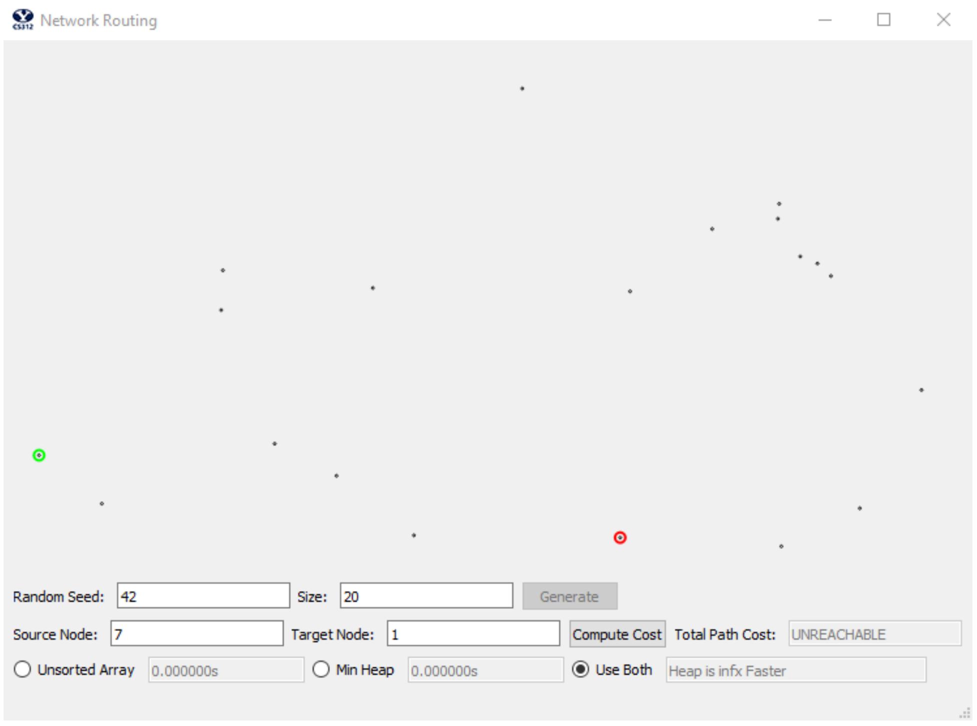
Matthew R. Christensen

CS 312 (001) – Dr Martinez, Tony R

February 25, 2021

Network Routing Algorithm Report

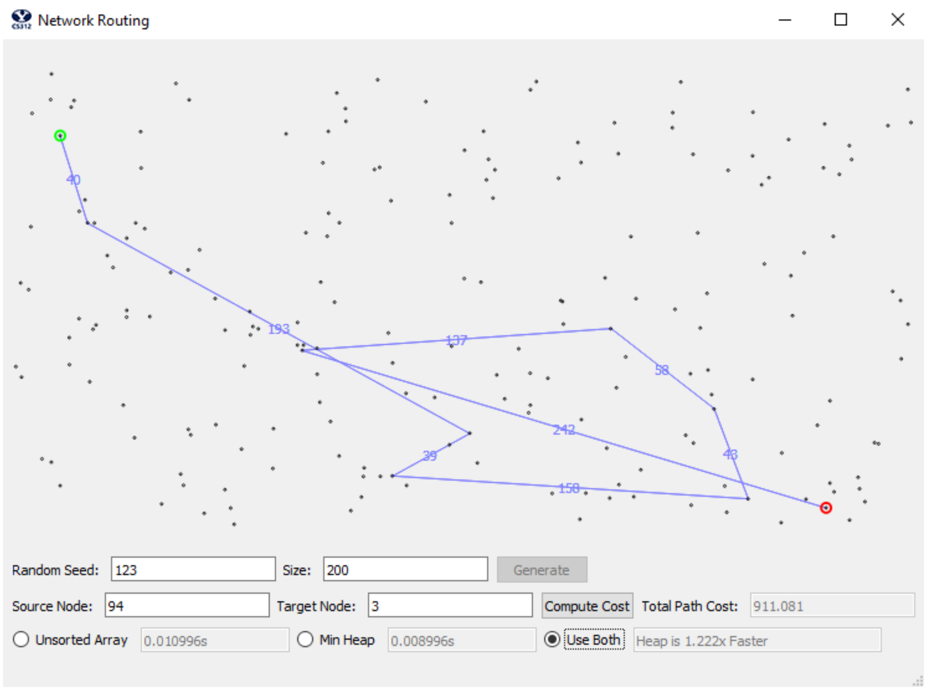
Screenshots



Random Seed: 42 Size: 20

Source Node: 7 Target Node: 1 Total Path Cost: UNREACHABLE

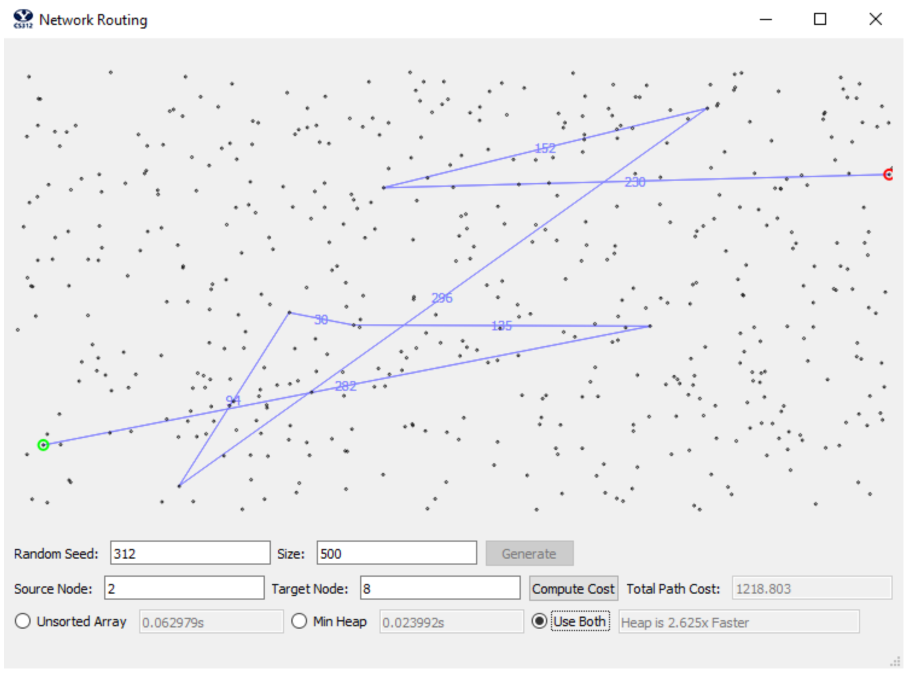
○ Unsorted Array 0.000000s ○ Min Heap 0.000000s ● Use Both Heap is infx Faster



Random Seed: 123 Size: 200 Generate

Source Node: 94 Target Node: 3 Compute Cost Total Path Cost: 911.081

Unsorted Array 0.010996s Min Heap 0.008996s [✓ Use Both] Heap is 1.222x Faster



Random Seed: 312 Size: 500 Generate

Source Node: 2 Target Node: 8 Compute Cost Total Path Cost: 1218.803

Unsorted Array 0.062979s Min Heap 0.023992s [✓ Use Both] Heap is 2.625x Faster

Time and Space Complexity

We first will talk about the time and space complexity of the program generally followed by a specific review of the two implementations using different data structures: the unsorted array implementation vs the heap implementation of the priority queue.

First, it is important to note that the theoretical time that we want to hit for the slack implementation is $O(n^2)$ while the goal of the heap implementation is $O((|E| + n) \log n)$. It can be noted that the binary heap implementation should be better than the array implementation when $|E| < n^2 / \log n$. Additionally, n here represents the number of vertices in the given graph problem. This $O(|V|)$ is equivalent to $O(n)$ for the purposes of this paper.

The algorithm used for graph exploration, in both implementations, is Dijkstra's Algorithm. The pseudo code is as follows:

```
H = makequeue(V)
While H is not empty:
    U = deletemin(H)
    For all edges (u, v) of in E:
        If dist(v) > dist(u) + l(u,v):
            dist(v) = dist(u) + l(u,v)
            prev(v) = u
            decreasekey(H,v)
```

Creating the queue, the delete minimum function, and the decrease key function all depend on their data structure for their respective time and space complexity. However, we know that the while loop with run at most n times as it must go through the entirety of the queue, and that each point with have at most 3 edges, for a O(n(E)) = O(3n)

However, the other part of this algorithm involves tracing the queue backwards to find and report the shortest path. This part of the algorithm is not dependent on the data structure used in implementation and thusly will be discussed in this section of the paper. The pseudo code is as follows:

```
current_node = final_node
while current_node is not none:
    previous_node = getPrevNode(current_node)
    for edge in previous_node.edges:
        if edge.dest is current_node:
            shortest_past.append(current_node)
    current_node = previous_node
```

We can see that in the worst case scenario we have the shortest path take every node, and thus have to visit and store n number of nodes, for a time and space complexity of O(n) (because the edges reduce down due to not being significant compared to O(n)).

Unsorted Array Implementation Analysis

Now we will analyze the unsorted array implementation of the algorithm. First, the create queue function:

```
For node in graph:
    Array.append(node)
```

We will have a time and space complexity of O(n), where n represents the number of vertices (|V|), because of the need to initially iterate and store all of the vertices in the queue.

Next, the delete minimum function:

```
For node in queue:
    If node.dist < min:
        min = node
return queue.pop(min)
```

Because the array is unsorted, we need to iterate through the entire array to guarantee that we found the smallest node distance. Thus our time complexity is O(n) while our space complexity is O(1), because of the lack of need to store anything meaningful. We will improve on this complexity in our heap due to storing nodes in an order that we can traverse to increase our time efficiency.

Next, we have the decrease function call. Because we don’t retain any sense of ordering within our array data structure implementation, we do not need to implement such a function (and thus skip this function for the array implementation).

This proves our case that the unsorted array should be of O(n^2). Because we know that getting the sortest path is of O(n), and that Dikstras algorithm is of O(3n(n + n)), we know that the totally order of complexity for the stack implementation is O(n + 3n(n + n)) = O(n + 3n(2n)) = O(n + 6n^2) = O(n^2)

Heap Priority Queue Implementation Analysis

Now we will analyze the algorithm theoretically with the heap priority queue implementation.

First, the create queue function:

```
For node in graph:
    Queue.set_node(node)
    Percolate_up(node)
```

Because we go through each node, and percolating is a function of O(log n) time complexity, we get a time complexity of O(n log n). While we only have to store a max of n node, for a space complexity of O(n)

Next, the delete minimum function:

//ran out of time, see code below

O(log n)

Next, we have the decrease function call.

//ran out of time, see code below

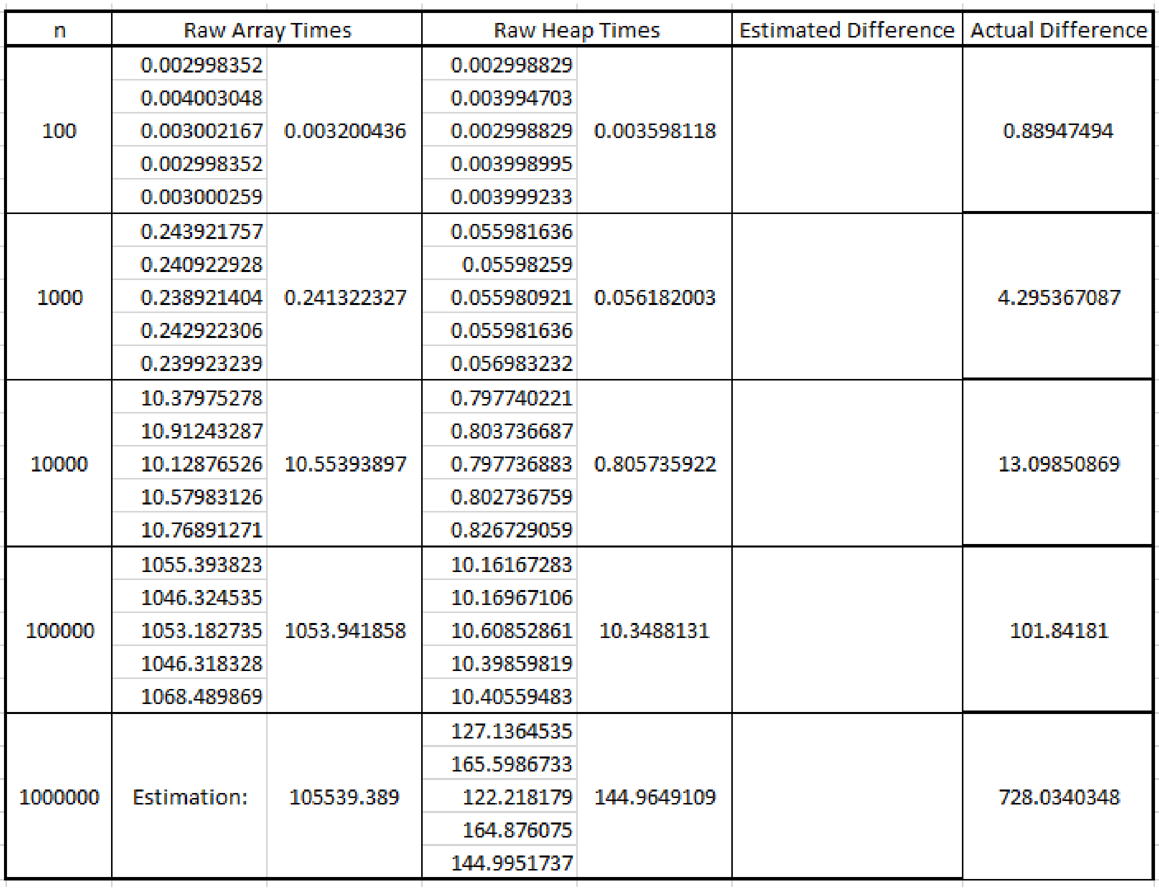
O(log n)

Thus we see that our heap implementation of the algorithm does indeed fit our theoretical complexity of O(n log n), as O(n + 3n(log n + log n + log n) = O(n + 3n(3log n) =

O(n + 6n log n) = O(n log n).

Empirical Algorithm Result Analysis

Now that we have defined the complexity of the algorithm, including both implementation of the priority queue (unsorted array and heap), we can compare what we expect to see against real world results. We let n be powers of 10, where *n* = {100, 1000, 10000, 100000, 1000000} indicates the number of points, or vertices, in the given execution. We then run our algorithm, using both the unsorted array and heap implementation of the priority queue, for every value of n to create the results seen below:



n	Raw Array Times		Raw Heap Times		Estimated Difference	Actual Difference
	0.002998352		0.002998829			
	0.004003048		0.003994703			
100	0.003002167	0.003200436	0.002998829	0.003598118		0.88947494
	0.002998352		0.003998995			
	0.003000259		0.003999233			
	0.243921757		0.055981636			
	0.240922928		0.05598259			
1000	0.238921404	0.241322327	0.055980921	0.056182003		4.295367087
	0.242922306		0.055981636			
	0.239923239		0.056983232			
	10.37975278		0.797740221			
	10.91243287		0.803736687			
10000	10.12876526	10.55393897	0.797736883	0.805735922		13.09850869
	10.57983126		0.802736759			
	10.76891271		0.826729059			
	1055.393823		10.16167283			
	1046.324535		10.16967106			
100000	1053.182735	1053.941858	10.60852861	10.3488131		101.84181
	1046.318328		10.39859819			
	1068.489869		10.40559483			
			127.1364535			
			165.5986733			
1000000	Estimation:	105539.389	122.218179	144.9649109		728.0340348
			164.876075			
			144.9951737			

Please note that the results for the unsorted array for the n value of 1,000,000 are estimated using a constant k. The constant k is solved for using the following equation:

$$Practical = k \times Theoretical$$

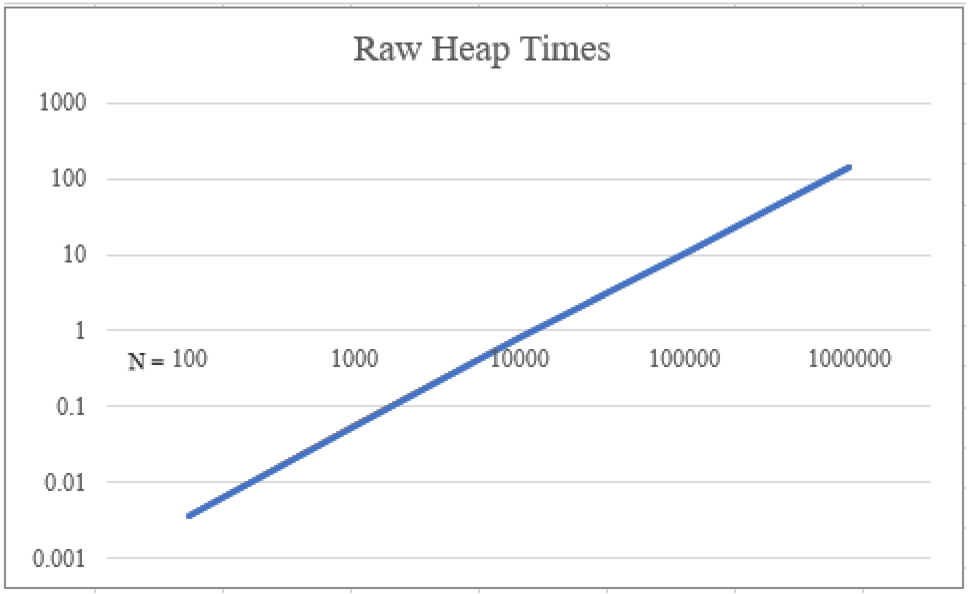
We can simply solve for *k* by finding what our real practical results are divided by the theoretical result of n^2. We can them simply solve for k, and by doing so find:

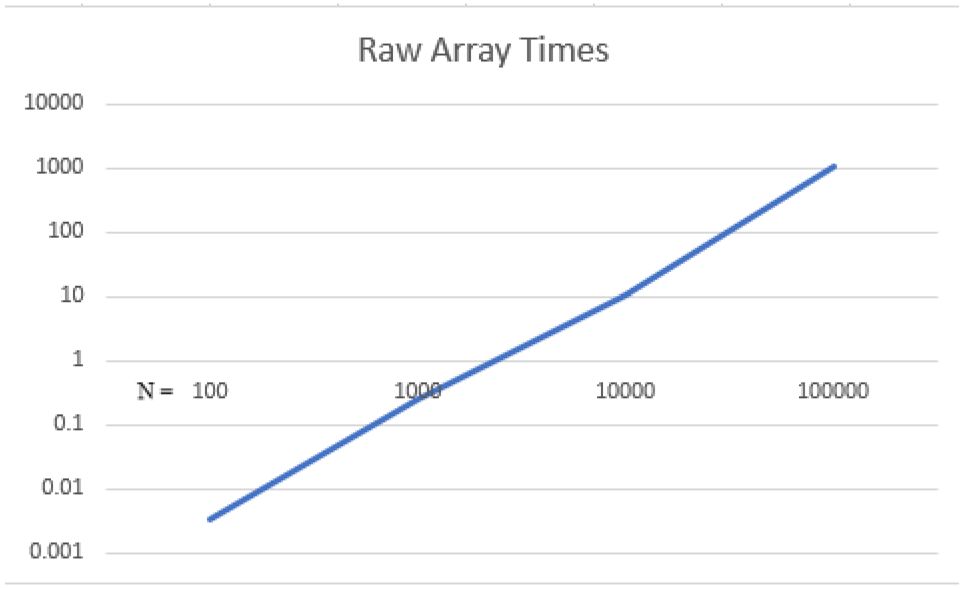
$$k = \frac{10.55393893897}{10000^2} \qquad = 1.05539389 \times 10^{-7}$$

This process gives us a *k* of 1.05539389E-07. By multiplying n^2 by our k value for an n value of 1,000,000 to achieve out estimate of 1055.39389 seconds:

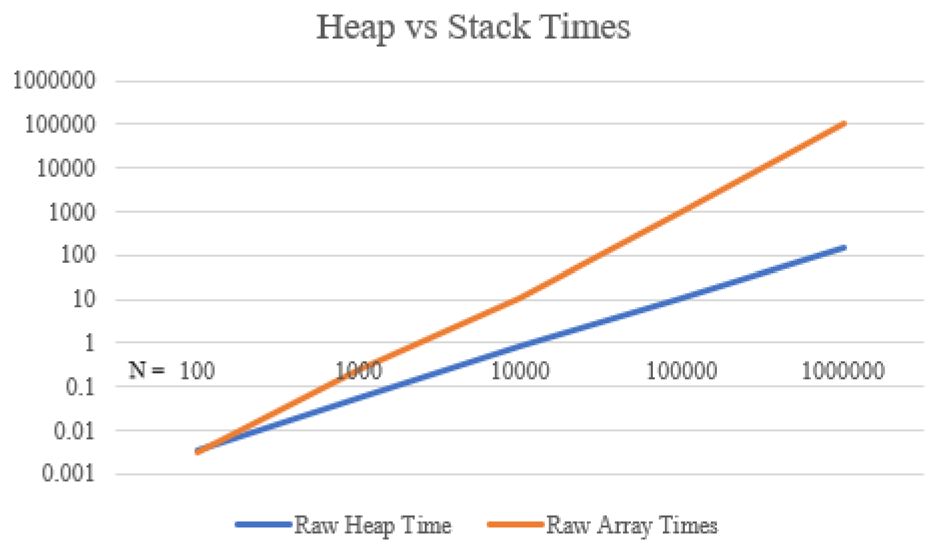
$$100000^2 \cdot .000000105539389 \qquad = 1055.39389$$

Plotting the times of the array and heap computation times, on a logarithmic scale, we see:





This confirms our theoretical analysis of O(n log n) and O(n^2) respectively because of the straight nature of the graphs. Plotting the results of the stack and heap implementations against each we can see the difference between the logarithmic and exponential grown:



Conclusion

We can thus conclude that our algorithm is correctly running at an approximate time complexity of n log n and space complexity for the heap implementation while the unsorted array implementation does indeed run at a time complexity of n^2. Furthermore, we have defined the constants k's that each algorithm runs in time with.

Appendix

```
from CS312Graph import *

import time

import math
```

```
class NetworkRoutingSolver:

    def __init__( self):

        pass



    def initializeNetwork( self, network ):

        assert( type(network) == CS312Graph )

        self.network = network

        self.results = {}



    def getShortestPath( self, destIndex ):

        print("getShortestPath")

        self.dest = destIndex

        path_edges = []

        total_length = 0

        node = self.network.nodes[self.dest]
        while self.results[node.node_id]["prev"] is not None:  # Traverse the graph backwards
            previous_node = self.network.nodes[self.results[node.node_id]['prev']]

            for neighbor in previous_node.neighbors:
                if neighbor.dest is node:
                    total_length = total_length + neighbor.length
                    path_edges.append((neighbor.src.loc, neighbor.dest.loc,
'{:.0f}'.format(neighbor.length)))

            node = previous_node

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

    def computeShortestPaths(self, src_index, use_heap=False):
        print("computeShortestPaths")
        t1 = time.time()

        if use_heap:
            queue = HeapPriorityQueue(self.network, src_index)
```

```
else:
    queue = UnsortedArrayPriorityQueue(self.network, src_index)


for node in self.network.nodes:
    self.results[node.node_id] = {'dist': math.inf, 'prev': None}


self.results[src_index]['dist'] = 0


print("Started queue")
while queue.is_not_empty():
    print("Queue length: ", len(queue))
    u = queue.delete_min()
    edges = self.network.nodes[u['id']].neighbors
    for edge in edges:
        v = self.results[edge.dest.node_id]
        # v2 = edge.dest
        if v['dist'] > u['dist'] + edge.length:
            v['dist'] = u['dist'] + edge.length
```

```
                v['prev'] = u['id']

                queue.decrease_key(edge.dest.node_id)

                queue.update_node(edge.dest.node_id, v["dist"])

        print("Finished queue")



        t2 = time.time()

        print(t2-t1)

        return t2-t1




class UnsortedArrayPriorityQueue:

    def __init__(self, graph, source_index):

        print("Start init for array pq")

        self.num_nodes = len(graph.nodes)

        self.queue = {}



        for index in range(self.num_nodes):

            if index == source_index:
```

```
            self.queue[graph.nodes[index].node_id] = {'dist': 0}

        else:

            self.queue[graph.nodes[index].node_id] = {'dist': math.inf}

    print("Finish init for array pq")



def delete_min(self):

    print("started delete")

    smallest_index = -1

    smallest_distance = math.inf



    for index, node in self.queue.items():

        if self.queue[index]['dist'] < smallest_distance:

            smallest_distance = self.queue[index]['dist']

            smallest_index = index

    smallest_node = {'id': smallest_index, 'dist': smallest_distance}

    if smallest_index is -1:

        first_node = self.queue.popitem()

        print("Finished delete")
        return {'id': first_node[0], 'dist': first_node[1]['dist']}

    del self.queue[smallest_index]

    print("Finished delete")

    return smallest_node



def update_node(self, index, distance):

    self.queue[index]['dist'] = distance



def is_not_empty(self):

    if len(self.queue) > 0:

        return True

    else:

        return False



def decrease_key(self, foo):

    pass
```

```
class HeapPriorityQueue:

    def __init__(self, graph, src_index):

        self.heap = []



        for node in graph.nodes:

            if node.node_id == src_index:

                self.insert_node(node.node_id, 0)

            else:

                self.insert_node(node.node_id, math.inf)





    def __len__(self):

        return len(self.heap) - 1



    def insert_node(self, node_id, distance):

        print("started insert")

        self.heap.append({'id': node_id, 'dist': distance})

        self.percolate_up(len(self))
    print("Finished insert")

def delete_min(self):
    print("Started delete min")
    return_node = self.heap[0]
    self.heap[0] = self.heap[len(self)]
    self.heap.pop()
    self.percolate_down(0)
    print("Ended delete min")
    return return_node

def decrease_key(self, node_id):
    self.percolate_up(node_id)

def percolate_up(self, index):
    if index is 0:
        return
    parent_index = index // 2

    if self.heap[parent_index]['dist'] > self.heap[index]['dist']:

        self.swap_node(index, parent_index)

        self.percolate_up(parent_index)



def percolate_down(self, parent_index):

    print("started percolate_down")

    if parent_index is len(self):

        return



    while parent_index * 2 <= len(self):

        mc = self.min_child(parent_index)

        if self.heap[parent_index]["dist"] > self.heap[mc]["dist"]:

            self.swap_node(mc, parent_index)

        parent_index = mc

    print("finished percolate_down")



def min_child(self, index):
    print("started min_child")
    if index * 2 + 1 > len(self):
        print("finished min_child")
        return index * 2

    if self.heap[index * 2]['dist'] < self.heap[index * 2 + 1]["dist"]:
        print("finished min_child")
        return index * 2

    print("finished min_child")
    return index * 2 + 1

def update_node(self, node_id, distance):  # todo check to see if this is being used
    for node in self.heap:
        if node['id'] is node_id:
            node["dist"] = distance
            break
```

```
def swap_node(self, index_1, index_2):

    node = self.heap[index_1]

    self.heap[index_1] = self.heap[index_2]

    self.heap[index_2] = node



def is_not_empty(self):

    if len(self) > 0:

        return True

    else:

        return False
```