus sparse this for loop isn't ran for long and can just be added to the overall complexity, and since |E| = |V| cause of the predefined out-degree of each node the complexity of the while loop becomes O(|V| + |V|) but the |V| is dropped and the overall complexity is thus just O(|V|). The space complexity is O(|V|) because we are making an array of path edges between nodes and the worst case would be to have all of the nodes a part of the path.

Class Dictionaries: All of the functions here are time complexity of an amortized O(1) since all inserts into a dictionary, setting a key-value in a dictionary, and getting the whole dictionary are all time averaged out to be O(1) for any kind of python dictionary. The space here though is O(3|V|) since there are three dictionaries of size |V|, the multiplicative constant can be dropped though to just give a space complexity of O(|V|).

Switch: The time complexity here is O(1). Because a switch happes in constant time between the parent node and the child node. And updating the nodes values in the index dictionary is also constant time. The space complexity here is O(1) since we are not taking up any new memory.

BubbleUp: The time complexity here is O(log|V|). This function moves the node from its current position up the array if its distance is less than its parent. If the child's distance is less than the parent's it calls switch. Bubble up can be called recursively up to log|V| times. This is because a balanced binary tree (which is what a heap is) has a height of log|V| and if the new node's distance is less than all of the nodes already in the queue it will travel up the entire height of the tree which is log|V| as said before. The space complexity is O(1) since we aren't adding or taking away from the heap, just moving things around.

BubbleDown: The time complexity here is O(log|V|). This function moves the top node to its new position in the array. It grabs the nodes left and right child indices. It then checks to make sure the leftChildIndex is still in the array and O(1) operation. If it is it compares the parent's distance to the child's distance O(1). If parent's distance is more than it, it then checks to see if there is a right child O(1). If there is a right child it compares the left and right child's distance to see whose is lower O(1). If the left is lower than the right, switch is called on the parent and the left child O(1), and then bubble down is called on the leftChild because this is now the parent. If right is smaller than the left then switch is called on parent and the right child O(1), and then bubble down is called on the rightChild because it is now the parent. If the leftChildIndex was out of the array check to see if the rightChild is in the array and check if the parent is bigger than the right child's distance O(1) and if it is switch the parent and the right child O(1). Then bubble down is called again. Because the array is a balanced binary tree (which is what a heap is) the height is log|V|. Thus the most recursive calls bubble down can have is log|V| because the node's distance could be larger than all of the other node's distances. This would cause the node to go from the top of the heap all the way to the bottom of the heap which takes as long as the heap's height which is log|V|.

getQueue: Time and space complexity are both O(1) here since it is just returning the queue and not adding to it.

Delete-Min: The time complexity here is O(log|V|) because it calls bubbleDown which as seen above was O(log|V|). The space complexity here is O(1) since we aren't storing any new information.

Insert: The time complexity here is O(log|V|) because it calls bubbleUp which as seen above was O(log|V|). The space complexity here is O(|V|) since that is the size of the heap array made.

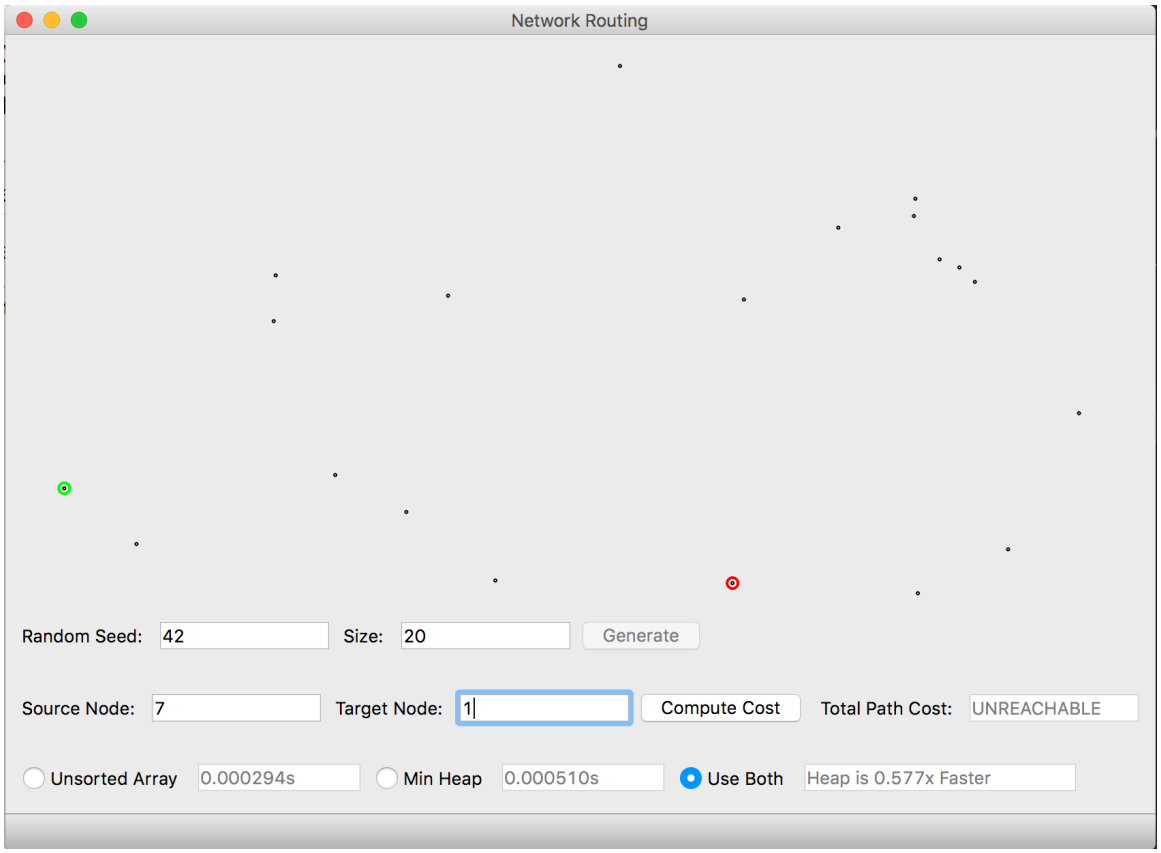
makeQueue: The time complexity here is O(|V|log|V|) this is because it runs a for loop through the size of the network nodes which is |V| and calls insert |V| times. Since insert is time complexity of O(log|V|) makeQueue is O(|V|log|V|). The space complexity is O(|V|) since we are making a new array of size |V|.

decreaseKey: The time complexity here is O(log|V|) this is because it calls bubbleUp which as seen above was O(log|V|). The space complexity here is O(1) since nothing is being added to the array.

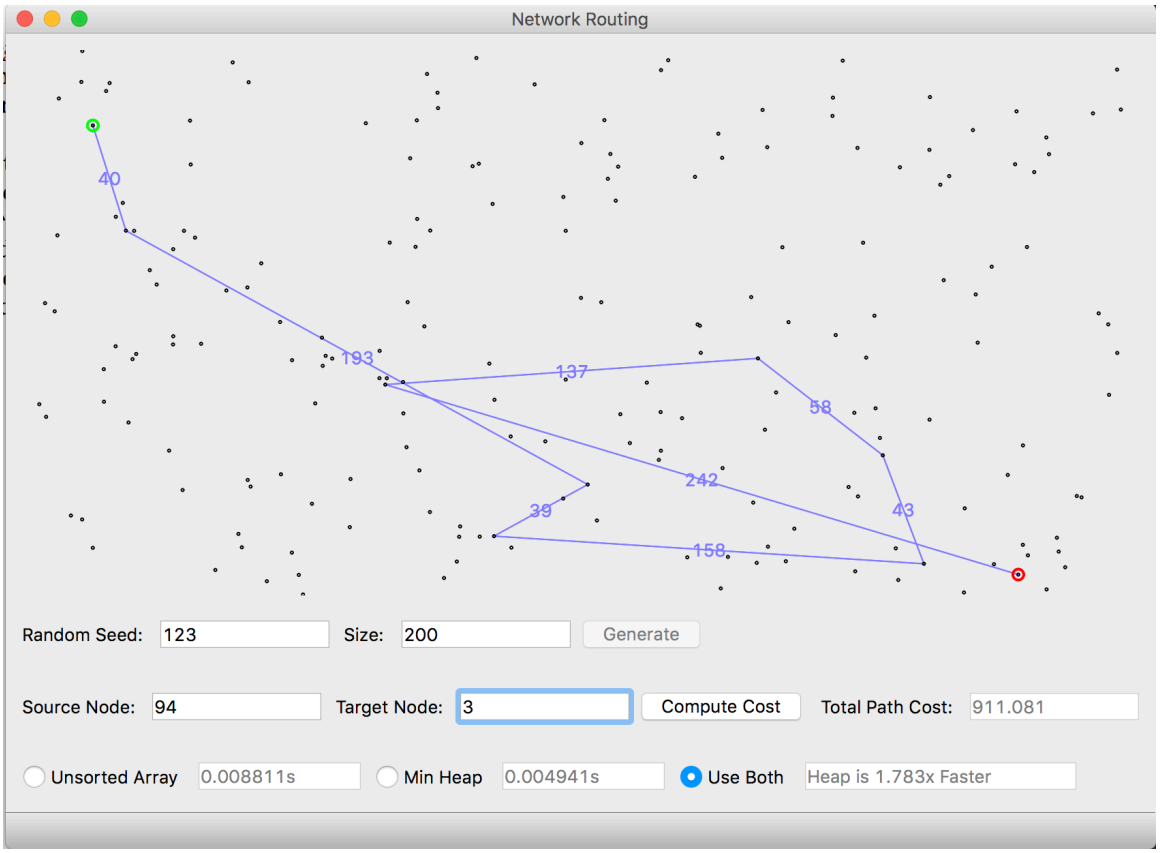
Thus the overall time complexity when using a heap is O(|V|log|V|) if the graph is sparse. Otherwise if the graph was dense the time complexity would become O(|V^2|log|V|) because there would be a lot of edges coming off of one node which would make |E| become equal to |V^2|. But since there are at most three edges for any node the graph is sparse. The space complexity is O(|V|) where V is the number of nodes in the network.

4.

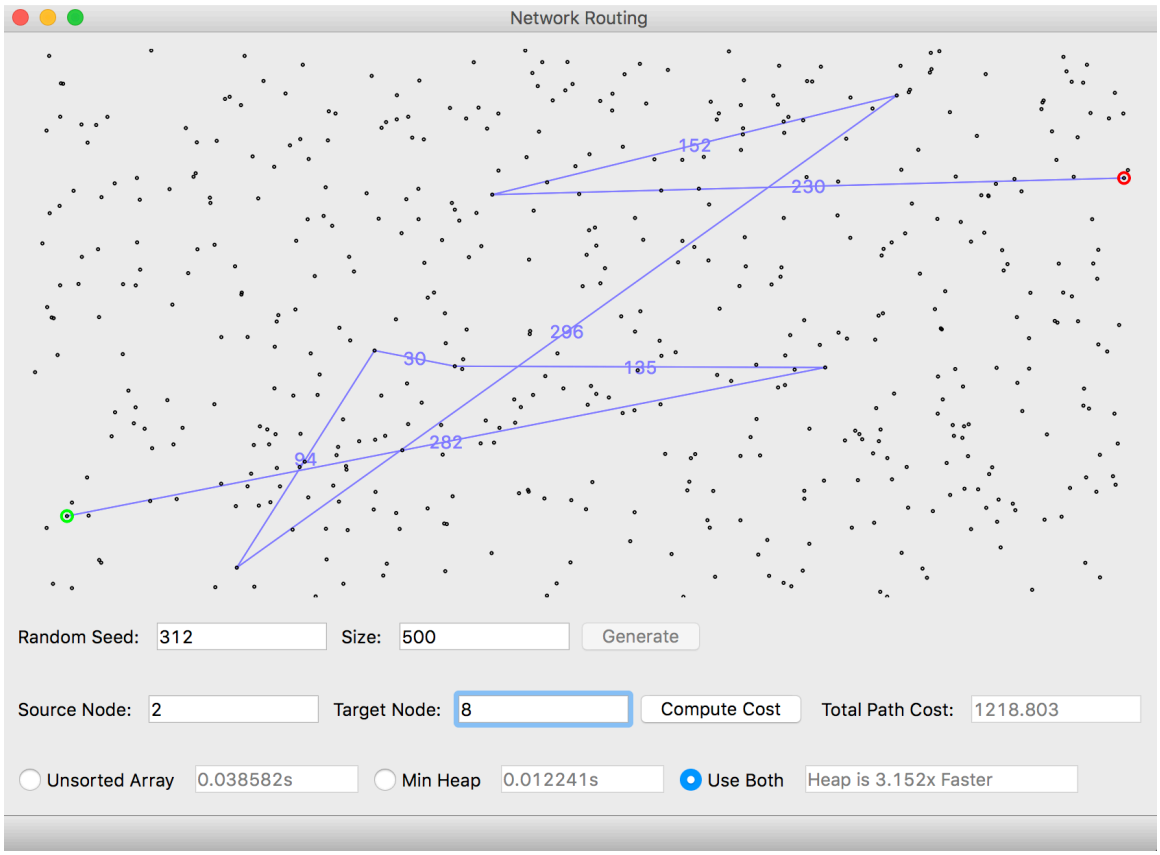
a.



b.


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
0.008811s
Min Heap
0.004941s
Use Both
Heap is 1.783x Faster

c.



5.

Size	Array Time (seconds)	Binary Heap Time (seconds)
100	0.002863	0.003051
100	0.003108	0.003131
100	0.003182	0.002981
100	0.003162	0.003037
100	0.002997	0.002972
Average Times	0.0030624	0.0030344
Difference: Array average / Binary Heap average	Heap is 1.009x Faster (0.000028 sec)	

Size	Array Time (seconds)	Binary Heap Time (seconds)
1000	0.155495	0.027644
1000	0.164046	0.028050
1000	0.159603	0.028024
1000	0.160405	0.028025
1000	0.161690	0.028673
Average Times	0.1602478	0.0280832

Difference: Array average / Binary Heap average	Heap is 5.706x Faster (.1321646 sec)

Size	Array Time (seconds)	Binary Heap Time (seconds)
10000	14.629159	0.426381
10000	14.847451	0.400289
10000	15.023647	0.404555
10000	15.445876	0.401548
10000	14.723485	0.401977
Average Times	14.933924	0.40695
Difference: Array average / Binary Heap average	Heap is 36.697x Faster (14.5269736 sec)	

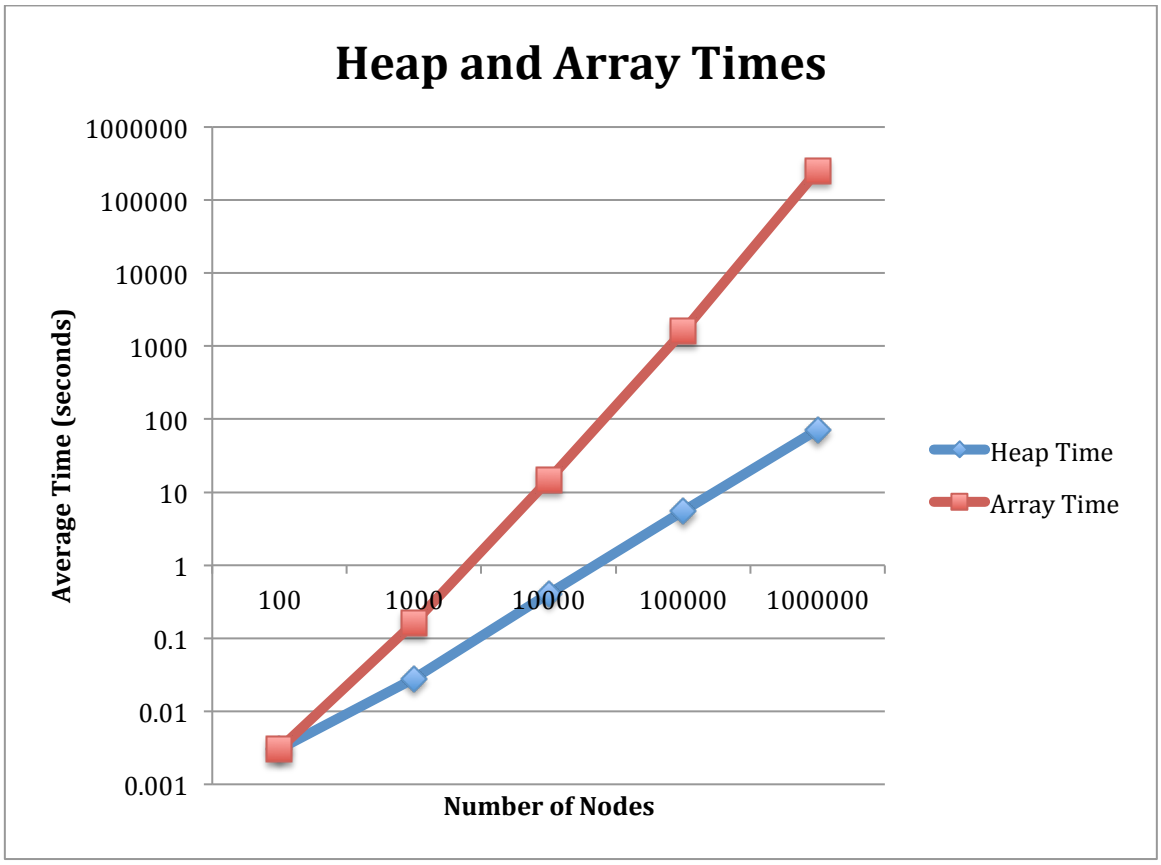
Size	Array Time (seconds)	Binary Heap Time (seconds)
100000	1412.163763	5.430976
100000	1512.826201	5.639656
100000	1740.849083	5.504946
100000	1671.608850	5.634989
100000	1662.709230	5.513027
Average Times	1600.031425	5.544719
Difference: Array average / Binary Heap average	Heap is 288.569x Faster (1594.486707 sec)	

Size	Binary Heap Time (seconds)
1000000	72.480937
1000000	72.276193
1000000	71.805835
1000000	71.774979
1000000	73.487324
Average Time	72.365054

Estimating Array for 1,000,000	How much longer it takes compared to last runtime	Average Array Time (seconds)
100	N/A	0.0030624
1000	52.33x longer than 100	0.1602478
10000	93.19x longer than 1000	14.933924
100000	107.14x longer than 10000	1600.031425
1000000	156.99x longer than 100000	251185.5752

My approximation for how long the array implementation for 1,000,000 nodes would take is 251,185.5752 seconds.

I was able to get this calculation by seeing how much of a difference between the average times was seen between powers of 10. The 10^2 takes 0.0030624 sec on average, where 10^3 takes 0.1602478 sec. To find the ratio of how much longer it takes between 10^2 and 10^3 one just takes 0.1602478 / 0.0030624 = 52.33x longer for 1000 nodes than for 100 nodes. Continuing in this manner we see that 14.933924 / 0.1602478 = 93.19x longer for 10000 nodes than for 1000, 1600.031425 / 14.933924 = 107.14x longer for 100000 nodes than for 10000. Then I looked at how much each ratio differed by. 93.19 / 52.33 = 1.7808 difference in the ratio between 10000 and 1000, and 107.14 / 93.19 = 1.1497 difference in the ratio between 100000 and 10000. Thus, to see how much the next power of 10's ratio differes from the last one I will take the average of the above numbers (1.7808 + 1.1487) / 2 = 1.46525 is how much of a factor increase happens when going up by one power of 10. So, to find the ratio for one million nodes I multiplied 1.46535 * 107.14 = 156.99x longer for 1000000 nodes than for 100000. After getting the ratio I multiplied 156.99 * 1600.031425 = 251,185.5752 seconds for the array implementation of 1,000,000 nodes.



This graph along with the raw data shows that the heap becomes faster and faster than the unsorted array implementation the more nodes/vertices are added to the

network. This makes sense because the overall Big-O for the heap implementation was O(|V|log|V|) and the array Big-O was O(|V^2|). Thus the time it takes for the heap is less than the time it takes for the array since |V^2| dominates |V|log|V|. This means that |V|log|V| grows slower than |V^2| and is seen in the graph as more nodes are added. At first the heap isn't much faster with such a small sized network (100) but going up a power of 10 (1000) we can start to see that the heap becomes faster. This makes sense since the array takes V^2 time. In other words it grows much more quickly the more nodes that are added to the network than the heap. However, we know that each node has an out-degree of 3. This means that the overall graph is quite sparse (not a lot of connections between one node and the rest of the nodes). But if the graph were to become dense (many connections between one node and the rest of the nodes) the Big-O of the heap would become O(|V^2|log|V|) (see reasoning under "Time and Space Complexity for Heap implementation" for why the heap becomes this when it is dense) which grows faster than O(|V^2|). Thus a dense graph would make the heap implementation take more time than the array. But, since this is to simulate a network routing table (ie the internet) which is sparse in real life the heap implementation would stay O(|V|log|V|) overall.