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  
 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()  
  
 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  
 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):  
 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):  
 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  
  
 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:

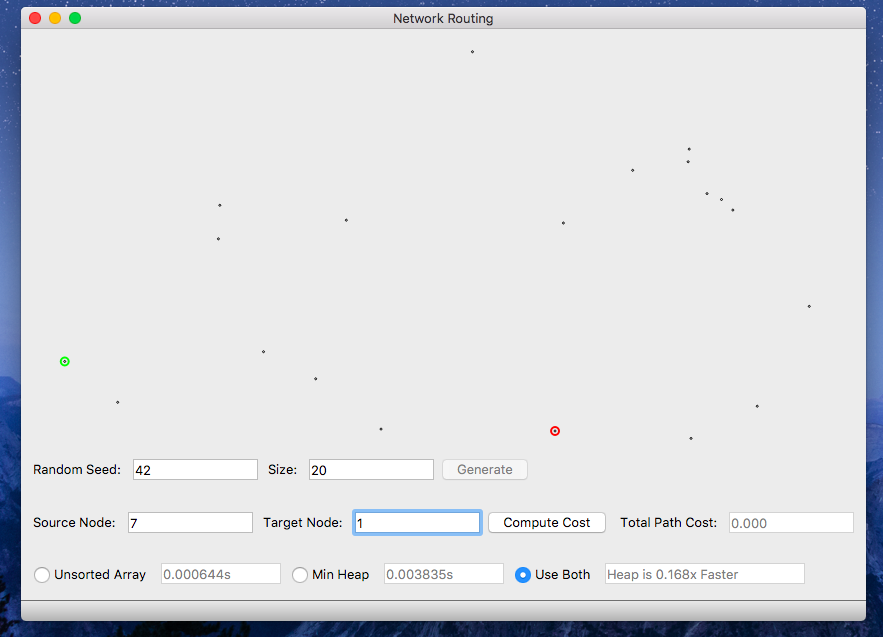
The algorithm keeps 2 arrays of size n for total distances and previous nodes thus O(2n)

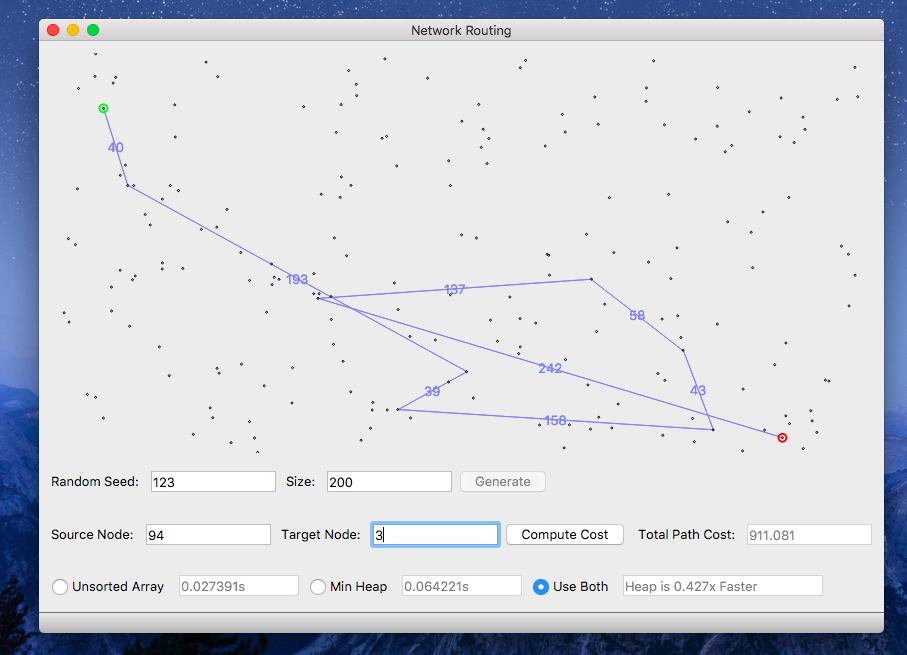
Array: The array implementation stores another 3 arrays of size n for the graph, nodes, and popped nodes. Thus total space is O(5n)

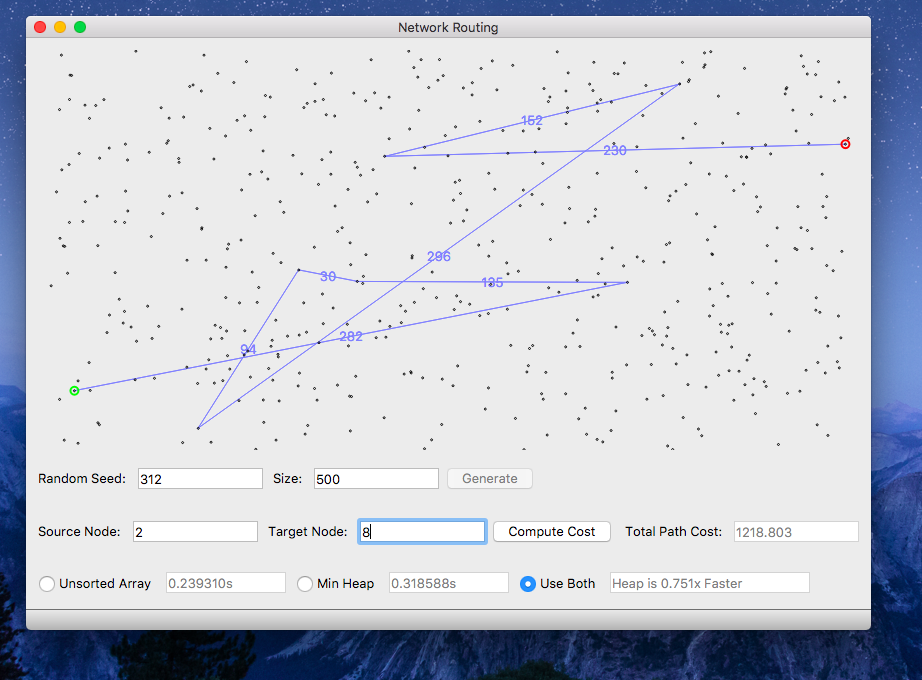
Heap: The heap implementation also stores another 3 arrays. Thus is is the same space complexity O(5n).

Screenshots:

There is no path.







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 |  |  |  |  |  |  |

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: