Project 3 - Network Routing

Entire Pseudo Code (NetworkRoutingSolver.py)

```
#!/usr/bin/python3


from CS312Graph import *
import time
import math


class Queue:
    def __init__(self):
        self.queue = []

    def __len__(self):
        return len(self.queue)

    def deleteMin(self, dist):
        pass

    def decreaseKey(self, idx, dist):
        pass

    def insert(self, dist_arr_idx, dist):
        pass

    def makeQueue(self, num_dist_arr_indices, dist):      # Fn - T O(|V|), calls insert |V| times
        for i in range(num_dist_arr_indices):
            self.insert(i, dist)

class ArrayQueue(Queue):
    def deleteMin(self, dist):         # Fn - T O(log|V|), worst amount of recursive calls - S O(1), just stores ints
        min_idx = 0     # Index which points to smallest item in dist
        for i in range(len(self.queue)):
            if dist[self.queue[i]] < dist[self.queue[min_idx]]:     # Compare values in dist that exist in queue
                min_idx = i
        min_val = self.queue[min_idx]
        del self.queue[min_idx]
        return min_val

    def decreaseKey(self, idx, dist):         # Fn - T O(1) - S O(1)
        pass    # Don't have to do anything here for array implementation

    def insert(self, dist_arr_idx, dist):                                                    # Fn - T O(1), appending is constan
        self.queue.append(dist_arr_idx)


class HeapQueue(Queue):
    def __init__(self):
        super().__init__()
        self.map = []   # Says where to find nodes in the queue

    def __get_parent_idx(self, child_idx):
        if child_idx <= 2:
            return 0
        return (child_idx - 1) // 2

    def __get_left_child_idx(self, parent_idx):
        return (parent_idx * 2) + 1

    def __get_right_child_idx(self, parent_idx):
        return (parent_idx * 2) + 2

    def __get_last_idx(self):
        return len(self.queue) - 1

    def __get_min_child_idx(self, parent_idx, dist):     # Fn - T O(1) - S O(1), just stores ints
        lc_idx = self.__get_left_child_idx(parent_idx)
        rc_idx = self.__get_right_child_idx(parent_idx)

        # Edge cases
        if lc_idx > self.__get_last_idx() or rc_idx > self.__get_last_idx():
            return -1
```

```
        if (dist[self.queue[lc_idx]] < dist[self.queue[rc_idx]]):
            return lc_idx
        return rc_idx

    def __swap_values(self, idx1, idx2): # Fn - T O(1), just assignments - S O(1), just 2 ints created
        # Swap queue values
        temp = self.queue[idx1]
        self.queue[idx1] = self.queue[idx2]
        self.queue[idx2] = temp
        # Swap map values
        temp = self.map[self.queue[idx1]]
        self.map[self.queue[idx1]] = self.map[self.queue[idx2]]
        self.map[self.queue[idx2]] = temp

    def __bubble_up(self, idx, dist):         # Fn - T O(log|V|), calls bubble up - S O(1), makes 1 int
        parent_idx = self.__get_parent_idx(idx)

        # Bubble up (rec) if the parent is bigger than the child
        if dist[self.queue[parent_idx]] > dist[self.queue[idx]]:
            self.__swap_values(idx, parent_idx)
            self.__bubble_up(parent_idx, dist)

    def __sift_down(self, idx, dist):         # Fn - T O(log|V|), worst amount of recursive calls - S O(1), makes 1 int
        min_child_idx = self.__get_min_child_idx(idx, dist)

        # Edge cases
        if min_child_idx < 1 or min_child_idx > self.__get_last_idx():
            return

        # Sift down (rec) if parent is bigger than the smallest child
        if dist[self.queue[idx]] > dist[self.queue[min_child_idx]]:
            self.__swap_values(idx, min_child_idx)
            self.__sift_down(min_child_idx, dist)

    def deleteMin(self, dist):               # Fn - T O(log|V|), calls sift down - S O(1), just stores an int
        first_val = self.queue[0]
        self.queue[0] = self.queue[self.__get_last_idx()]   # Replace 1st val with last
        self.map[self.queue[0]] = 0
        self.queue.pop()                                    # Remove last item
        self.__sift_down(0, dist)
        return first_val

    def decreaseKey(self, node_val, dist):    # Fn - T O(log|V|), calls bubble up - S O(1), stores 1 int
        idx = self.map[node_val]
        self.__bubble_up(idx, dist)

    def insert(self, dist_arr_idx, dist):  # Fn - T O(log|V|), appending is constant but then calls bubble up - S O(1), adds values to arra
        self.queue.append(dist_arr_idx)
        self.map.append(dist_arr_idx)
        self.__bubble_up(self.__get_last_idx(), dist)


class NetworkRoutingSolver:
    def __init__(self):
        self.dist = []
        self.prev = []

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

    def getShortestPath( self, destIndex ):     # Complexity here depends on implementation, see #2 and #3 for details
        self.dest = destIndex

        path_edges = []

        cur_edge = self.prev[self.dest] # Start with last edge

        while cur_edge: # Work backwards
            path_edges.append( (cur_edge.src.loc, cur_edge.dest.loc, '{:.0f}'.format(cur_edge.length)) )
            cur_edge = self.prev[cur_edge.src.node_id]

        return {'cost':self.dist[self.dest], 'path':path_edges}


    def computeShortestPaths( self, srcIndex, use_heap=False ):  # Complexity here depends on implementation, see #2 and #3 for details
        self.source = srcIndex
        t1 = time.time()

        # Clear arrays each time you start over
```

```
        self.dist.clear()
        self.prev.clear()

        queue = HeapQueue() if use_heap else ArrayQueue()   # Sets the right queue implementation based on settings

        # Start of Dijkstra's                               #  T O(|V|(cost-to-insert+cost-to-deletemin)+|E|(cost-to-decreasekey)),
        self.dist = [math.inf] * len(self.network.nodes)    # See above, S O(|V|), biggest array size is amount of vertices
        self.prev = [None] * len(self.network.nodes)
        self.dist[srcIndex] = 0

        queue.makeQueue(len(self.network.nodes), self.dist)                                     # T O(|V|)*cost-to-insert
        while len(queue) > 0:                                                                   # T O(|V|), loop
            cur_node_idx = queue.deleteMin(self.dist)                                           # T O(1)*cost-to-deletemin

            cur_edges = self.network.nodes[cur_node_idx].neighbors

            for i in range(len(cur_edges)):                                                     # T O(|E|), loops through all edges
                dest_node_idx = cur_edges[i].dest.node_id

                if self.dist[cur_node_idx] + cur_edges[i].length < self.dist[dest_node_idx]:
                    self.dist[dest_node_idx] = self.dist[cur_node_idx] + cur_edges[i].length
                    self.prev[dest_node_idx] = cur_edges[i]
                    queue.decreaseKey(dest_node_idx, self.dist)                                 # T O(1)*cost-to-decreasekey

        t2 = time.time()
```

In code, `Fn` refers to function complexity
In code, comments with `T O(...)` are time complexities, then comments with `S O(...)` are space complexities.
If no time complexity is provided next to a line of code, it is O(1) constant time.
If no space complexity is provided next to a line of code, it doesn't cost any space.

1. Dijkstra's Algorithm

```
# Start of Dijkstra's          #  T O(|V|(cost-to-insert+cost-to-deletemin)+|E|(cost-to-decreasekey)),
self.dist = [math.inf] * len(self.network.nodes)         # S O(|V|), biggest array size is amount of vertices
    self.prev = [None] * len(self.network.nodes)
    self.dist[srcIndex] = 0

    queue.makeQueue(len(self.network.nodes), self.dist)                                 # T O(|V|)*cost-to-insert
    while len(queue) > 0:                                                               # T O(|V|), loop
        cur_node_idx = queue.deleteMin(self.dist)                                       # T O(1)*cost-to-deletemin

        cur_edges = self.network.nodes[cur_node_idx].neighbors

        for i in range(len(cur_edges)):                                                 # T O(|E|), loops through all edges
            dest_node_idx = cur_edges[i].dest.node_id

            if self.dist[cur_node_idx] + cur_edges[i].length < self.dist[dest_node_idx]:
                self.dist[dest_node_idx] = self.dist[cur_node_idx] + cur_edges[i].length
                self.prev[dest_node_idx] = cur_edges[i]
                queue.decreaseKey(dest_node_idx, self.dist)
```

2. Function Complexities

Array Implementation

`insert()` - $O(1)$

```
def insert(self, dist_arr_idx, dist):   # Fn - T O(1), appending is constant - S O(1), adds one item to array
    self.queue.append(dist_arr_idx)
```

Simply appends an item to the end of an array, which is constant.

`deleteMin()` - $O(|V|)$

```
  def deleteMin(self, dist):         # Fn - T O(log|V|), worst amount of recursive calls - S O(1), just stores ints
    min_idx = 0     # Index which points to smallest item in dist
    for i in range(len(self.queue)):
        if dist[self.queue[i]] < dist[self.queue[min_idx]]:     # Compare values in dist that exist in queue
            min_idx = i
    min_val = self.queue[min_idx]
    del self.queue[min_idx]
    return min_val
```

It's unavoidably $O(|V|)$ since we'll have to loop through each item in the queue to find out which queue item corresponds to the smallest distance.

`decreaseKey()` - $O(1)$

```
def decreaseKey(self, idx, dist):             # Fn - T O(1) - S O(1)
        pass    # Don't have to do anything here for array implementation
```

Constant since we don't have to implement anything here for the array implementation (in the binary heap we have to do extra work here to let the queue know that priority has changed and it needs sorted, but here our queue isn't sorted so we'll have to loop through everything anyway on the next iteration).

Binary Heap Implementation

Necessary helper functions

`bubble_up()` - $O(log|V|)$

```
def __bubble_up(self, idx, dist):     # Fn - T O(log|V|), calls bubble up - S O(1), makes 1 int
    parent_idx = self.__get_parent_idx(idx)

    # Bubble up (rec) if the parent is bigger than the child
    if dist[self.queue[parent_idx]] > dist[self.queue[idx]]:
        self.__swap_values(idx, parent_idx)
        self.__bubble_up(parent_idx, dist)
```

To grab an index is essentially constant (requires multiplication but it's negligible since that's not tied to the input size). Swapping values is linear, because you'll only have to move around 2 values at most. That said, the dominant operation here is the recursive call to itself to bubble up. Since our data structure is a tree, the max number of times that this function could be called before hitting the base case is $log|V|$, so that's the overall complexity since it's the dominant operation.

`sift_down()` - $O(log|V|)$

```
def __sift_down(self, idx, dist):    # Fn - T O(log|V|), worst amount of recursive calls - S O(1), makes 1 int
    min_child_idx = self.__get_min_child_idx(idx, dist)

    # Edge cases
    if min_child_idx < 1 or min_child_idx > self.__get_last_idx():
        return

    # Sift down (rec) if parent is bigger than the smallest child
    if dist[self.queue[idx]] > dist[self.queue[min_child_idx]]:
        self.__swap_values(idx, min_child_idx)
        self.__sift_down(min_child_idx, dist)
```

To grab an index is essentially constant (requires multiplication but it's negligible since that's not tied to the input size). Swapping values is linear, because you'll only have to move around 2 values at most. That said, the dominant operation here is the recursive

call to itself to sift down. Since our data structure is a tree, the max number of times that this function could be called before hitting the base case is $log|V|$, so that's the overall complexity since it's the dominant operation.

`insert()` - $O(log|V|)$

```
def insert(self, dist_arr_idx, dist):   # Fn - T O(log|V|), appending is constant but then calls bubble up - S O(1), adds values to array
    self.queue.append(dist_arr_idx)
    self.map.append(dist_arr_idx)
    self.__bubble_up(self.__get_last_idx(), dist)  # See __bubble_up() in src code above
```

Appending to 2 arrays is constant so those are negligible in terms of time, compared to the call to bubble up. Bubble up is $O(log(|V|)$ (see `__bubble_up()` code above for details), which is the dominating operation, so the insert function is $O(log(|V|)$ overall.

`deleteMin()` - $O(log|V|)$

```
def deleteMin(self, dist):          # Fn - T O(log|V|), calls sift down - S O(1), just stores an int
    first_val = self.queue[0]
    self.queue[0] = self.queue[self.__get_last_idx()]   # Replace 1st val with last
    self.map[self.queue[0]] = 0
    self.queue.pop()                                    # Remove last item
    self.__sift_down(0, dist)
    return first_val
```

Re-assigning values in the arrays (first 3 lines) and popping an item off (4th line) are all constant operations. The dominating operation here is the call to sift down, which has a complexity of $O(log|V|)$ (see `sift_down()` code above for details), so the deleteMin function is $O(log(|V|)$ overall.

`decreaseKey()` - $O(log|V|)$

```
def decreaseKey(self, node_val, dist):   # Fn - T O(log|V|), calls bubble up - S O(1), stores 1 int
    idx = self.map[node_val]
    self.__bubble_up(idx, dist)
```

Setting a variable value from referencing an array is constant, so the dominating operation here is the second line which makes a call to bubble up. Bubble up is $O(log(|V|)$ (see `__bubble_up()` code above for details), so the decreaseKey function is $O(log(|V|)$ overall.

(This speed up from $O(|V|)$ to $O(log|V|)$ is made possible because we store and update the `map` array which keeps track of the mapping between nodes and queue indices so that we don't have to loop through the entire queue (linear) to find a node).

3. Overall Complexities (Time and Space)

Array Implementation

Time - $O(|V|^2)$

To figure out the overall time complexity, we can just sum up each of our array functions and look at the overall results in Big O (also multiplying for iterations). Insert and decreaseKey are constant so those won't have a noticeable effect in Big O, at least compared to deleteMin, which is $O(|V|)$ (see explantions in #2 above for rationale). Since |V| + 1 + 1 comes out to be $O(|V|)$ in Big O, that's the cost of 1 iteration in Big O, and then with |V| iterations, the overall time complexity comes out to be $O(|V|^2)$.

Space - $O(|V|)$

We have to store a queue (`queue`) that has initially been allocated such that there's an array item for each vertex/node. We have other arrays also made like the distance array (`dist`) and previous array (`prev`) which store those arrays of the same size. So technically we use around 3 times the space of the number of nodes with some extra spaces for variables and such, but in Big O that will come out to be $O(|V|)$ space complexity.

Binary Heap Implementation

Time - $O((|V| + |E|)log|V|)$

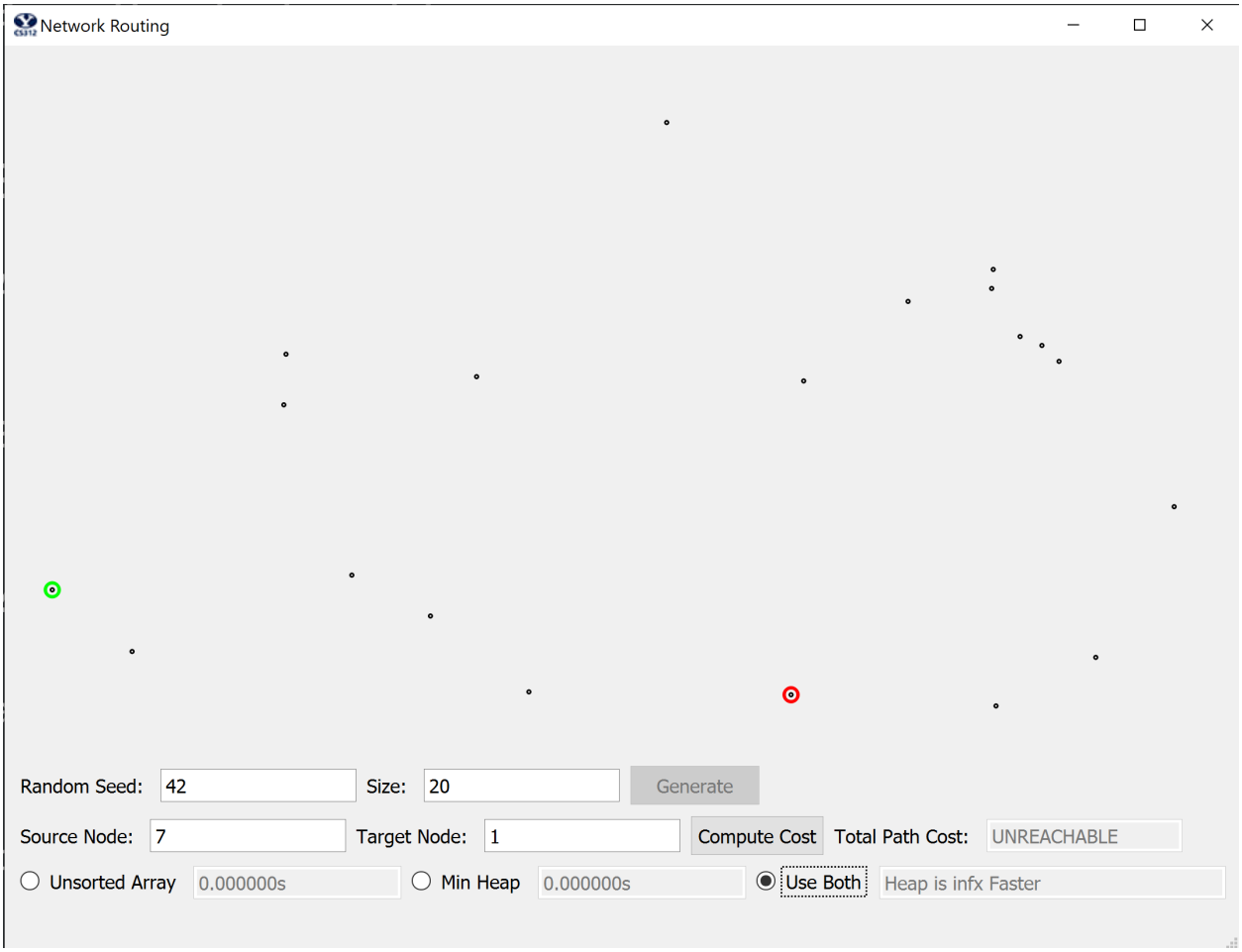
To figure out the overall time complexity, we can just sum up each of our array functions and look at the overall results in Big O (also multiplying for iterations). Insert, decreaseKey, and deleteMin all are $O(log|V|)$ (see #2 explanation above for details), so added them gets you 3log|V| or $O(log|V|)$ in Big O (this is for one iteration). Since we might have to loop through all nodes and all edges, we multiply that one iteration by |V| and |E|, which turns out to have a $O((|V| + |E|)log|V|)$ Big O complexity. We like this better than the array, because $|E|$ is better than $|V|^2$ for all graphs. In general, we prefer log|V| over |V|, so this has clear advantages overall even though its insert and decreaseKey are slower.

Space - $O(|V|)$

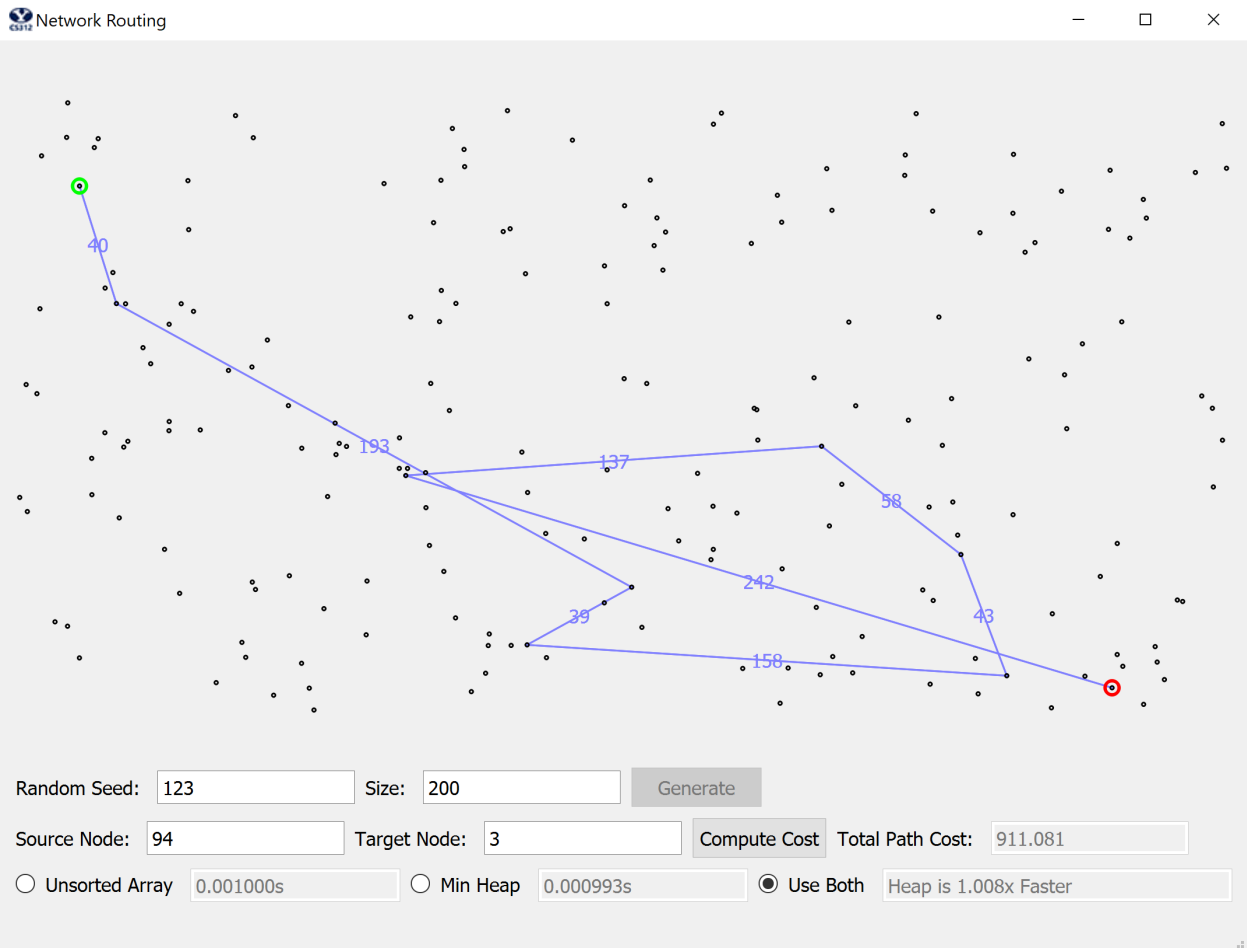
We have to store a queue (`queue`) that has initially been allocated such that there's an array item for each vertex/node. We have other arrays also made like the distance array (`dist`) and previous array (`prev`) which store those arrays of the same size. With the binary heap we also have to store an extra array that maps nodes to indexes in our queue (`map`). So technically we use around 4 times the space of the number of nodes with some extra spaces for variables and such, but in Big O that will come out to be $O(|V|)$ space complexity.

4. Shortest Path Test Screenshots

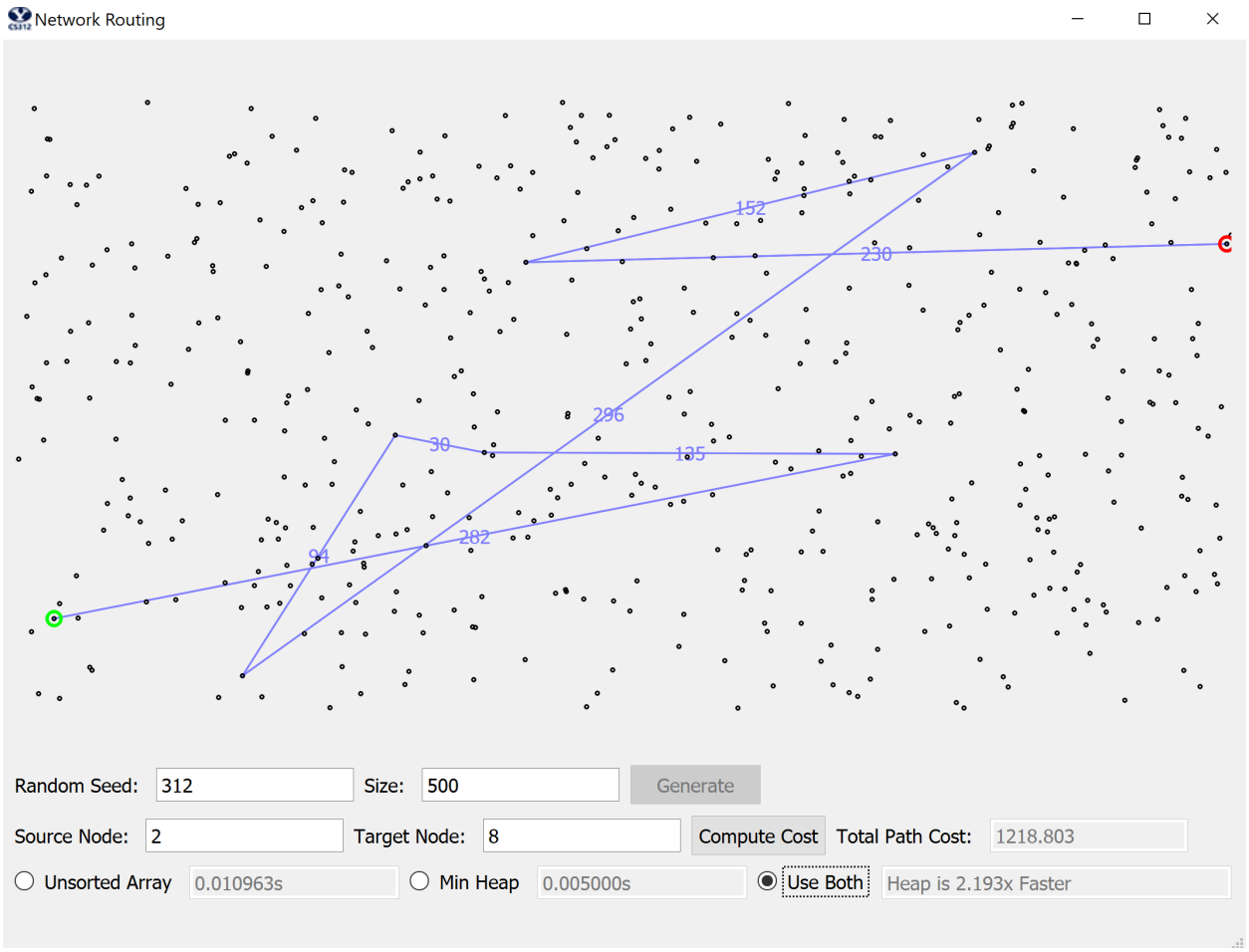
a.


Network Routing
Random Seed:
42
Size:
20
Generate
Source Node:
7
Target Node:
1
Compute Cost
Total Path Cost:
UNREACHABLE
Unsorted Array
0.000000s
Min Heap
0.000000s
Use Both
Heap is infx Faster

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
0.001000s
Min Heap
0.000993s
Use Both
Heap is 1.008x Faster

c.



5. Data Table, Graph, and Explanations

Tables

Array Implementation

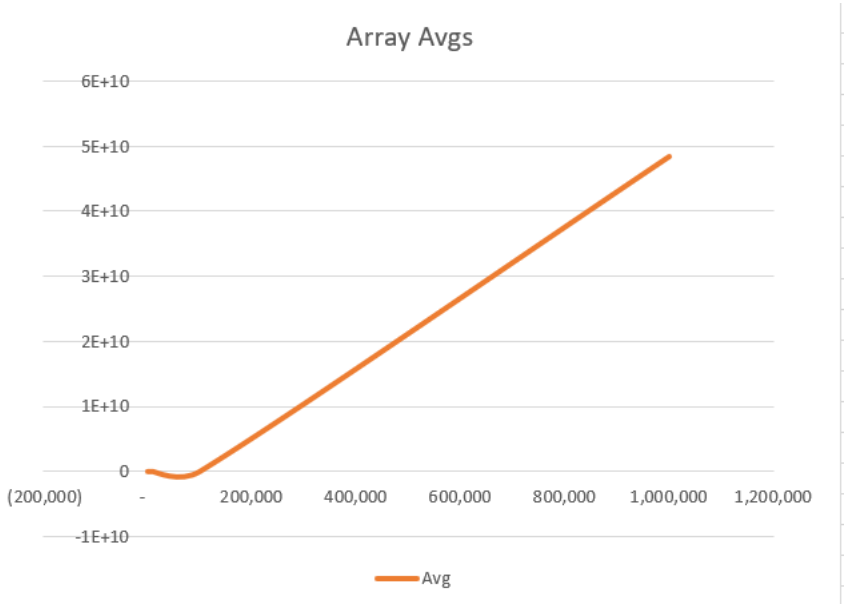
	Array				
n	100	1,000	10,000	100,000	1,000,000
					(Estimates)
5 Tests	0.0008	0.0450	5.3069	625.3376	61117989835.1844
(Elapsed	0.0009	0.0439	5.6335	662.3763	39857414861.5002
Time)	0.0009	0.0449	5.4519	643.2142	51035595454.5166
	0.0009	0.0470	5.4579	635.2309	45670709764.6557
	0.0009	0.4410	5.4069	621.3452	44222834255.0523
Avg	0.00088	0.12436	5.45142	637.50084	48380908834

Heap Implementation

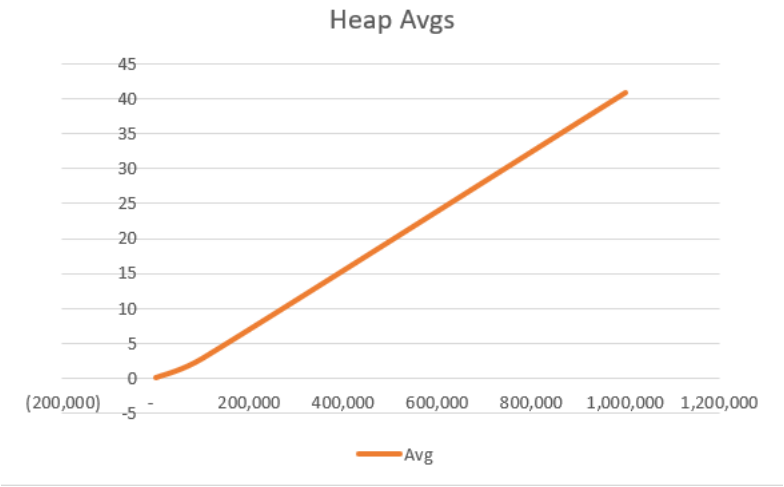
	Heap				
n	100	1,000	10,000	100,000	1,000,000
5 Tests	0.0009	0.0139	0.2169	2.2823	39.3210
(Elapsed	0.0010	0.0130	0.2310	2.8609	42.2381
Time)	0.0010	0.0140	0.2200	2.9859	38.1870
	0.0010	0.0129	0.2280	2.8610	39.8790
	0.0010	0.0139	0.2209	2.8959	44.8730
Avg	0.00098	0.01354	0.22336	2.7772	40.89962

Graphs (x axis is n, y axis is time in seconds)

Array Implementation



Heap Implementation



Explanations

The heap implementation is overall $O((|V| + |E|)log|V|)$ compared to the array implementation which is $O(|V|^2)$, so it makes sense that the heap implementation is generally faster than the array. With super small sample sizes, the array actually performs faster, probably since it requires less overhead and looping through n^2 times isn't much of a cost on a small data set. However, it quickly becomes obvious that on data sets 1,000 and up, the heap is the clear winner due to its logarithmic scaling as the size increases.