Project #3 - Network Routing

Code

```
class PQArray:
    def __init__(self, key):
        self.arrayQueue = []
        self.key = key

    # The time complexity for this function is O(V), where V is the number
    # of vertices that are being inserted in the list
    def makeQueue(self, nodes):
        for node in nodes:
            self.arrayQueue.append(node)

    # This function takes care of removing the node with the smallest distance value
from H.
    # The time complexity for this function is O(V), in which V is the
    # number of vertices, as when locating the smallest node can take up
    # to the total number of nodes in the worst case scenario
    def deleteMin(self):
        minIndex = 0
        key = self.arrayQueue[0].node_id
        for i in range(len(self.arrayQueue)):
            keyIndex = self.arrayQueue[i].node_id
            if self.key[key] > self.key[keyIndex]:
                minIndex = i
                key = keyIndex
        return self.arrayQueue.pop(minIndex)

    # This function runs in O(1) time, as it is pretty straightforward -
    # it's not doing anything for this implementation
    def decreaseKey(self, index):
        pass
    # This function runs in O(1) time to find the size of the queue
    def getSize(self):
        return len(self.arrayQueue)
```

```
class PQHeap:

    def __init__(self, key):
        self.heap = []
        self.distKey = key
        self.pointer = [] # pointers to keep track
        self.size = 0
    # The time complexity for this function is O(log(V)), mainly because the operation
to bubble up the tree
    # takes in O(log(V)).
    def insert(self, node):
        self.bubble_up(node)

    # The time complexity for this function is O(log(V)) - this operation is in charge
of locating and substituting the correct values of the given key. This follows a
similar pattern as the bubble_up operation, cutting the lookup in half.
    def decreaseKey(self, index):
        updatedIndex = self.pointer[index]
        while self.distKey[self.heap[updatedIndex].node_id] <
self.distKey[self.heap[int((updatedIndex - 1) / 2)].node_id]:
            self.heap[int((updatedIndex - 1) / 2)
                      ], self.heap[updatedIndex] = self.heap[updatedIndex],
self.heap[int((updatedIndex - 1) / 2)]
            self.pointer[self.heap[updatedIndex].node_id], self.pointer[
                self.heap[int((updatedIndex - 1) / 2)].node_id] =
self.pointer[self.heap[
                    int((updatedIndex - 1) / 2)].node_id],
self.pointer[self.heap[updatedIndex].node_id]
            updatedIndex = int((updatedIndex - 1) / 2)

    # This function runs in O(log(V)), mainly because it's calling the insert
operations V number of times and it implements
    # the bubble_up function, which runs in O(log(V))
    def makeQueue(self, nodes):
        for node in nodes:
            self.insert(node)

    # This functions runs in O(log(V)). The reason for that, is because it's only
requiring to check nodes that are above itself,
    # thus we never look for nodes that are below itself, cutting the lookup in
basically half.
```

```
    def bubble_up(self, node):
        self.heap.append(node)
        index = self.size
        self.pointer.append(index)

        while index != 1 and self.distKey[self.heap[index].node_id] <
self.distKey[self.heap[int((index-1)/2)].node_id]:
            self.heap[int(
                (index-1)/2)], self.heap[index] = self.heap[index],
self.heap[int((index-1)/2)]
            index = int((index-1)/2)
        self.size += 1
        self.pointer[node.node_id] = index

    # This function also runs in O(log(V)) time, for the same reasons as the
bubble_up, except rather
    # than checking above, it checks the nodes below it in order to move the high
value tuple down the min heap.
    def sift_down(self, i):

        leftChildIndex = self.distKey[self.heap[int(
            (i * 2) + 1)].node_id]
        rightChildIndex = self.distKey[self.heap[int(
            (i * 2) + 2)].node_id]

        if self.distKey[self.heap[i].node_id] > leftChildIndex or
self.distKey[self.heap[i].node_id] > rightChildIndex:
            if (leftChildIndex < rightChildIndex):  # bubble up smaller child
                self.pointer[self.heap[i].node_id], self.pointer[
                    self.heap[int((i * 2) + 1)].node_id] = self.pointer[self.heap[
                        int((i * 2) + 1)].node_id], self.pointer[self.heap[i].node_id]
                self.heap[i
                          ], self.heap[int((i * 2) + 1)] = self.heap[int((i * 2) +
1)], self.heap[i]
                i = int((i * 2) + 1)
            else:
                self.pointer[self.heap[i].node_id], self.pointer[
                    self.heap[int((i * 2) + 2)].node_id] = self.pointer[self.heap[
                        int((i * 2) + 2)].node_id], self.pointer[self.heap[i].node_id]
                self.heap[i
                          ], self.heap[int((i * 2) + 2)] = self.heap[int((i * 2) +
2)], self.heap[i]
```

```
                i = int((i * 2) + 2)

            return i

    # This function runs simply in O(1) because it is simply retrieving the child
given its indices
    def min_child(self, currIndex):
        leftChildIndex = self.distKey[self.heap[int(
            (currIndex * 2) + 1)].node_id]

        if self.distKey[self.heap[currIndex].node_id] > leftChildIndex:
            self.pointer[self.heap[currIndex].node_id], self.pointer[
                self.heap[int((currIndex * 2) + 1)].node_id] = self.pointer[self.heap[
                    int((currIndex * 2) + 1)].node_id],
self.pointer[self.heap[currIndex].node_id]
            self.heap[currIndex
                      ], self.heap[int((currIndex * 2) + 1)] =
self.heap[int((currIndex * 2) + 1)], self.heap[currIndex]
            currIndex = int((currIndex * 2) + 1)

            return currIndex

    # This function is running in O(log(V)) because it makes a call to sift_down,
which runs in O(log(V))
    def deleteMin(self):
        x = self.heap[0]
        self.heap[0] = self.heap[self.size-1]
        self.pointer[self.heap[0].node_id] = 0
        self.pointer[x.node_id] = None
        self.heap.pop()
        self.size -= 1
        prevIndex = 0

        while prevIndex != None and int((prevIndex*2) + 1) < self.size:
            if int((prevIndex*2) + 2) < self.size:
                prevIndex = self.sift_down(prevIndex)
            else:
                prevIndex = self.min_child(prevIndex)
        return x
    # This functions is just a helper function to get the size in O(1)
    def getSize(self):
        return self.size
```

```
from CS312Graph import *
import time
import math
from PQArray import *
from PQHeap import *


class NetworkRoutingSolver:
    def __init__(self):
        pass

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

    def getShortestPath(self, destIndex):
        self.dest = 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
        index = destIndex
        nodes = self.network.nodes[self.dest]
        edges_left = len(self.network.nodes)
        if self.prev[index] is None:
            print("UNREACHABLE")

        while self.prev[index] is not None:
            prevEdge = None
            node = self.prev[index]
            for i in node.neighbors:
                if i.dest.node_id == index:
                    prevEdge = i

            path_edges.append(
                (prevEdge.dest.loc, prevEdge.src.loc,
'{:.0f}'.format(prevEdge.length)))
            index = node.node_id

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

```
    def computeShortestPaths(self, srcIndex, 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)
        self.dijsktra(srcIndex, use_heap)
        t2 = time.time()
        return (t2-t1)

    # Heap PQ Implementation: The total runtime complexity for the Heap implementation
is
    # O((|V| + |E|)log(V)), because the operations itself is executed in O(V + E)
times, and because it
    # implements multiple operations in O(log(V)) time, we then multiply them both,
resulting
    # in O((|V| + |E|)log(V))
    # Array PQ Implementation: The total runtime complexity for the Array
implementation is
    # O(V^2), because the operations itself is executed in O(V + E) times, and because
it
    # implements multiple operations in O(V) time, we then multiply them both,
resulting
    # in O(V^2), resulting in a significantly slower operation.
    def dijsktra(self, start_node, use_heap):
        INFINITY = float('inf')  # try math.inf if too slow
        # initialize all the node distances to infinity
        self.dist = [INFINITY] * len(self.network.nodes)
        # initializes all the prev node to None
        self.prev = [None] * len(self.network.nodes)
        # initializes distance of starting node as 0
        self.dist[start_node] = 0
        if use_heap is False:
            # implement PQ class using unsorted array
            queue = PQArray(self.dist)
            queue.makeQueue(self.network.getNodes())
        else:
            # implement PQ class using heap
            queue = PQHeap(self.dist)
            queue.makeQueue(self.network.getNodes())
```

```
        # Implements the logic for Dijkstras which is the same, just different class
calls
        while queue.getSize() > 0:  # While H is not empty
            u = queue.deleteMin()
            for edge in u.neighbors:
                if self.dist[edge.dest.node_id] > self.dist[u.node_id] + edge.length:
                    self.dist[edge.dest.node_id] = self.dist[u.node_id] + edge.length
                    self.prev[edge.dest.node_id] = u
                    queue.decreaseKey(edge.dest.node_id)
```

Time-Complexity Analysis

The Heap PQ Implementation has a total runtime complexity of O((|V| + |E|)log(V)), because the operations itself is executed in O(V + E) times, and because it implements multiple operations in O(log(V)) time, we then multiply them both, resulting in O((|V| + |E|)log(V)).

The Array PQ Implementation has a total runtime complexity of O(V^2), because the operations itself is executed in O(V + E) times, and because it implements multiple operations in O(V) time, we then multiply them both, resulting in O(V^2), resulting in a significantly slower operation.

Unsorted Array Implementation Operations:

```
def makeQueue(self, nodes):
```

This function runs in O(V) because we are creating a new list and appending the nodes to it.

```
def deleteMin(self):
```

This function runs in O(V) as well, because V is the number of vertices we are searching through the entire list in order to locate the smallest value.

```
def decreaseKey(self, index):
```

O(1), it's not doing anything in the Array Implementation

```
def getSize(self):
```

O(1) as well, it's simply returning the current size of the queue.

Binary Implementation Operations:

```
def insert(self, node):
```

This functions runs in O(log(V)) because it simply calls bubble_up, which runs in O(log(V)

```
def decreaseKey(self, index):
```

This function runs in O(log(V), because once we change the key value we need to somehow put it back in the correct place. We search the entire heap, but we only search half of it, giving us a more efficient runtime.

```
def makeQueue(self, nodes):
```

This function runs in O(log(V) mainly because it's calling the insert operation, which already runs in O(log(V)).

```
def bubble_up(self, node):
```

This function runs in O(log(V). The reason for that is because it's only requiring to check nodes that are above itself, thus we never look for nodes that are below itself, basically cutting the lookup time in half. We keep doing this pair comparison until the values are in its correct position.

```
def sift_down(self, i):
```

Basically the same idea as bubble up, but instead looks at the nodes below it, causing it to have a runtime of O(log(V)).

```
def min_child(self, currIndex):
```

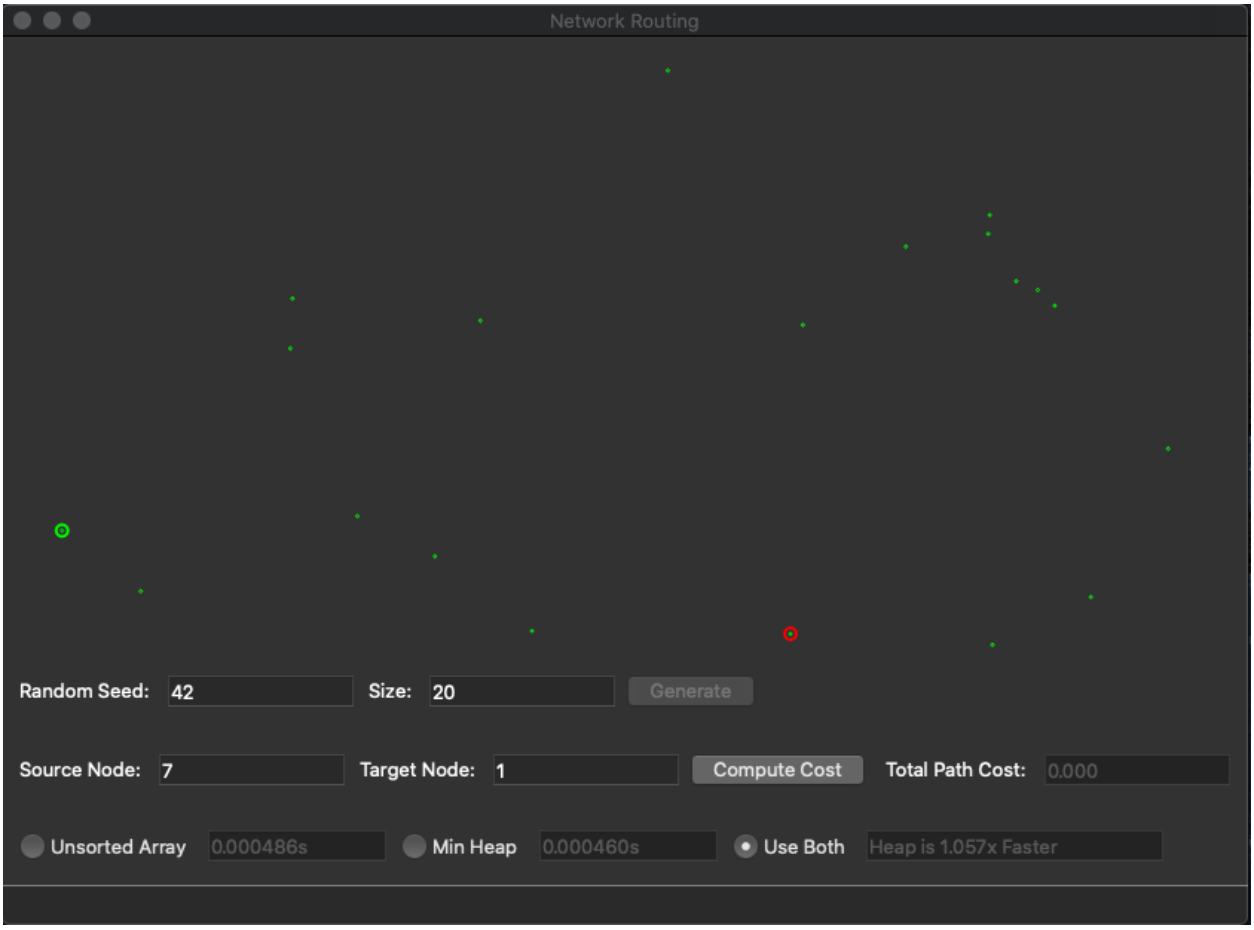
Simply O(1) as it's simply retrieving the index of the smallest child given currelndex.

```
def deleteMin(self):
```

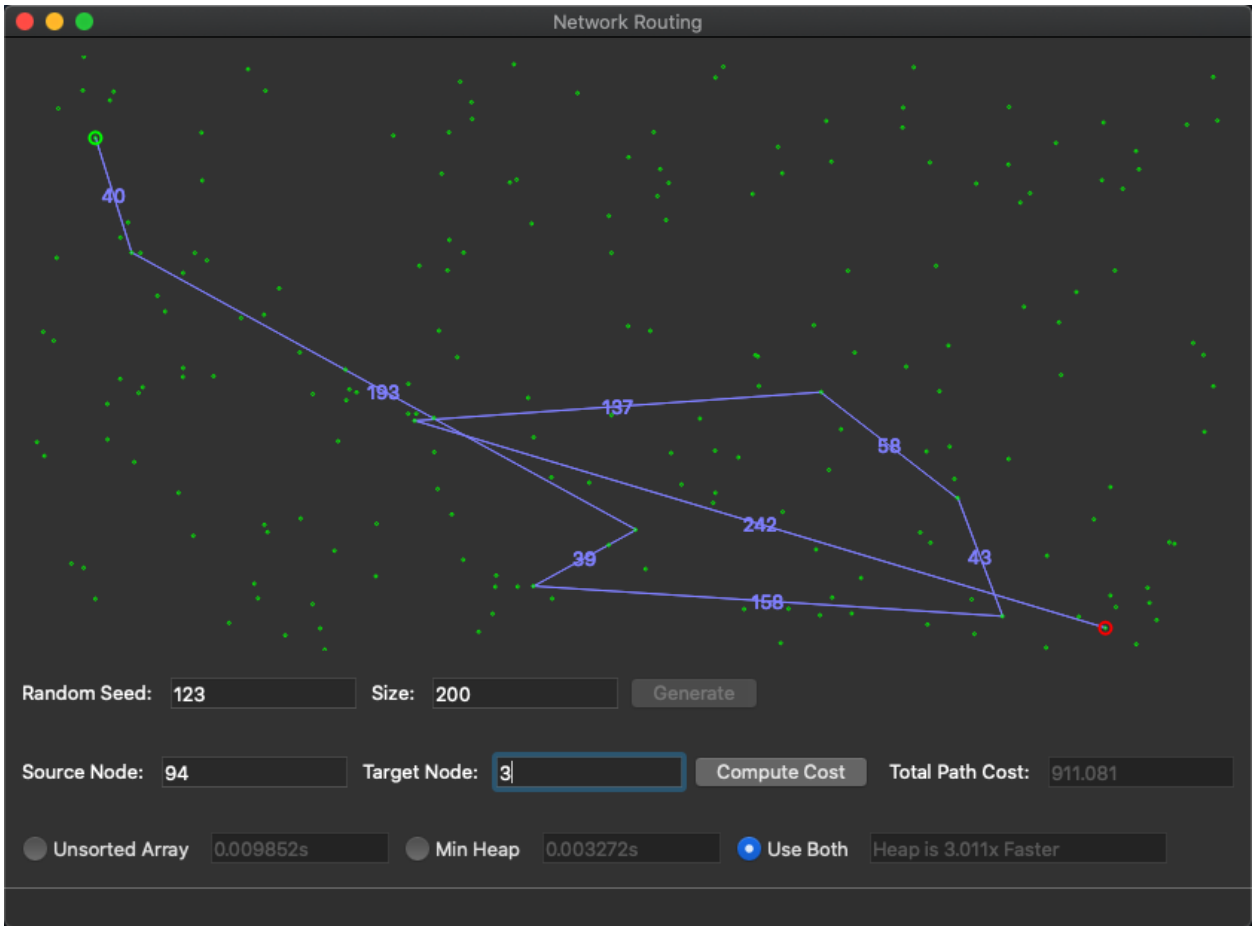
O(log(V) because it makes a call to sift_down, which runs in O(log(V).

Screenshots

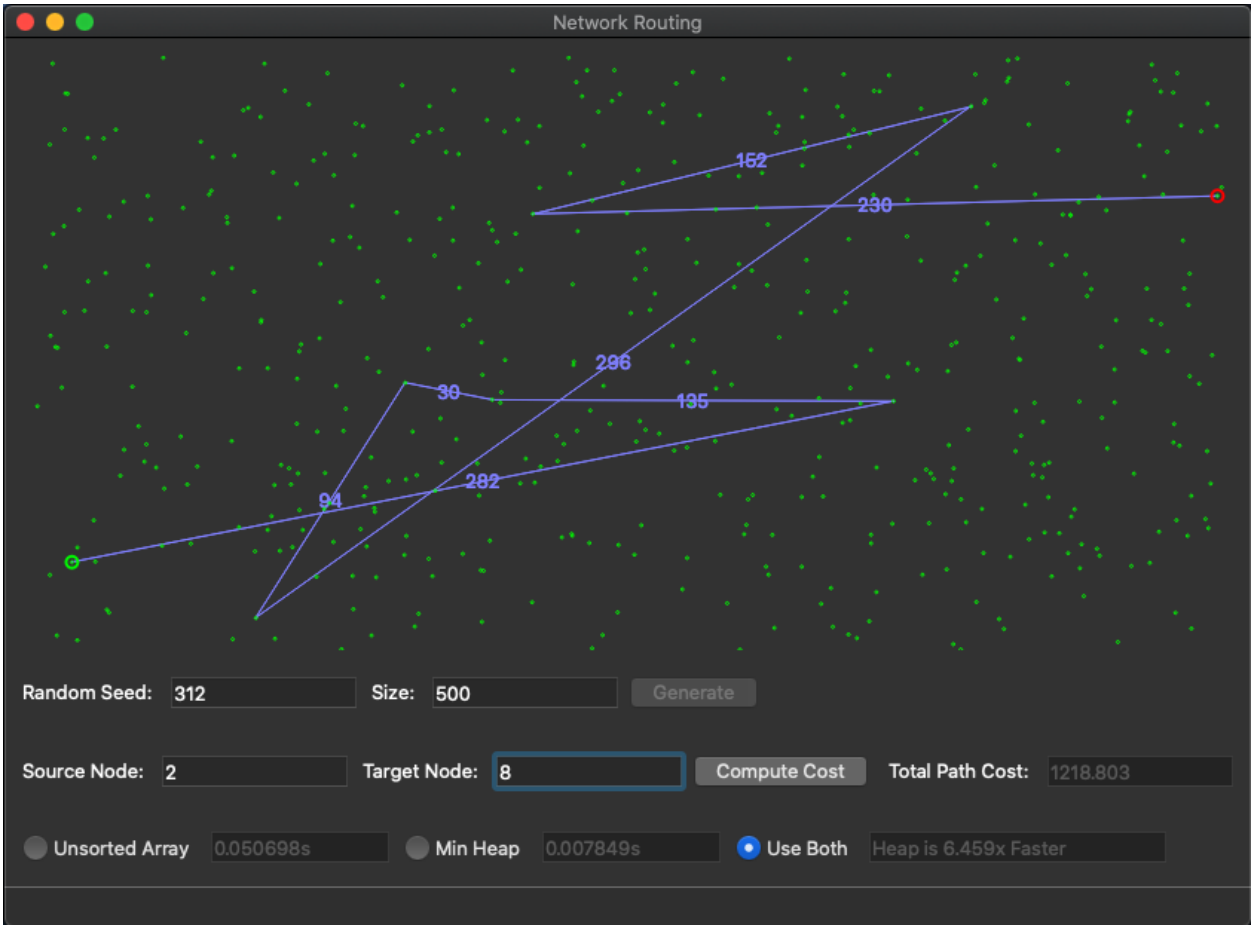
Random seed 42 - Size 20, use node 7 as the source and node 1 as the destination



Random seed 123 - Size 200, use node 94 as the source and node 3 as the destination

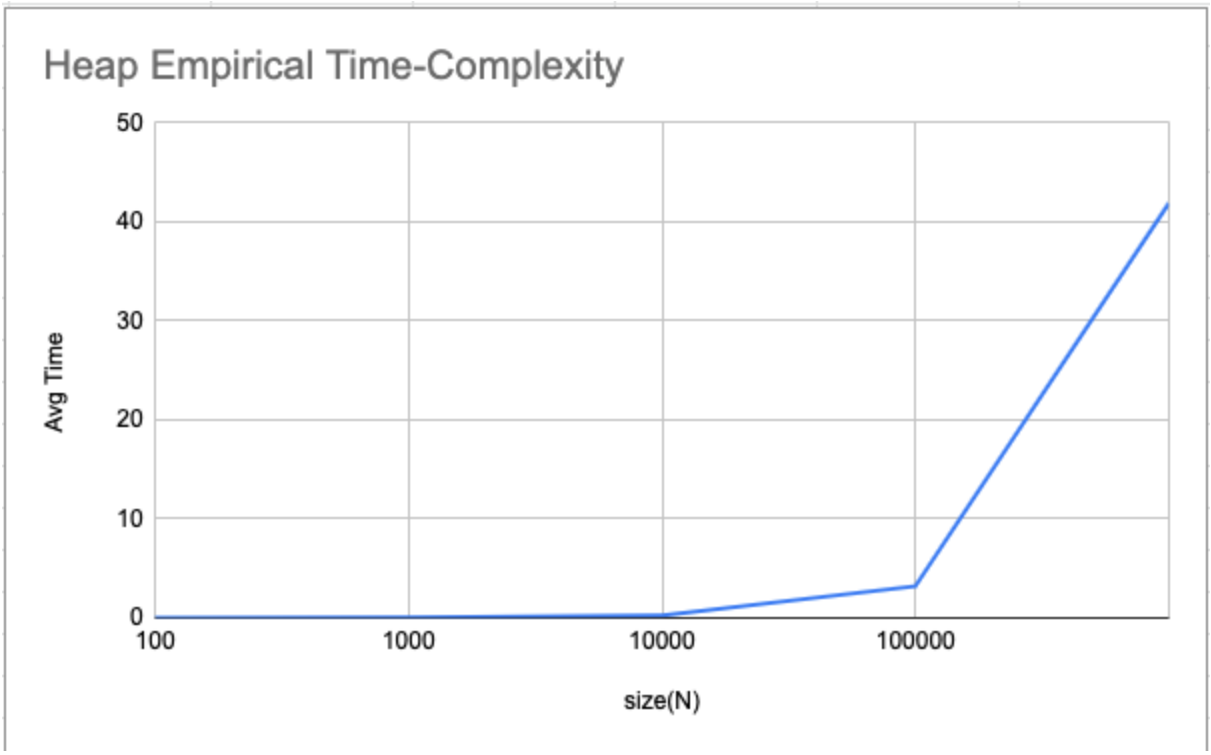


Random seed 312 - Size 500, use node 2 as the source and node 8 as the destination



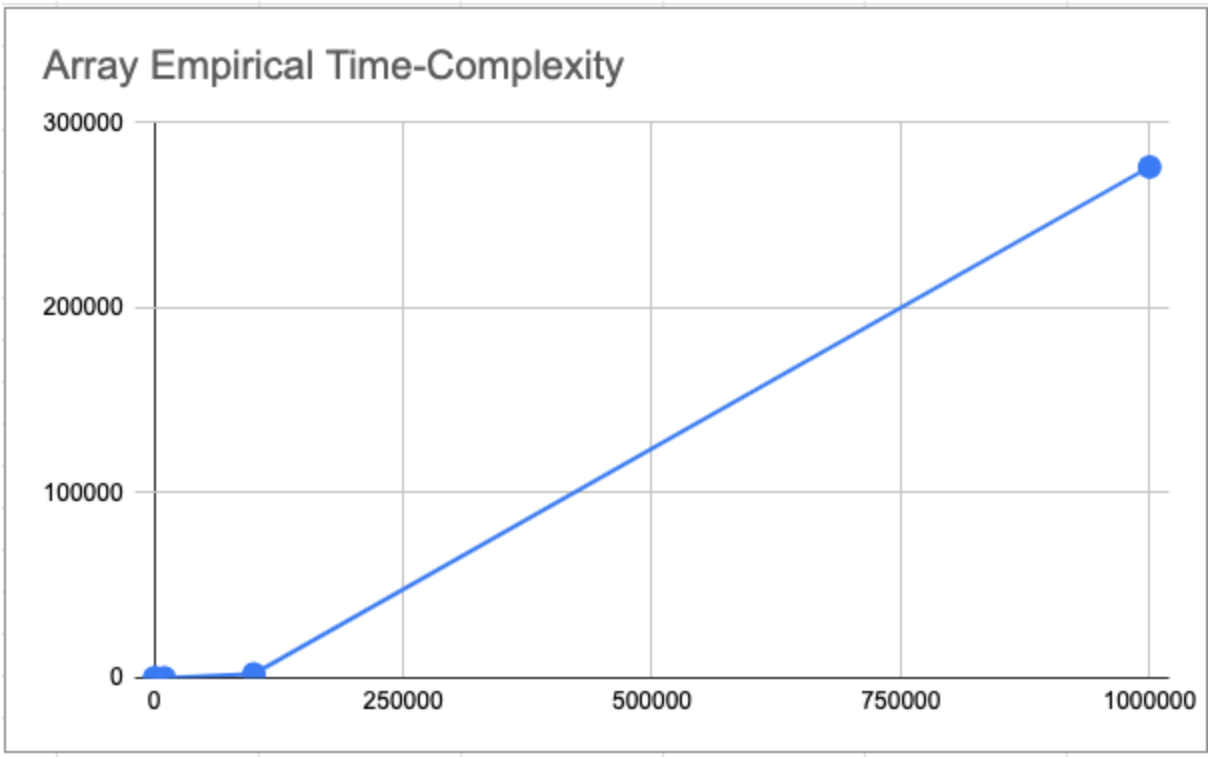
Empirical Evaluation

Heap						
size(N)	Set 1 Time	Set 2 Time	Set 3 Time	Set 4 Time	Set 5 Time	Avg Time
100	0.001725	0.001779	0.001746	0.001777	0.00182	0.0017694
1000	0.016241	0.016506	0.016684	0.016796	0.016056	0.0164566
10000	0.223322	0.221827	0.235477	0.234151	0.242306	0.2314166
100,000	3.0988	3.243202	3.393612	3.127931	3.119392	3.1965874
1,000,000	41.73737	41.533544	41.760854	42.386442	41.773803	41.8384026



Array:

size(N)	Set 1 Time	Set 2 Time	Set 3 Time	Set 4 Time	Set 5 Time	Avg Time
100	0.003301	0.003416	0.003303	0.003118	0.003396	0.0033068
1000	0.170253	0.172132	0.171848	0.17001	0.167056	0.1702598
10000	15.829941	15.802107	15.911927	15.828901	15.8496	15.8444952
100,000	2113.448274	2081.784565	2171.674452	2104.177298	2156.902107	2125.597339
1,000,000	275466.642	274121.3904	275621.6364	279750.5172	275707.0998	276133.4572



In order to estimate the time for the input of 1,000,000 nodes, I simply took the average results from the array table, and divided them from the heap table, and noticed that as the input grew, the heap implementation was faster by close to powers of 10. For example, my average time for the array implementation with 10,000 nodes was 15.84, and my heap implementation was 0.23. When I divide 15.84 by 0.23, I get around 68, meaning that my heap implementation was 68 times faster. If I get my implementation for 100,000 nodes, we see that my heap implementation was about 664 times faster. I took this observation, and simply made my array implementation for 1,000,000 roughly 6,600 times slower, resulting in an average of about 76 hours to run the array implementation, which seems about right once we take into account the overall time complexity of the project.

Thus, we can conclude the following based of our observations and based off the times we got from the empirical tests:

The Heap PQ Implementation has a total runtime complexity of O((|V| + |E|)log(V)), because the operations itself is executed in O(V + E) times, and because it implements multiple operations in O(log(V)) time, we then multiply them both, resulting in O((|V| + |E|)log(V)).

The Array PQ Implementation has a total runtime complexity of O(V^2), because the operations itself is executed in O(V + E) times, and because it implements multiple operations in O(V) time, we then multiply them both, resulting in O(V^2), resulting in a significantly slower operation.