Cody Kesler
CS 312 – 2
Lab 3

Code:

```
from CS312Graph import *
import time
import sys

class NetworkRoutingSolver:
    def __init__( self, display ):
        pass

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

// O(n) – worst case. This loops through one path of nodes(could visit each node)
    def getShortestPath( self, dest_index ):
        self.dest = dest_index
        path_edges = []
        total_length = 0
        self.cur_node = self.queue.nodes[self.dest]
        while(self.cur_node.node_id != self.source):
            cur_loc = self.cur_node.loc
            prev = self.queue.get_prev_node(self.cur_node)
            if(prev != None):
                prev_loc = prev.loc
            else:
                break
            length = self.queue.get_prev_length(self.cur_node)
            path_edges.append((cur_loc, prev_loc, '{:.0f}'.format(length)))
            self.cur_node = self.queue.get_prev_node(self.cur_node)
            total_length += length

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

// Thus total is O((n + e)*log(n))
    def computeShortestPaths( self, srcIndex, use_heap=False ):
        self.source = srcIndex
```

Cody Kesler
CS 312 – 2
Lab 3

```
    t1 = time.time()
    if(use_heap):
                //O(n)
        self.queue = Heap(self.network, srcIndex)
    else:
                //O(n)
        self.queue = Array(self.network, srcIndex)



  //Repeats edges times
    while(not self.queue.queue_empty()):
        lowest_node = self.queue.delete_min()
        if(lowest_node == -1):
            break

        if(lowest_node.node_id != 0):
                        // O(n) Ends up being n since reaches total for all the nodes
            self.queue.push_neighbors_on_queue(lowest_node)

        neighbors = lowest_node.neighbors








                // O(E) since it visits all the edges for each nodes
        for i in range(0, len(neighbors)):
            if(self.queue.get_distance_to(neighbors[i].dest) >
                        self.queue.get_distance_to(lowest_node) + neighbors[i].length):
                self.queue.set_distance_to(neighbors[i].dest,
                        self.queue.get_distance_to(lowest_node) + neighbors[i].length)
                self.queue.set_prev_node(neighbors[i].dest, lowest_node)
                self.queue.decrease_key()
```

Cody Kesler
CS 312 – 2
Lab 3

```
        t2 = time.time()


        return (t2-t1)
class Heap:
//O(n) This only calls insert and makes lists the size of the nodes in the graph
    def __init__(self, graph, src):
        def __init__(self, graph, src):
        self.nodes = graph.nodes
        self.queue = list()
        self.queue_size = 0
        self.node_dist = [sys.maxsize-1]*len(self.nodes)
        self.prev_node = [None]*len(self.nodes)
        self.popped_nodes = list()

        self.insert(CS312GraphEdge(self.nodes[src], self.nodes[src], 0))
        self.node_dist[src] = 0
        self.prev_node[src] = self.nodes[src]
        self.push_neighbors_on_queue(self.nodes[src])


                pass

//O(log(n)) This function gets called on just the 3 neighbors of the function and calls sift up for each of them
    def push_neighbors_on_queue(self, node):
        for i in range(len(node.neighbors)):
            if(node.neighbors[i].dest.node_id not in self.popped_nodes):
                self.insert(node.neighbors[i])
                self.sift_up(self.queue_size-1)
//O(1)
    def insert(self, edge):
        self.queue.append(edge)
        self.queue_size = self.queue_size + 1

//O(Log(n) It repeats the height of the tree which is log(n)
    def sift_up(self, i):
        parent = (i-1) // 2
        while i != 0 and self.node_dist[self.queue[i].dest.node_id] <
```

```
                              self.node_dist[self.queue[parent].dest.node_id]:
            temp = self.queue[parent]
            self.queue[parent] = self.queue[i]
            self.queue[i] = temp
            i = parent
            parent = (i - 1) // 2

//O(Log(n) It uses the sift down function which is Log(n)
    def delete_min(self):
        if (self.queue_empty()):
            return -1

        return_node = self.queue[0].dest
        self.queue[0] = self.queue[-1]
        self.queue.pop()
        self.popped_nodes.append(return_node.node_id)
        self.queue_size = self.queue_size - 1
        self.sift_down(0)

        return return_node


// O(log(n) This is Log(n) as it starts as the vertex of the tree and only visits its children until it hits the bottom which is Log(n) levels thus O(log(n)
def sift_down(self, i):
        if(not self.queue_empty()):
            while (i * 2) < len(self.queue):
                min_child = self.min_child(i)
                if(min_child == -1):
                    break
                if (self.node_dist[self.queue[i].dest.node_id] >
                                        self.node_dist[self.queue[min_child].dest.node_id]):
                    temp = self.queue[i]
                    self.queue[i] = self.queue[min_child]
                    self.queue[min_child] = temp
```

Cody Kesler
CS 312 – 2
Lab 3

```
            i = min_child

//This function just looks at the two children and returns its lowest value child. O(1)
    def min_child(self, i):
        left_child = i * 2 + 1
        right_child = i * 2 + 2
        if left_child >= len(self.queue):
            return -1
        elif right_child >= len(self.queue):
            return left_child
        else:
            if self.node_dist[self.queue[left_child].dest.node_id] <
                            self.node_dist[self.queue[right_child].dest.node_id]:
                return left_child
            else:
                return right_child

//O(1)
    def queue_empty(self):
        if (len(self.queue) > 0):
            return False
        else:
            return True
//O(1)
    def get_distance_to(self, node):
        return self.node_dist[node.node_id]
//O(1)
    def get_prev_length(self, node):
        if (self.prev_node[node.node_id] != None):
            return self.node_dist[node.node_id] –
                            self.node_dist[self.prev_node[node.node_id]]
        else:
            return 0
//O(1)
    def get_prev_node(self, node):
        if (self.prev_node[node.node_id] != None):
```

Cody Kesler
CS 312 – 2
Lab 3

```
            return self.nodes[self.prev_node[node.node_id]]
//O(1)
    def set_distance_to(self, node, distance):
        self.node_dist[node.node_id] = distance
//O(1)
    def set_prev_node(self, node, prev_node):
        self.prev_node[node.node_id] = prev_node.node_id
//O(1)
    def decrease_key(self):
        return True

//O(1)
class Array:
//O(n) This only calls insert and makes lists the size of the nodes in the graph
    def __init__(self, graph, src):
        self.graph = graph
        self.nodes = graph.getNodes()
        self.node_dist = list()
        self.prev_node = list()
        self.popped_nodes = list()
        self.queue = list()
        nodes = graph.getNodes()

// This for loop is O(n) as it repeats for the number of nodes
        for i in range(len(nodes)):
            if(nodes[i].node_id == src):
                self.queue.append(CS312GraphEdge(nodes[i], nodes[i], 0))
                self.push_neighbors_on_queue(nodes[i])
                self.node_dist.append(0)
                self.prev_node.append(src)
            else:
                self.node_dist.append(sys.maxsize-1)
                self.prev_node.append(None)

// This is O(n) total
    def push_neighbors_on_queue(self, node):
```

Cody Kesler
CS 312 – 2
Lab 3

```
        for i in range(0,len(node.neighbors)):
            if(node.neighbors[i].dest.node_id not in self.popped_nodes):
                self.queue.append(node.neighbors[i])

// O(1)
    def queue_empty(self):
        if(len(self.queue) > 0):
            return False
        else:
            return True

// O(
    def delete_min(self):
        temp_highest = sys.maxsize
        node_index = -1
        queue_index = -1
        for i in range(len(self.queue)):
            if(self.node_dist[self.queue[i].dest.node_id] < temp_highest
               and self.queue[i].dest.node_id not in self.popped_nodes):
                temp_highest = self.node_dist[self.queue[i].dest.node_id]
                node_index = self.queue[i].dest.node_id
                queue_index = i

        if(node_index == -1):
            return -1
        else:
            return_node = self.queue[queue_index].dest
            self.queue.pop(queue_index)
            self.popped_nodes.append(return_node.node_id)
            return return_node

    def get_distance_to(self, node):
        return self.node_dist[node.node_id]

    def set_distance_to(self, node, distance):
        self.node_dist[node.node_id] = distance
```

Cody Kesler
CS 312 – 2
Lab 3

```
def get_prev_node(self, node):
    if (self.prev_node[node.node_id] != None):
        return self.nodes[self.prev_node[node.node_id]]


def set_prev_node(self, node, prev_node):
    self.prev_node[node.node_id] = prev_node.node_id

def get_prev_length(self, node):
    if (self.prev_node[node.node_id] != None):
        return self.node_dist[node.node_id] - self.node_dist[self.prev_node[node.node_id]]
    else:
        return 0

def decrease_key(self):
    return True
```

Complexity Analysis:

Array: Making the queue is (n). Inserting is O(1). Deleting the Min is O(n). Decreasing the Key doesn't do much
Thus the total time complexity is n + n + n(n) + e(1) = O(n^2 + e)

Heap: Making the queue is n. Inserting comes to log(n). Deleting in is log(n). Decreasing key runs at log(n)
Thus the total time complexity is n + log(n) + n*log(n) + elog(n) = O((n + e)*log(n))

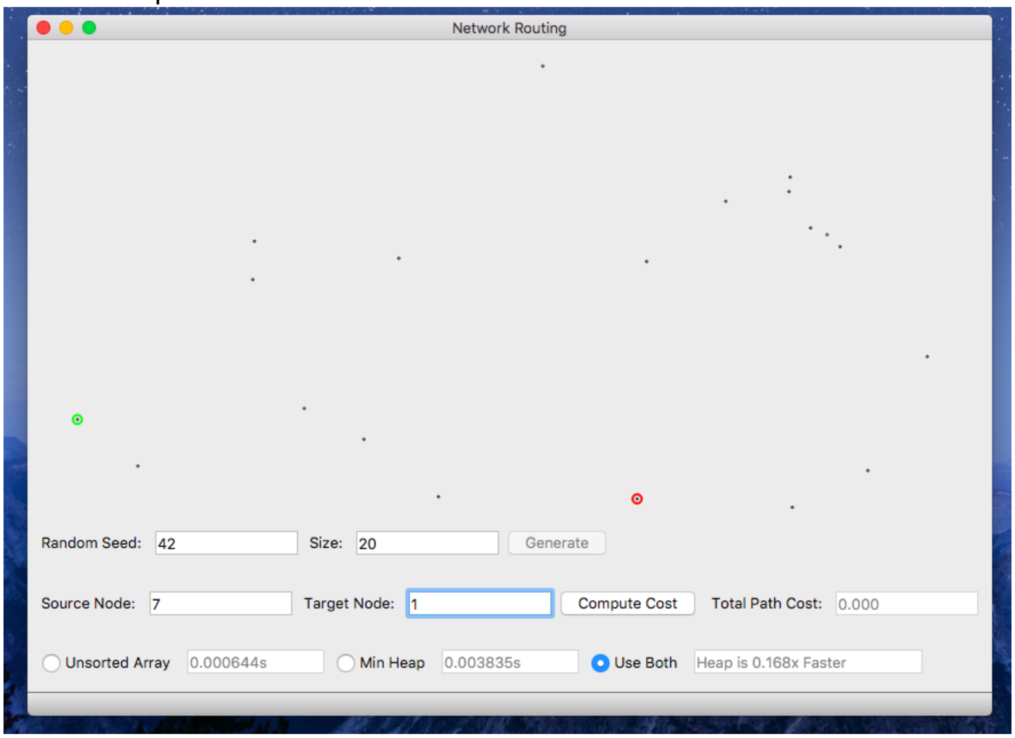
Drawing the shortest path yields O(n) added to each of the implementations.
So it is either
O((n + e)*log(n) + n)   Or   O(n^2 + e + n)

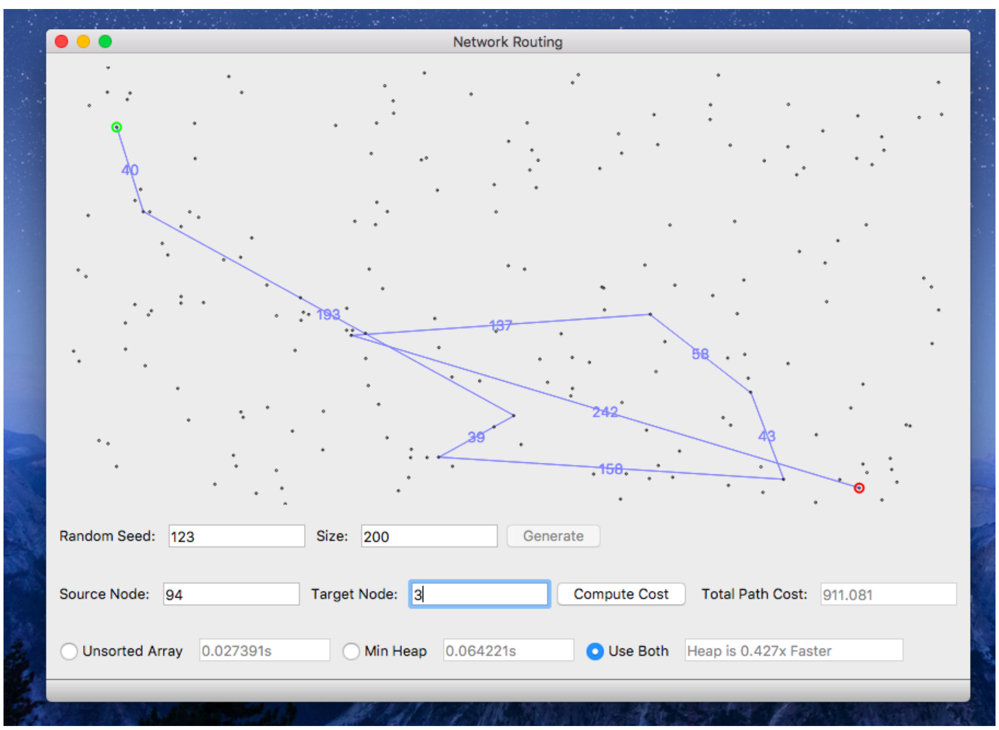
Space Complexity:
The algorithm keeps 2 arrays of size n for total distances and previous nodes thus O(2n)
Array: The array implementation stores another 3 arrays of size n for the graph, nodes, and popped nodes. Thus total space is O(5n)
Heap: The heap implementation also stores another 3 arrays. Thus is is the same space complexity O(5n).

Cody Kesler
CS 312 – 2
Lab 3

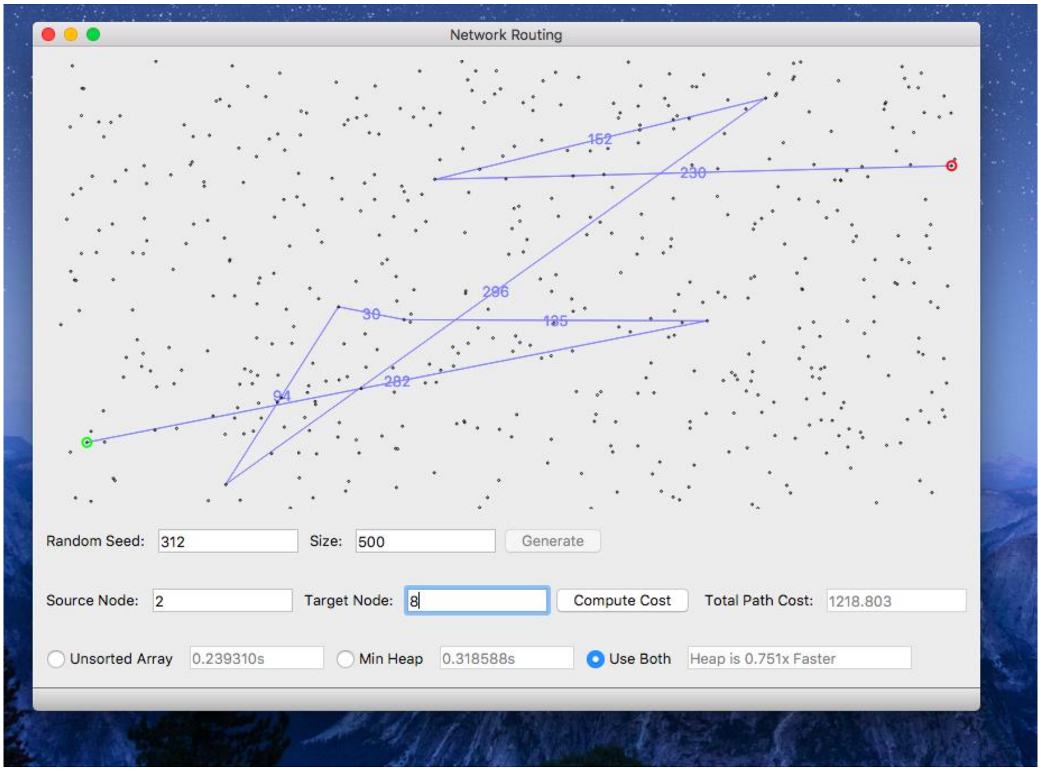
Screenshots:

There is no path.





Cody Kesler
CS 312 – 2
Lab 3



Graph and Chart:

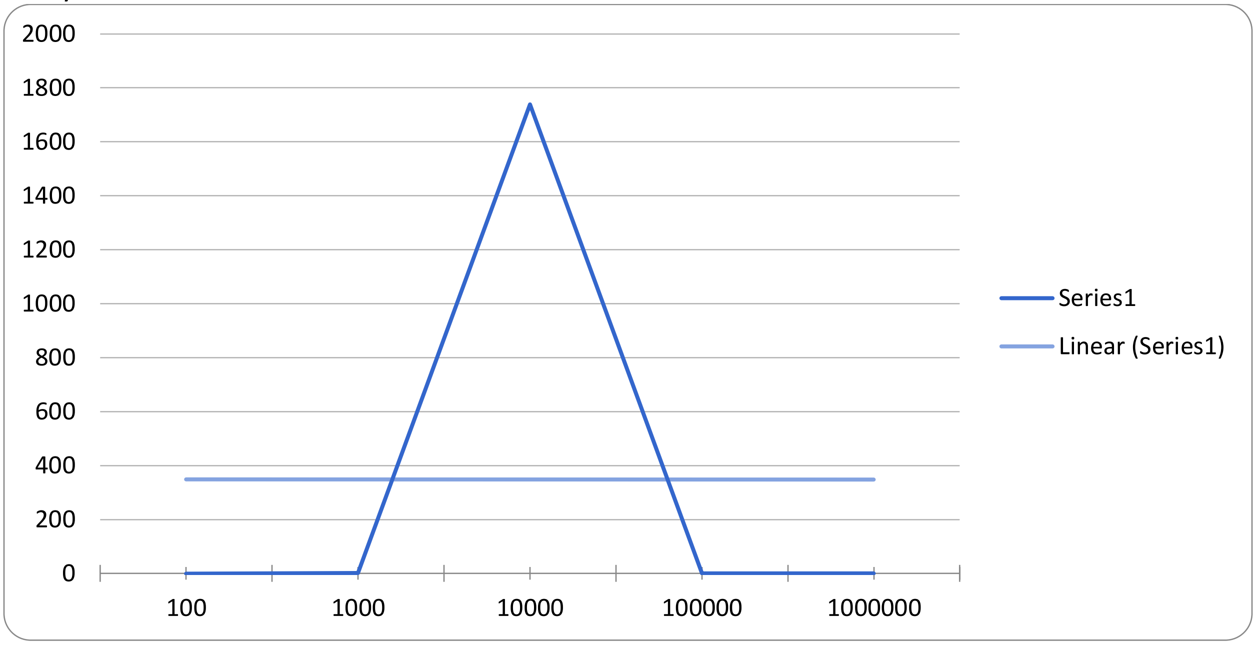
Array: Seed: 312		Trial1	Trial 2	Trial 3	Trial 4	Trial5	Avg
	100	0.009675	0.0078	0.00789	0.00743	0.00833	0.008225
	1000	1.658	1.903	1.734	1.674	1.926	1.779
	10000	1798.54	1711.89	1726.45	1753.27	1705.66	1739.162
	100000						
	1000000						

Heap: Seed: 312		Trial1	Trial 2	Trial 3	Trial 4	Trial5	Avg
	100	0.01977	0.1006	0.0207	0.0375	0.0239	0.040494
	1000	0.856	0.8695	0.8496	0.8471	1.006	0.88564
	10000	86.68	86.45	86.77	86.34	86.1	86.468
	100000						
	1000000						

Cody Kesler
CS 312 – 2
Lab 3

For some reason the program wouldn't run in an efficient manner on my laptop when running the higher node counts so I couldn't complete the time trials. But these are the graphs projecting the higher node count run times:

Array:



Heap: