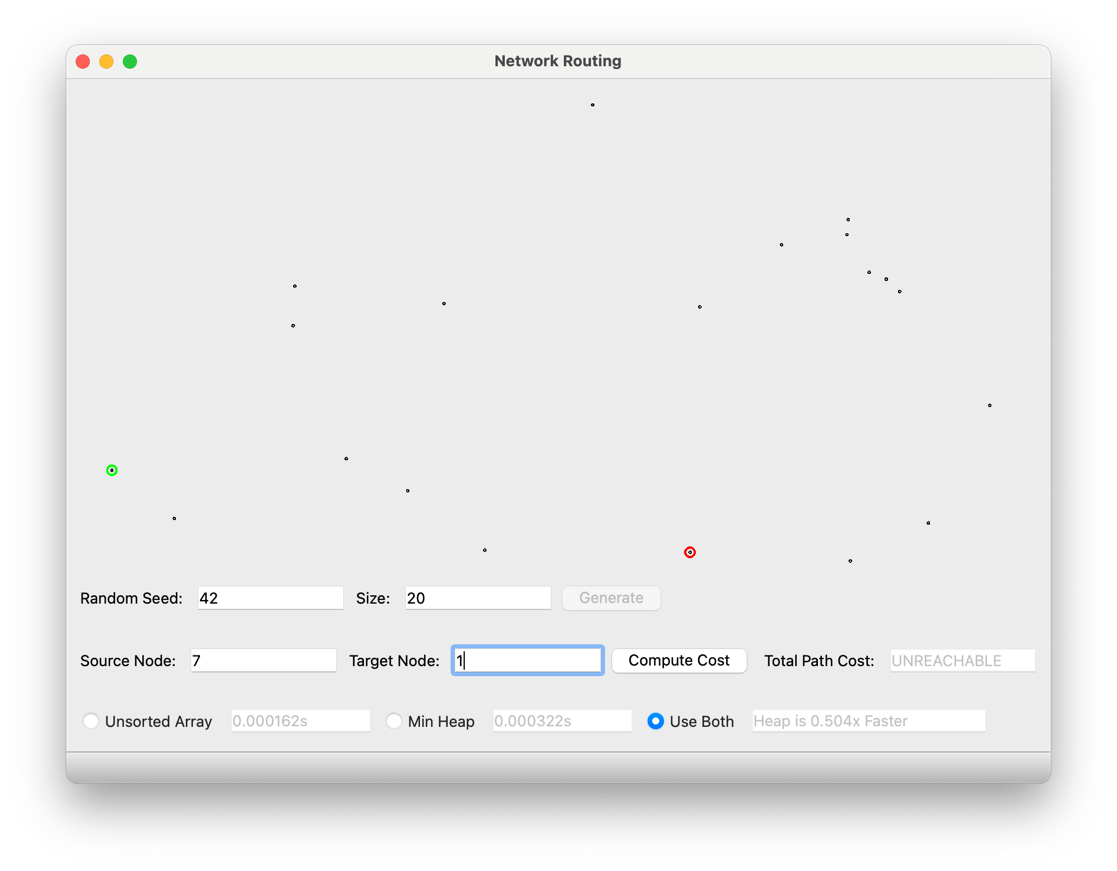
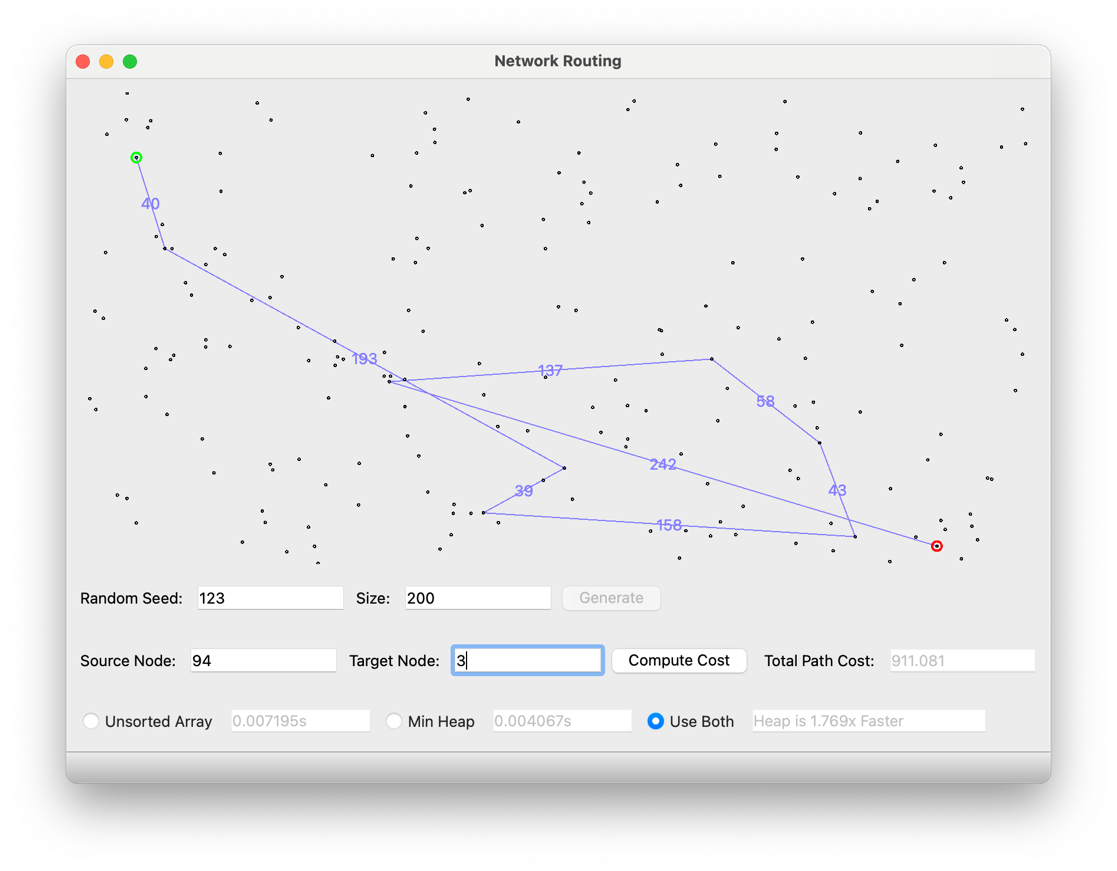
Network Routing Report

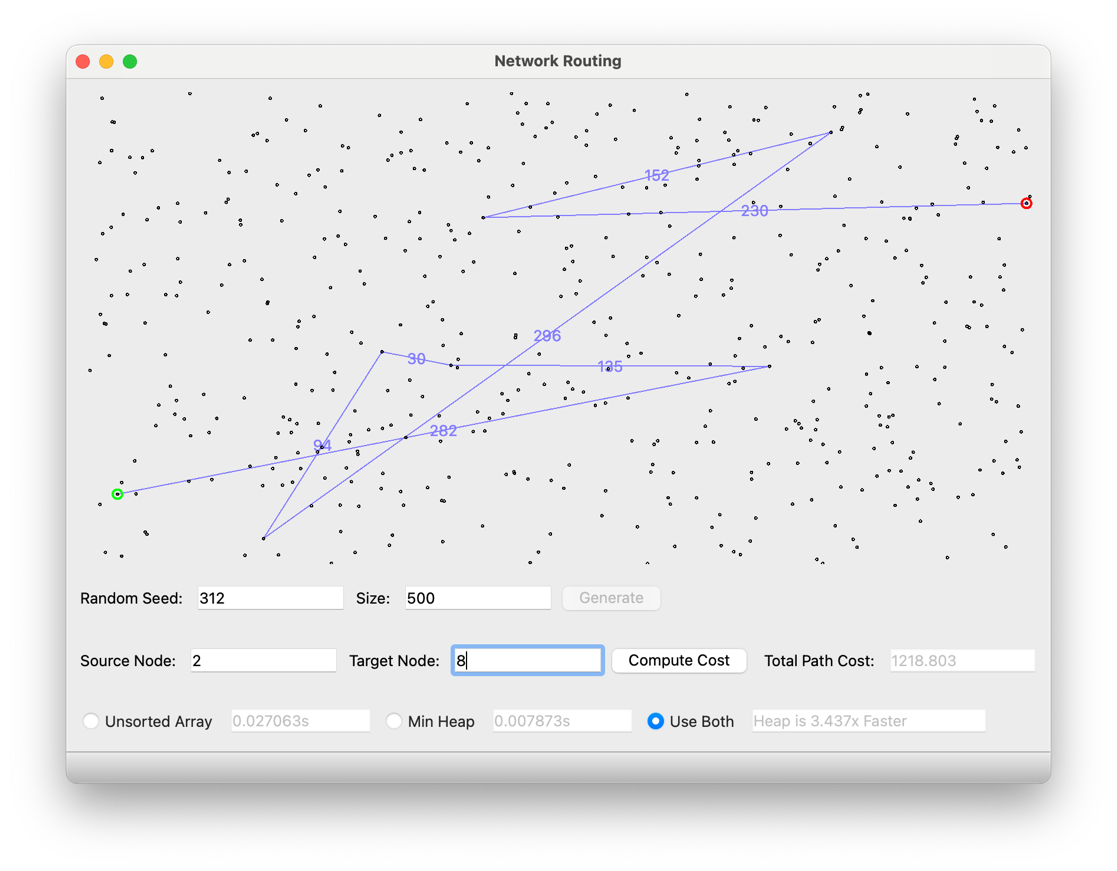
Niels Turley

1. **Correct functioning code**
2. **Correct implementation of array and binary heap**

#!/usr/bin/python3  
from BinaryHeap import BinaryHeap  
from CS312Graph import \*  
import time  
  
  
class NetworkRoutingSolver:  
 def \_\_init\_\_(self):  
 pass  
  
 def initializeNetwork(self, network):  
 assert (type(network) == CS312Graph)  
 self.network = network  
  
 def getShortestPath(self, destIndex):  
 self.dest = destIndex  
  
 begin\_node = self.network.nodes[self.source]  
 end\_node = self.network.nodes[self.dest]  
 path\_edges = []  
 total\_length = end\_node.dist  
  
 # Iterate back through the path to find the edges  
 if total\_length != float('inf'):  
 reached\_source = False  
 while not reached\_source:  
 next\_node = end\_node.path  
 path\_edges.append((end\_node.loc, next\_node.loc, '{:.0f}'.format(end\_node.dist - next\_node.dist)))  
 if next\_node == begin\_node:  
 reached\_source = True  
 else:  
 end\_node = next\_node  
 return {'cost': total\_length, 'path': path\_edges}  
  
 def computeShortestPaths(self, srcIndex, use\_heap=False):  
 self.source = srcIndex  
 t1 = time.time()  
  
 if use\_heap:  
 self.heap(srcIndex)  
 else:  
 self.array(srcIndex)  
  
 t2 = time.time()  
 return (t2 - t1)  
  
 """Time complexity: O(V^2). Inserting into the pq takes O(1) time, and we do this V times. Removing from the pq   
 takes O(V) time, and we do this V times. Changing the distance of a node takes O(1) time, and we do this E   
 times. Thus, O( (V\*1) + (V\*V) + (E\*1) ) = O(V^2)."""  
 """Space complexity: O(V). We create a pq of size V to store the nodes."""  
 def array(self, srcIndex):  
 # Initialize the array  
 pq = []  
 for node in self.network.nodes:  
 node.dist = float('inf')  
 node.known = False  
 node.path = None  
 self.network.nodes[srcIndex].dist = 0  
 pq.extend(self.network.nodes) # O(1) time (insert)  
  
 # Run Dijkstra's algorithm  
 while len(pq) > 0:  
 # Find the node with the smallest distance  
 min\_dist = float('inf')  
 min\_node = pq[0]  
 for node in self.network.nodes: # O(V) time (delete\_min)  
 if not node.known and node.dist < min\_dist:  
 min\_dist = node.dist  
 min\_node = node  
  
 pq.remove(min\_node)  
  
 # Update the distances of the neighbors  
 for edge in min\_node.neighbors:  
 if edge.dest.dist > min\_node.dist + edge.length:  
 edge.dest.dist = min\_node.dist + edge.length # O(1) time (decrease\_key)  
 edge.dest.path = min\_node  
 min\_node.known = True  
  
 """Time complexity: O( (V + E) \* log(V)). Inserting into the pq takes O(log(V)) time, and we do this V times.   
 Removing from the pq takes O(log(V)) time, and we do this V times. Changing the distance of a node takes O(log(  
 V)) time, and we do this E times. Thus, O( (V\*log(V)) + (V\*log(V)) + (E\*log(V)) ) = O( (V + E) \* log(V))."""  
 """Space complexity: O(V). We create a pointer array of size V to point to the nodes in the heap, and we create a   
 binary tree of size V to store the nodes. O(V) + O(V) = O(V)."""  
 def heap(self, srcIndex):  
 # Initialize the binary heap  
 heap = BinaryHeap()  
 for node in self.network.nodes:  
 node.dist = float('inf')  
 node.known = False  
 node.path = None  
 heap.push((node.node\_id, node.dist)) # O(log(V) time (insert)  
 self.network.nodes[srcIndex].dist = 0  
 heap.update(srcIndex, 0)  
  
 # Run Dijkstra's algorithm  
 while heap.size > 0:  
 # Pop the node with the smallest distance  
 (node\_id, index) = heap.pop() # O(log(V)) time (delete\_min)  
  
 node = self.network.nodes[node\_id]  
  
 # Update the distances of the neighbors  
 for edge in node.neighbors:  
 if edge.dest.dist > node.dist + edge.length:  
 edge.dest.dist = node.dist + edge.length  
 edge.dest.path = node  
 heap.update(edge.dest.node\_id, edge.dest.dist) # O(log(V)) time (decrease\_key)

class BinaryHeap:  
 def \_\_init\_\_(self):  
 self.heap = [] # array of tuples (node\_id, index)  
 self.pointer\_array = {} # maps node\_id to index in heap  
 self.size = 0  
  
 def push(self, item):  
 self.heap.append(item) # item is a tuple (node\_id, index)  
 self.size += 1  
 tree\_index = self.bubble\_up(self.size - 1)  
 self.pointer\_array[item[0]] = tree\_index  
  
 def pop(self):  
 if self.size == 0:  
 return None  
  
 # Swap the root with the last element  
 root = self.heap[0]  
 self.pointer\_array[root[0]] = None  
 self.heap[0] = self.heap[self.size - 1]  
 self.pointer\_array[self.heap[0][0]] = 0  
 self.heap.pop()  
 self.size -= 1  
 self.bubble\_down(0)  
  
 return root  
  
 def bubble\_up(self, i):  
 while i > 0 and self.heap[i][1] < self.heap[self.parent(i)][1]:  
 self.swap(i, self.parent(i))  
 i = self.parent(i)  
 return i  
  
 def bubble\_down(self, i):  
 min\_index = i  
  
 # Recursively bubble down the smaller child  
 l = self.left(i)  
 if l < self.size and self.heap[l][1] < self.heap[min\_index][1]:  
 min\_index = l  
  
 r = self.right(i)  
 if r < self.size and self.heap[r][1] < self.heap[min\_index][1]:  
 min\_index = r  
  
 if i != min\_index:  
 self.swap(i, min\_index)  
 self.bubble\_down(min\_index)  
  
 def parent(self, i):  
 return (i - 1) // 2  
  
 def left(self, i):  
 return 2 \* i + 1  
  
 def right(self, i):  
 return 2 \* i + 2  
  
 def swap(self, i, j):  
 self.heap[i], self.heap[j] = self.heap[j], self.heap[i]  
 self.pointer\_array[self.heap[i][0]] = i  
 self.pointer\_array[self.heap[j][0]] = j  
  
 def update(self, index, dist):  
 # Use the pointer array to find the index in the heap, then update the heap  
 tree\_index = self.pointer\_array[index]  
 self.heap[tree\_index] = (self.heap[tree\_index][0], dist)  
 self.bubble\_up(tree\_index)  
 self.bubble\_down(tree\_index)

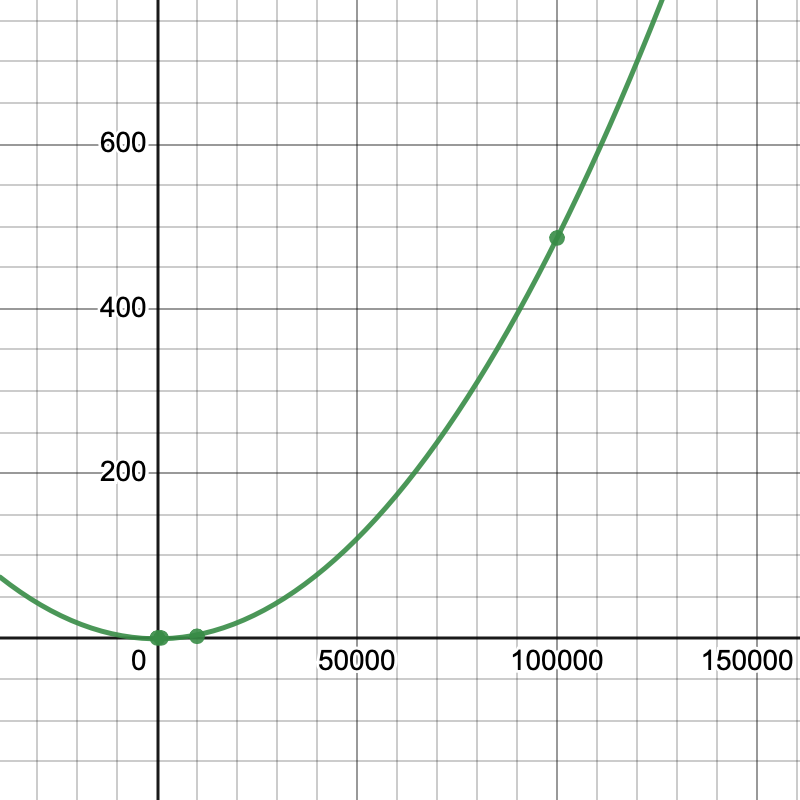
1. **Discuss the time and space complexity**
   1. array()
      1. Time complexity: O(V^2). An array is easy to index into and add on to the end. However, it takes time to find the minimum node in it since it isn’t sorted. Inserting into the pq takes O(1) time, and we do this V times. Removing the minimum node from the pq takes O(V) time since it has to go through all of them to find the minimum, and we do this V times. Changing the distance of a node takes O(1) time, and we do this E times. Thus, O( (V\*1) + (V\*V) + (E\*1) ) = O(V^2).
      2. Space complexity: O(V). We create a pq of size V to store the nodes.
   2. heap()
      1. Time complexity: O( (V + E) \* log(V)). Since we are using a binary heap to keep track of the nodes, every operation will need to iterate down or up the depth of the tree to keep the tree organized—in this case, log2 since it’s a binary heap. Inserting into the pq takes O(log(V)) time, and we do this V times. Removing from the pq takes O(log(V)) time, and we do this V times. Changing the distance of a node takes O(log(V)) time, and we do this E times. Thus, O( (V\*log(V)) + (V\*log(V)) + (E\*log(V)) ) = O( (V + E) \* log(V)).
      2. Space complexity: O(V). We create a pointer array of size V to point to the nodes in the heap, and we create a binary tree of size V to store the nodes. O(V) + O(V) = O(V).
2. **Screenshots**
   1. **Seed 42, Size 20, Source 7, Dest 7**
   2. **Seed 123, Size 200, Source 94, Dest 3**
   3. **Seed 312, Size 500, Source 2, Dest 8**

****

1. **Comparison of time complexity**

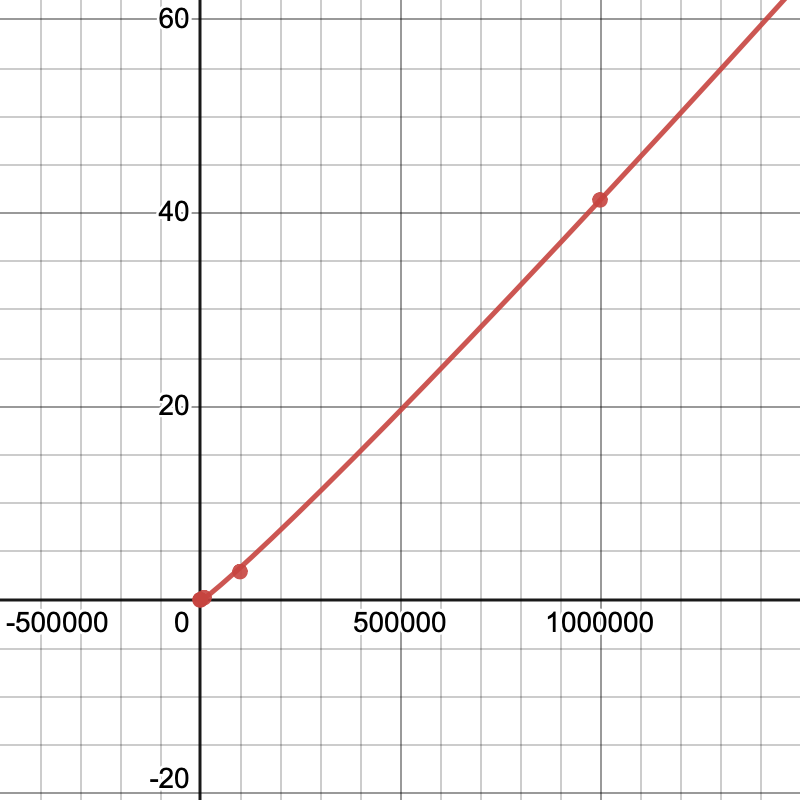
Array implementation

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | 100 | 1,000 | 10,000 | 100,000 | 1,000,000 |
| 1 | 0.000957 | 0.030395 | 1.952750 | 482.771369 | X |
| 2 | 0.001039 | 0.031402 | 2.011900 | 462.470715 | X |
| 3 | 0.001325 | 0.029792 | 2.040490 | 496.039915 | X |
| 4 | 0.000860 | 0.029745 | 2.117524 | 498.887080 | X |
| 5 | 0.000720 | 0.030875 | 2.224730 | 488.692089 | X |
| Average | 0.000980 | 0.030442 | 2.069479 | 485.7722336 | X |



Regression line plotted of form , with and

Estimated time for 1,000,000: 48667.6536543s

Binary heap implementation

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | 100 | 1,000 | 10,000 | 100,000 | 1,000,000 |
| 1 | 0.002384 | 0.017502 | 0.244154 | 2.863683 | 38.005387 |
| 2 | 0.002325 | 0.017431 | 0.240479 | 2.601054 | 47.896496 |
| 3 | 0.002381 | 0.016576 | 0.235332 | 3.164868 | 38.922685 |
| 4 | 0.002330 | 0.016677 | 0.233347 | 2.990265 | 38.013725 |
| 5 | 0.002343 | 0.016874 | 0.235511 | 3.018648 | 43.984187 |
| Average | 0.002353 | 0.017012 | 0.237565 | 2.927704 | 41.364496 |

Regression line plotted of form , with and

Discussion of the difference:

Although the array implementation has an insert and decrease\_key complexity of O(1) while the binary heap has O(logV), the biggest difference (in this case) comes down to the delete\_min. The array has to sort through the entire array to find the minimum distance node with O(V) time (total being O(V^2)) while the binary heap only has to pop off the root and bubble down to reset the tree in O(log V) time (total being O((V + E) log V). This means that the binary heap is faster in cases where the graph is sparse, or where there aren’t many edges between nodes (like the one we implemented).

However, when the graph is dense and there is close to the maximum number of edges (meaning that E is roughly equal to V^2), the array implementation can actually be faster. Its total time complexity is still O(V^2) since it still has to go through all of the nodes to delete the minimum one, which it does V times. Changing the distance of a node, while now being roughly equal to V^2 \* 1, isn’t any more dominating than it was in a sparse graph. With the binary heap, though, the time complexity is now O((V + E) log V) = O((V^2) log V), since we now have to change the distance of a node V^2 times, and it takes log(V) time.