Part 1: See Appendix
Part 2: See Appendix

Part 3: Time Complexity

PriorityQueueArray - deleteMin / makequeue

```
def delete_min(self):
       min_node = min(self.queue, key=self.queue.get) # O(n) operation
       del self.queue[min_node]
       return min_node
def makequeue(self, nodes, distances): # O(n) operation
       for node in nodes:
           self.insert(node)
```

Explanation:

- The time complexity of these functions are O (|V|). This is because in the array/list implementation of a priority queue we have to iterate through all the nodes in the queue to find the smallest key then remove it. The makequeue has to iterate through all the nodes and throw them in the queue. So the time complexity for both of these functions is the number of vertices in the graph.
- The space complexity of delete_min is O(1) because we aren't storing anything that can vary in regard to the input. The make queue has a space complexity of O(|V|) though since it is loading up a queue varying by the length of the amount of nodes in the graph.

PriorityQueueHeap - insert / decreaseKey

```
   def insert(self, node, priority): # O(logn)
       self.heap.append(node)
       self.heap_node_positions[node.node_id] = len(self.heap) - 1
       self.priorities[node.node_id] = priority
       self.bubble_up(len(self.heap) - 1) # O(logn)

   def decrease_key(self, node, new_priority): # O(logn)
       index = self.heap_node_positions[node.node_id]
       self.priorities[node.node_id] = new_priority
       self.bubble_up(index) # O(logn)
```

Explanation:

- The time complexity of these functions are both O(Logn). The first 3 operations of the O(1), the O(log (|V|) complexity comes from the bubble_up function. This function has to at worst case iterate through the entire length of our graph starting from the bottom bubbling up a node until it reaches the top. So at worst case the entire depth of the tree upwards which is log(|V|). Decrease key gets its complexity from bubble_up as well.
- The space complexity of these functions are relatively constant O(1) because we

don't have any recursion going on and not any new data structures made.

PriorityQueueHeap - deleteMin

```
def delete_min(self): # O(logn)
        min_node = self.heap[0]
        last_node = self.heap.pop()
        if self.heap:
            self.heap[0] = last_node
            self.heap_node_positions[last_node.node_id] = 0
            self.sift_down(0) # O(logn)
        self.heap_node_positions.pop(min_node.node_id)
        self.priorities.pop(min_node.node_id)
        return min_node
```

Explanation:

- The time complexity of this function is O(log (|V|). This is what makes the priority queue with a heap implementation faster than an array implementation with dijkstra's algorithm. This is because when we first grab that min_node it is a constant operation since it is at the top of the heap by design. Then after we remove it from the heap we have to at worst case sift down fixing the heap the entire depth of the heap which is log|V| which is where the majority of the time complexity comes from.
- The space complexity of this function is O(log|V|). This comes from the sift_down function because the sift_down function as I will get into more below is a recursive function where in the worst case we have log|V| recursive calls sifting down the heap to fix it after deleting the min node.

PriorityQueueHeap - makeQueue

```
def makequeue(self, nodes, distances): # O(nlogn)
        for node in nodes:
            self.insert(node, distances[node])
```

Explanation:

- The time complexity of this function is O(|V|log|V|). This is because we are running insert |V| amount of times and insert has a time complexity of O(log|V|).
- The space complexity of this function is O(|V|) because we are filling up the heap according to the amount of nodes in the graph taking up |V| amount of space.

PriorityQueueHeap - bubbleUp / siftDown

```
def bubble_up(self, index): # O(logn)
        parent_index = (index - 1) // 2
        while index > 0 and self.compare_nodes(index, parent_index):
            self.swap(index, parent_index)
            index = parent_index
            parent_index = (index - 1) // 2

def sift_down(self, index): # O(logn)
        smallest = index
```

```
        left_index = 2 * index + 1
        right_index = 2 * index + 2

        if left_index < len(self.heap) and self.compare_nodes(left_index, smallest):
            smallest = left_index
        if right_index < len(self.heap) and self.compare_nodes(right_index, smallest):
            smallest = right_index

        if smallest != index:
            self.swap(index, smallest)
            self.sift_down(smallest)
```

Explanation:

- The time complexity of these function are O(log|V|). This is because in the worst case for both of these functions we have to navigate the entire depth of the heap either up or down. For bubble up that simply means when we place a new thing in the heap we have to bubble it up all the way until it reaches the root in the worst case. For the sift down it's the opposite. We place the last node in the heap at the bottom of the heap at the root then sift down in the worst case the entire depth of the heap. Both of these are O(log|V|) according to the depth of the heap.
- The space complexity of the bubble up operation is relatively constant since it is just an iterative function swapping until it reaches the bottom and isn't making any new data structures. The sift down function I would say has a space complexity of O(log|V|) because in the worst case is has log|V| recursive calls which take up space.

Dijkstra's algorithm

```
def dijkstra(self, src, pq): # O((V+E)logV) or O(V^2) depending on the priority queue
        # Initialize all distances to infinity
        distances = {}
        prev = {}
        for node in self.network.nodes:
            distances[node] = float('inf')
            prev[node] = None
        distances[src] = 0
        # Insert all nodes into the priority queue
        pq.makequeue(self.network.nodes, distances) # O(nlogn) or O(n) depending on queue
        #pq.printqueue()
        while not pq.is_empty():
            u = pq.delete_min() # O(logn) or O(n) depending on queue
            #print(f'Node from delete_min: {u.node_id}, Distance: {distances[u]}')
            for edge in u.neighbors:
                v = edge.dest
                if distances[v] > distances[u] + edge.length:
                    distances[v] = distances[u] + edge.length
                    prev[v] = u
                    pq.decrease_key(v, distances[v])
```

```
        return distances, prev
```

Explanation:

- The time complexity of this function is # O((|V|+|E|)log|V|)) or O(|V|^2) depending on the priority queue. This is because our complexity mostly comes from our priority queue implementation. With an array implementation the delete_min is O(|V|) and so is the decrease_key and makequeue. With the heap implementation decrease_key and delete_min are much faster at log|V|. I went into the details of these functions above^. So depending on our implementation of the priority queue our performance can drastically change, in this case the heap is much faster.
- The space complexity of this function is O(|V|). This is because at the maximum for the two dictionaries and the priority queue we have all the nodes in the graph stored.

getShortestPath

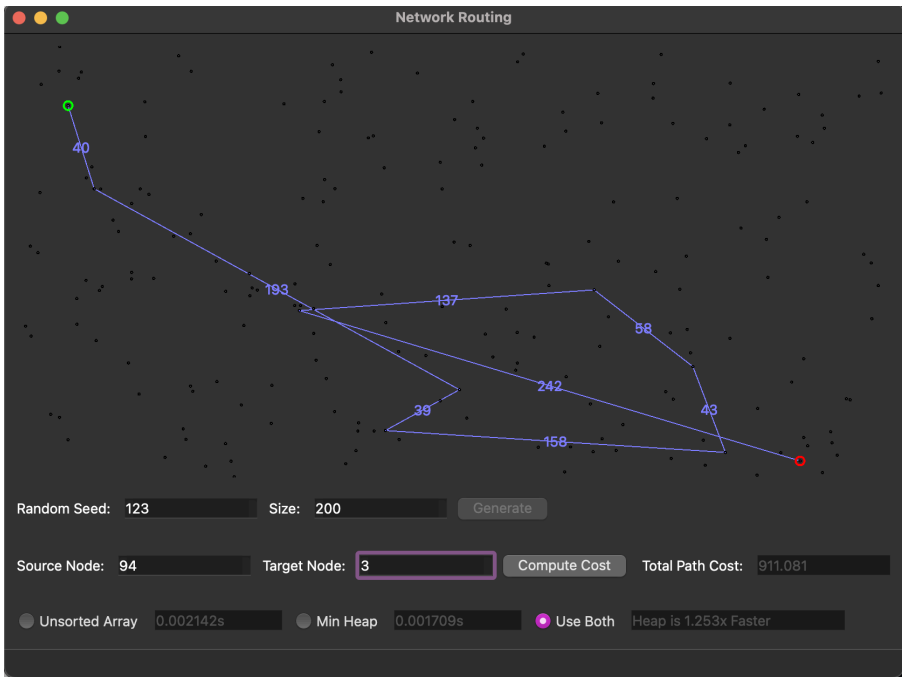
```
    def getShortestPath( self, destIndex ):
        self.dest = destIndex
        path_edges = []
        total_length = 0
        node = self.network.nodes[self.dest]

        while node != self.network.nodes[self.source]:
            # No path to node
            if self.prev[node] == None:
                return {'cost':float('inf'), 'path':[]}

            prev_node = self.prev[node]
            for edge in prev_node.neighbors:
                if edge.dest == node:
                    path_edges.append( (prev_node.loc, edge.dest.loc,
'{:.0f}'.format(edge.length)) )
                    total_length += edge.length
                    break
            node = prev_node
        path_edges.reverse() # Reverse the list to get the correct order
```

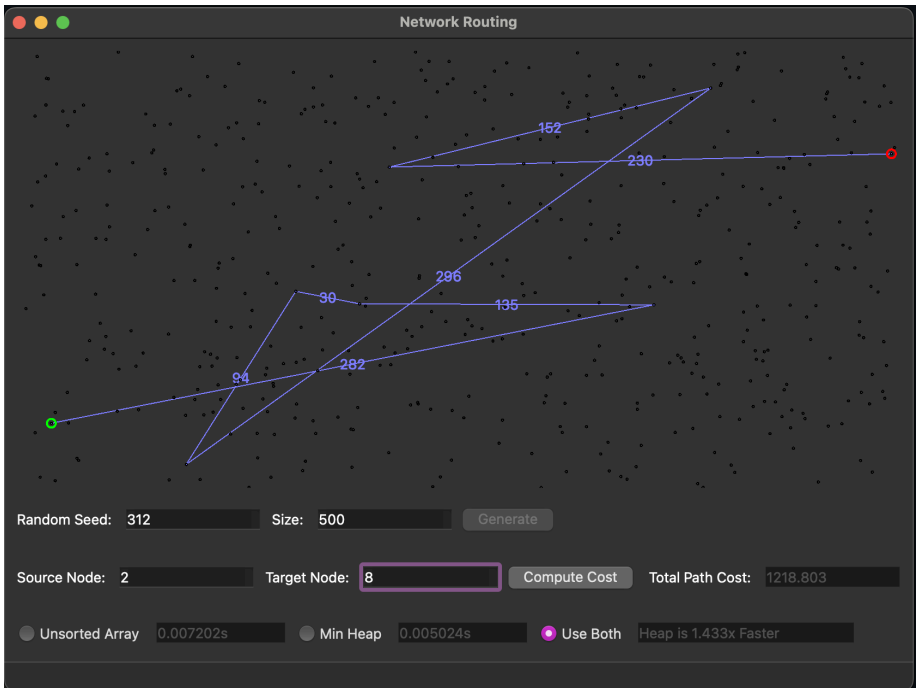
Explanation:

- The time complexity of this function is O(|V|). This is because of the outer while loop which iterates through every node in the graph. The inner for loop does not contribute much to the complexity because it is relatively constant since for this case in this lab the nodes have a maximum number of neighbors which is 3. So this makes the inner for loop relatively constant for looping through the number of neighbors (connected by edges). Reversing the list at the very end to get the correct order is also O(|V|) so the overall complexity is O(|V|).

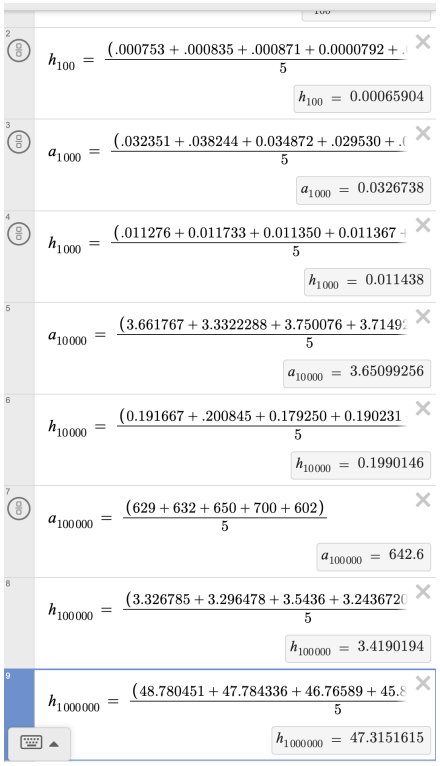
- The space complexity of this function is O(|V|) since we are building up the path_edges and in the worst case it would be every node in the graph involved in all these edges.

Part 4: Screenshots of working examples

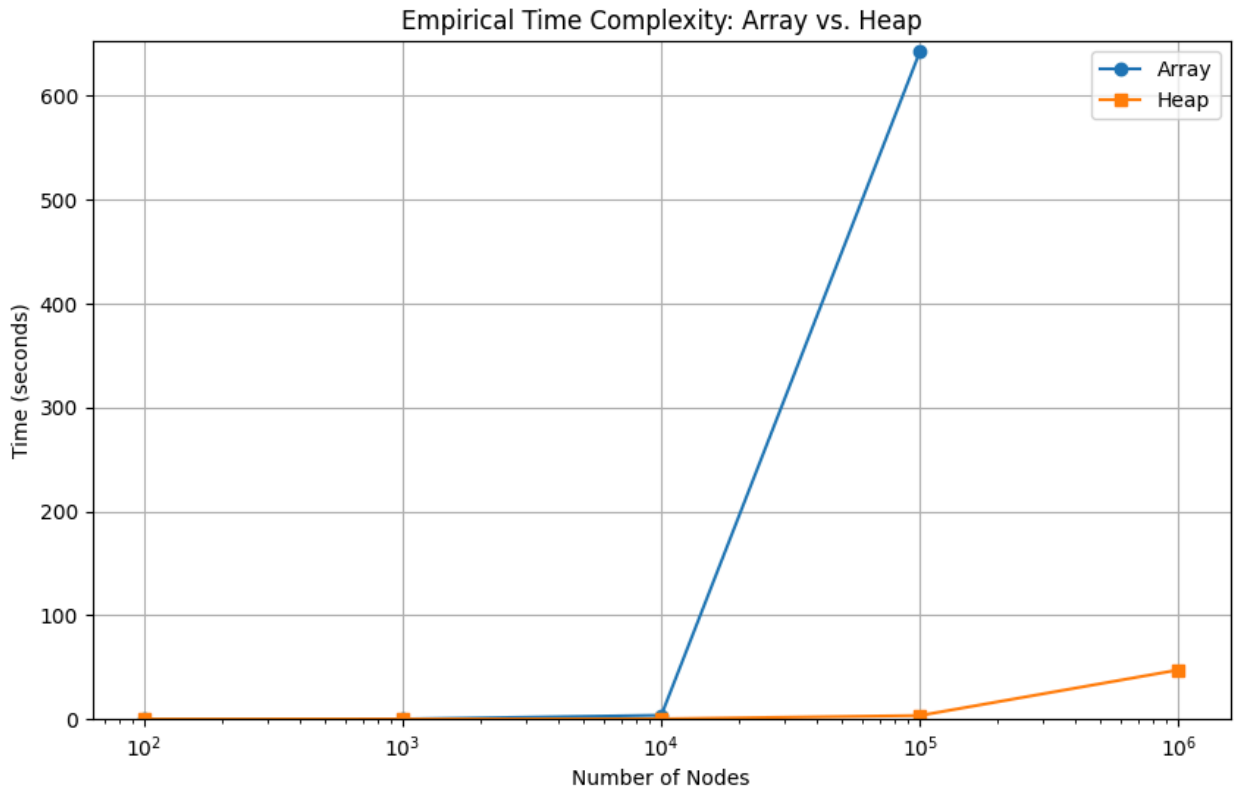




Part 5: Comparing heap vs array implementation of the priority queue.



n	heap_time	array_time
100	0.000659	0.000686
1000	0.011438	0.032674
10000	0.199015	3.650993
100000	3.419019	642.600000
1000000	47.315162	64260.000000



Discussion:

- As we've discussed the time complexity for the array implementation of a priority queue for dijkstra's algorithm is O(|V|^2). This lines up with the results from testing because as you can see the Array curve begins an exponential climb once the number of nodes reach past 100,000. You can see this trend in the table as well. The heap implementation stays pretty steady being able to handle 1,000,000 nodes in about 47 seconds. I included my prediction in the table for how long the array implementation would take to do 1,000,000 nodes. My estimate is about 64,260 seconds since the complexity is quadratic. If we increase the number of nodes by a factor of 10 (from 100,000 to 1,000,000), we would expect the time to increase by a factor of 10^2 = 100. So then 642.6*100 = 64,260.
- The curve for the heap is what was expected based on the time complexity following O((|V|+|E|)log|V|)) for the heap implementation of a priority queue for dijkstra's algorithm. This curve will continue to take a steady climb in a nlogn shape as the number of nodes increases.
- Overall the graph and table is about the results I expected.

Appendix

```python
#!/usr/bin/python3


from CS312Graph import *
import time

# helper methods
def print_distances(distances):
   print('Distances\n')
   for node in distances:
       print(f'Node: {node.node_id}, Distance: {distances[node]}')
def print_prev(prev):
   print('Previous\n')
   for node in prev:
       if(prev[node] == None):
           print(f'Node: {node.node_id}, Previous: None')
       else:
           print(f'Node: {node.node_id}, Previous: {prev[node].node_id}')

# Priority Queue using Array
class PriorityQueueArray:
   def __init__(self):
       self.queue = {}

   # Doesn't care about the order of the queue just throws it in.
   def insert(self, node): # O(1) operation
       self.queue[node] = float('inf')

   def decrease_key(self, node, new_val):
       self.queue[node] = new_val # O(1) operation

   def delete_min(self):
       min_node = min(self.queue, key=self.queue.get) # O(n) operation
       del self.queue[min_node]
       return min_node

   def is_empty(self):
       return len(self.queue) == 0 # O(1) operation

   def makequeue(self, nodes, distances): # O(n) operation
       for node in nodes:
           self.insert(node)

   def printqueue(self): # O(n) operation
       for node in self.queue:
           print(f'Node: {node.node_id}, Distance: {self.queue[node]}')

class PriorityQueueHeap:
   def __init__(self):
```

```
    self.heap = []  # List of CS312GraphNode
    self.heap_node_positions = {}  # Node_id: index in heap
    self.priorities = {}  # Node_id: priority

def insert(self, node, priority): # O(logn)
    self.heap.append(node)
    self.heap_node_positions[node.node_id] = len(self.heap) - 1
    self.priorities[node.node_id] = priority
    self.bubble_up(len(self.heap) - 1)

def decrease_key(self, node, new_priority): # O(logn)
    index = self.heap_node_positions[node.node_id]
    self.priorities[node.node_id] = new_priority
    self.bubble_up(index)

def delete_min(self): # O(logn)
    min_node = self.heap[0]
    last_node = self.heap.pop()
    if self.heap:
        self.heap[0] = last_node
        self.heap_node_positions[last_node.node_id] = 0
        self.sift_down(0)
    self.heap_node_positions.pop(min_node.node_id)
    self.priorities.pop(min_node.node_id)
    return min_node

def is_empty(self): # O(1)
    return len(self.heap) == 0

def makequeue(self, nodes, distances): # O(nlogn)
    for node in nodes:
        self.insert(node, distances[node])

def bubble_up(self, index): # O(logn)
    parent_index = (index - 1) // 2
    while index > 0 and self.compare_nodes(index, parent_index):
        self.swap(index, parent_index)
        index = parent_index
        parent_index = (index - 1) // 2

def sift_down(self, index): # O(logn)
    smallest = index
    left_index = 2 * index + 1
    right_index = 2 * index + 2

    if left_index < len(self.heap) and self.compare_nodes(left_index, smallest):
        smallest = left_index
    if right_index < len(self.heap) and self.compare_nodes(right_index, smallest):
        smallest = right_index

    if smallest != index:
        self.swap(index, smallest)
```

```
            self.sift_down(smallest)

    def compare_nodes(self, index1, index2): # O(1)
        node_id1 = self.heap[index1].node_id
        node_id2 = self.heap[index2].node_id
        return self.priorities[node_id1] < self.priorities[node_id2]

    def swap(self, index1, index2): # O(1)
        self.heap[index1], self.heap[index2] = self.heap[index2], self.heap[index1]
        self.heap_node_positions[self.heap[index1].node_id] = index1
        self.heap_node_positions[self.heap[index2].node_id] = index2

    def printqueue(self): # O(n)
        for node in self.heap:
            print(f'Node: {node.node_id}, Distance: {self.priorities[node.node_id]}')


class NetworkRoutingSolver:
    def __init__( self):
        pass

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

    def getShortestPath( self, destIndex ):
        self.dest = destIndex
        path_edges = []
        total_length = 0
        node = self.network.nodes[self.dest]

        while node != self.network.nodes[self.source]:
            # No path to node
            if self.prev[node] == None:
                return {'cost':float('inf'), 'path':[]}

            prev_node = self.prev[node]
            for edge in prev_node.neighbors:
                if edge.dest == node:
                    path_edges.append( (prev_node.loc, edge.dest.loc,
'{:.0f}'.format(edge.length)) )
                    total_length += edge.length
                    break
            node = prev_node
        path_edges.reverse() # Reverse the list to get the correct order

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

    def computeShortestPaths( self, srcIndex, use_heap=False ):
        self.source = srcIndex
        if use_heap:
            pq = PriorityQueueHeap()
```

```
        else:
            pq = PriorityQueueArray()

        t1 = time.time()
        self.distances, self.prev = self.dijkstra(self.network.nodes[srcIndex], pq)
        #print_distances(self.distances)
        #print_prev(self.prev)
        t2 = time.time()
        return (t2-t1)

    def dijkstra(self, src, pq): # O((V+E)logV) or O(V^2) depending on the priority queue
        # Initialize all distances to infinity
        distances = {}
        prev = {}
        for node in self.network.nodes:
            distances[node] = float('inf')
            prev[node] = None
        distances[src] = 0
        # Insert all nodes into the priority queue
        pq.makequeue(self.network.nodes, distances)
        #pq.printqueue()
        while not pq.is_empty():
            u = pq.delete_min()
            #print(f'Node from delete_min: {u.node_id}, Distance: {distances[u]}')
            for edge in u.neighbors:
                v = edge.dest
                if distances[v] > distances[u] + edge.length:
                    distances[v] = distances[u] + edge.length
                    prev[v] = u
                    pq.decrease_key(v, distances[v])

        return distances, prev
```