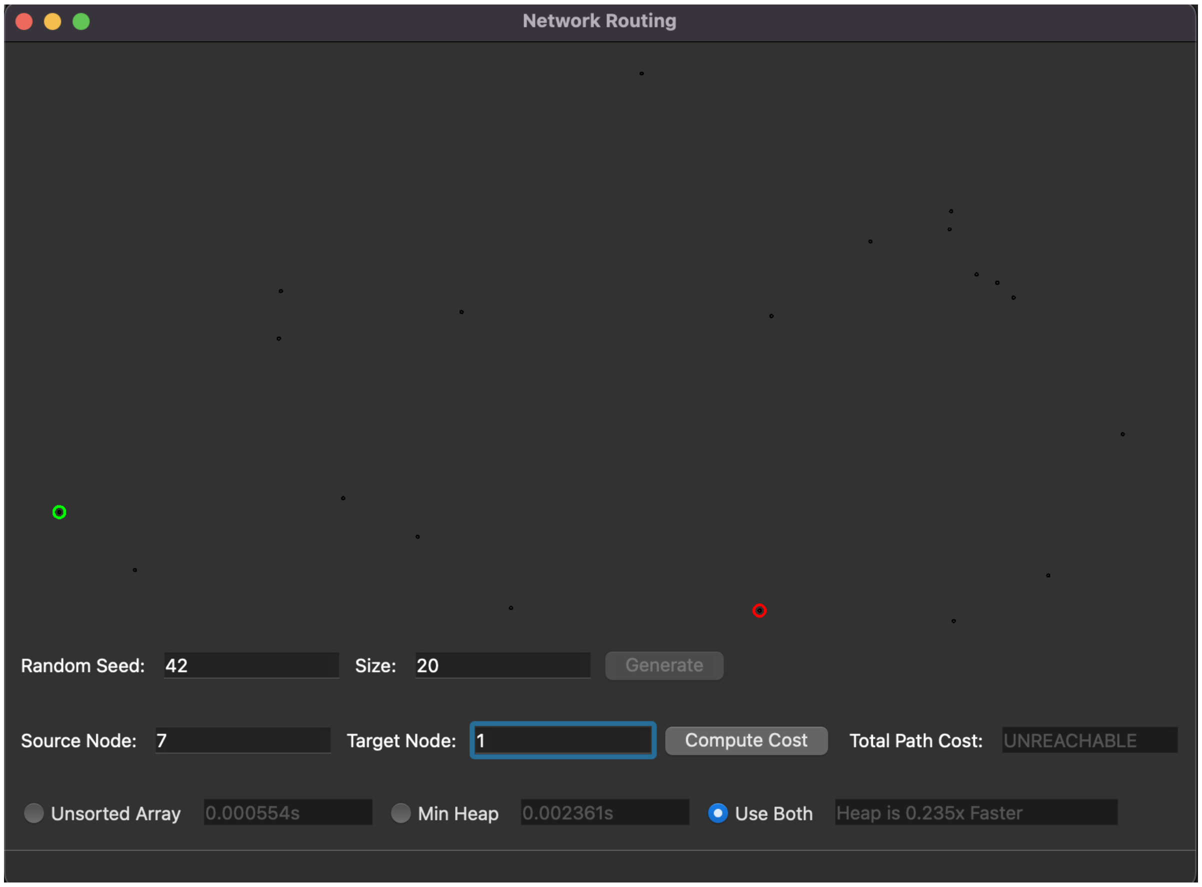
Lab 3: Network Routing

Seed 42, Size 20: No Reachable

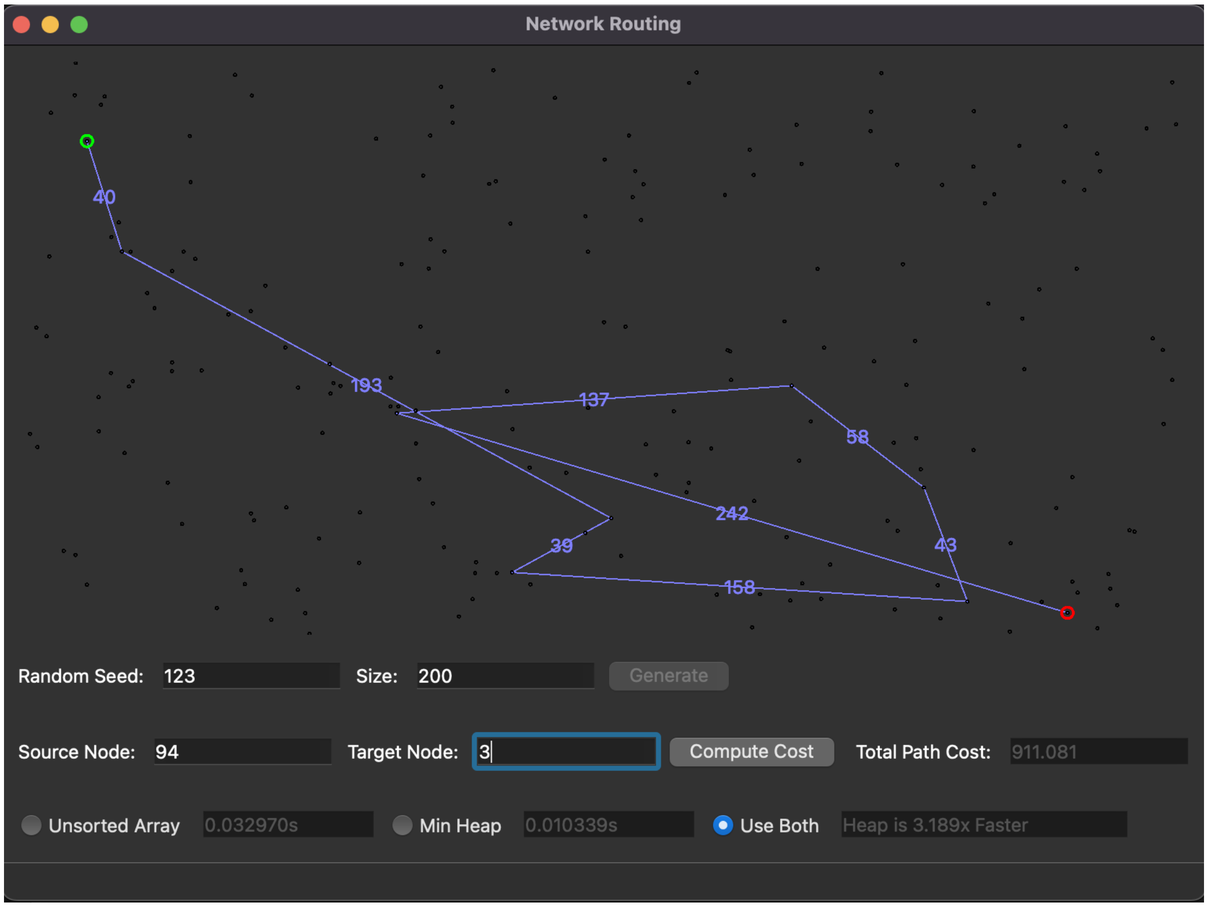


Random Seed: 42 Size: 20

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

Unsorted Array 0.000554s Min Heap 0.002361s [✓ Use Both] Heap is 0.235x Faster

Seed 123, Size 200: Length 911.081

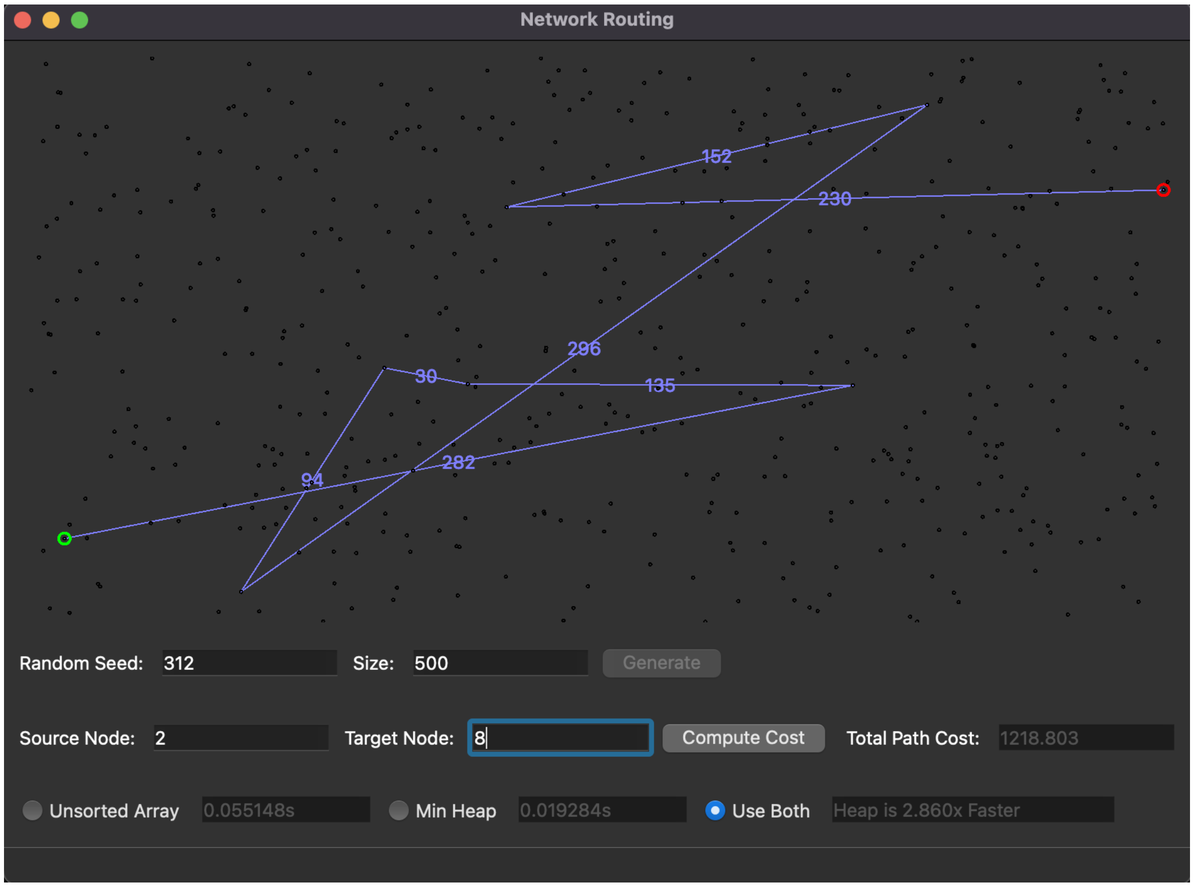


Random Seed: 123 Size: 200

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

Unsorted Array 0.032970s Min Heap 0.010339s [✓ Use Both] Heap is 3.189x Faster

Seed 312, Size 500: Length 1218.803

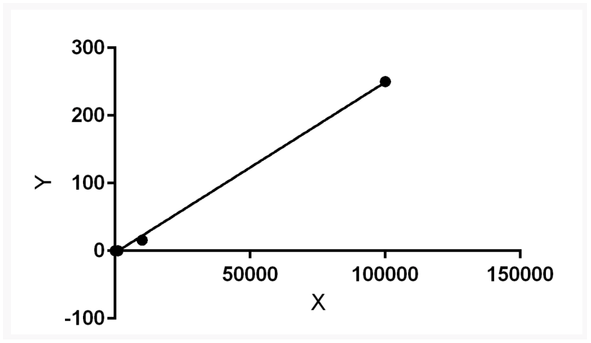
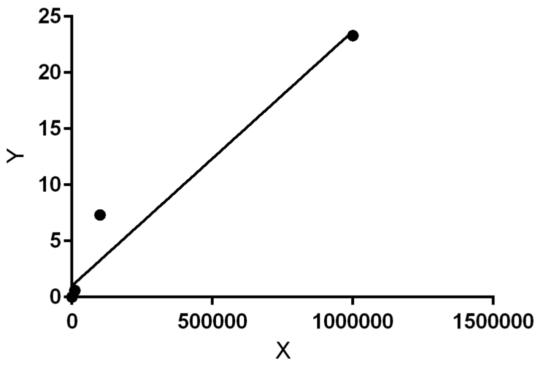


Random Seed: 312 Size: 500

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

Unsorted Array 0.055148s Min Heap 0.019284s [✓ Use Both] Heap is 2.860x Faster

n	Array T(n)	Heap T(n)	Faster
10	0.000203	0.000327	Array
100	0.0054	0.0055	Array
1000	0.1841	0.0424	Heap
10000	15.99	0.60	Heap
100000	200+	7.33	Heap
1000000		23.30	Heap

Array	Heap
	

As we can see on our graphs, we can see that the Array implementation of the Priority Queue gets to be more efficient than the Binary Min Heap implementation when the size of n is approximately n < 100. Once the size of n gets arbitrarily large the time complexity will increase radically.

Empirical Analysis:

In general, using a heap as a priority queue for Dijkstra's algorithm has a lower time complexity compared to using an array. The time complexity of Dijkstra's algorithm using an array as a priority queue is O(n^2), while the time complexity using a binary min heap as a priority queue is O((m+n)log(n)), where m is the number of edges in the graph and n is the number of nodes.

For small graphs (with n < 100), the difference in time complexity between using an array and a heap was not very significant. The array implementation was more optimal for this small cases. However, for the larger graphs with 1,000, 100,000 or

1,000,000 nodes, the time complexity difference was more pronounced, using an array at this point became prohibitively slow.

In general, it's recommended to use a binary min heap as a priority queue for Dijkstra's algorithm due to its lower time complexity. However, depending on the specific use case and constraints, an array implementation may still be better for small graphs or in situations where a simpler implementation is preferred over performance optimization.

Time and Space Complexity:

Dijkstra's Algorithm Array

- Insertion of an element will be O(1)
- Removal of the minimum will be O(n) since we need to transverse the whole array in order to find the new min

Dijkstra's Algorithm Heap

- Inserting an element will be O(log(n)) where n is the elements in the tree.
- Removing an element will also be O(log(n)) since every time we remove the root (minimum element in the heap) we need to take the farthest child from the tree and replace it in the root. Then we need to compare it with its immediate children and replace it with the smallest priority. If we swap we take the replaced node and repeat the process until our tree has the right property of a heap.
- Updating the priority of an element will be O(log(n)) because we need to find the node and that will mean we need to traverse our tree. Then, once we update our node, if the tree does not longer satisfy the properties of the Heap, then we need to bubble down the current node until the tree satisfy the property again. This means, O(log(n)).

Code:

```
#!/usr/bin/python3


from CS312Graph import *
import time


class NetworkRoutingSolver:
    def __init__(self):
        self.network = None
        self.source = None
        self.dest = None
        self.distances = {}
        self.previous = {}

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

    def getShortestPath(self, destIndex):
        assert(self.source is not None)

        self.dest = destIndex
        path_edges = []
        total_length = 0
        node = self.network.nodes[self.dest]
        while node.node_id != self.source:
            edge = self.previous[node.node_id]
            if edge is None:
                return {'cost': float('inf'), 'path': path_edges}
            path_edges.append((edge.src.loc, edge.dest.loc,
'{:.0f}'.format(edge.length)))
            total_length += edge.length
            node = edge.src
        path_edges.reverse()
        return {'cost': total_length, 'path': path_edges}

    def computeShortestPaths(self, srcIndex, use_heap=False):
        assert (self.network is not None)

        self.source = srcIndex
        self.distances = {node.node_id: float('inf') for node in self.network.nodes}
        self.distances[srcIndex] = 0
        self.previous = {node.node_id: None for node in self.network.nodes}
        visited = set()
        t1 = time.time()

        if use_heap:
            pq = BinaryMinHeap()
            pq.insertNode(srcIndex, 0)

            while pq.size() > 0:
                current_node_id, current_distance = pq.deleteMin()
                current_node = self.network.nodes[current_node_id]

                if current_node_id in visited:
                    continue

                visited.add(current_node_id)

                for edge in current_node.neighbors:
                    if edge.dest.node_id not in visited:
                        distance = self.distances[current_node_id] + edge.length
                        if distance < self.distances[edge.dest.node_id]:
                            self.distances[edge.dest.node_id] = distance
                            self.previous[edge.dest.node_id] = edge
                            pq.insertNode(edge.dest.node_id, distance)

        else:
            pq = PriorityQueue()
            pq.put(srcIndex, 0)

            while not pq.empty():
                current_node_id = pq.get()
                current_node = self.network.nodes[current_node_id]

                if current_node_id in visited:
                    continue

                visited.add(current_node_id)

                for edge in current_node.neighbors:
                    if edge.dest.node_id not in visited:
                        distance = self.distances[current_node_id] + edge.length
                        if distance < self.distances[edge.dest.node_id]:
                            self.distances[edge.dest.node_id] = distance
                            self.previous[edge.dest.node_id] = edge
                            pq.put(edge.dest.node_id, distance)

        t2 = time.time()
        return t2 - t1
```

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

    def empty(self):
        return len(self.queue) == 0

    def put(self, item, priority):
        self.queue.append((item, priority))
        self.queue.sort(key=lambda x: x[1])

    def get(self):
        return self.queue.pop(0)[0]


class BinaryMinHeap:
    def __init__(self):
        self.heap = []

    def insertNode(self, node, length):
        self.heap.append((node, length))
        self.bubbleUp(len(self.heap)-1)

    def getRightChild(self, position):
        if 2 * position + 1 < len(self.heap):
            return self.heap[2 * position + 1]
        return None

    def getLeftChild(self, position):
        if 2 * position < len(self.heap):
            return self.heap[2 * position]
        return None

    def getParent(self, position):
        return self.heap[position // 2]

    @staticmethod
    def getParentIndex(position):
        return position // 2

    @staticmethod
    def getRightChildIndex(position):
        return 2 * position + 1

    @staticmethod
    def getLeftChildIndex(position):
        return 2 * position

    def bubbleUp(self, i):
        p = BinaryMinHeap.getParentIndex(i)
        while i != 0 and self.heap[p][1] > self.heap[i][1]:
            self.heap[i], self.heap[p] = self.heap[p], self.heap[i]
            i = p
            p = BinaryMinHeap.getParentIndex(i)

    def pop(self):
        if len(self.heap) == 0:
            return None
        elif len(self.heap) == 1:
            return self.heap.pop()
        else:
            min_value = self.heap[0]
            self.heap[0] = self.heap.pop()
            self.siftDown(0)
            return min_value

    def decreaseKey(self, x, i):
        self.heap[i] = x
        self.bubbleUp(i)

    def deleteMin(self):
        if len(self.heap) == 0:
            return None
        elif len(self.heap) == 1:
            return self.heap.pop()
        else:
            min_value = self.heap[0]
            max_value = self.heap.pop()
            self.heap[0] = max_value
            self.siftDown(0)
            return min_value

    def siftDown(self, i):
        c = self.minChild(i)
        while c != 0 and self.heap[c][1] < self.heap[i][1]:
            self.heap[i], self.heap[c] = self.heap[c], self.heap[i]
            i = c
            c = self.minChild(i)

    def minChild(self, index):
        left_child_index = BinaryMinHeap.getLeftChildIndex(index)
        right_child_index = BinaryMinHeap.getRightChildIndex(index)
        if right_child_index < len(self.heap):
            if self.heap[left_child_index][1] < self.heap[right_child_index][1]:
                return left_child_index
            else:
                return right_child_index
        elif left_child_index < len(self.heap):
            return left_child_index
        else:
            return 0

    def size(self):
        return len(self.heap)

    def __str__(self):
        return self.heap.__str__()
```