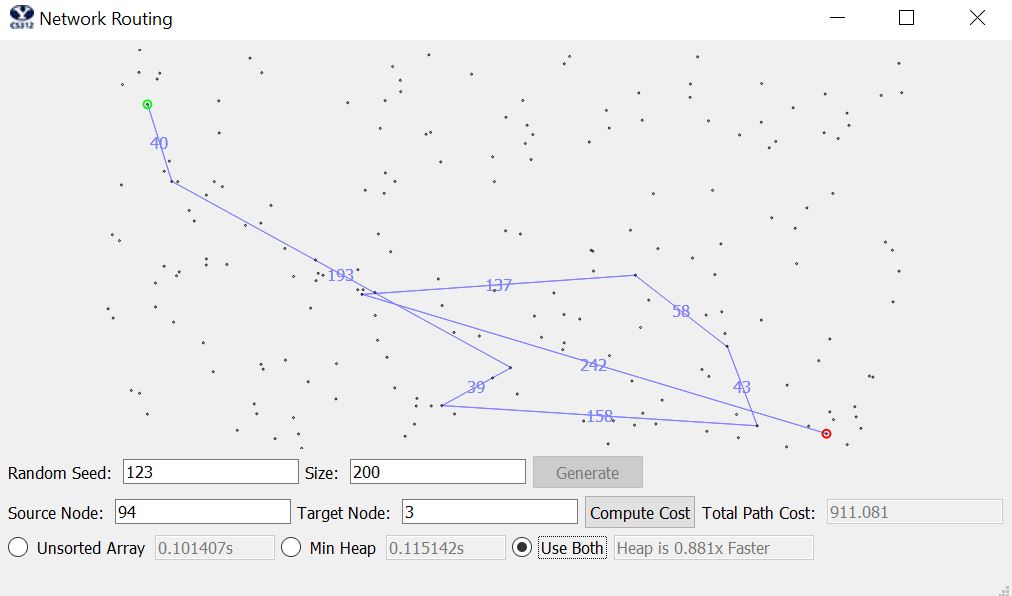
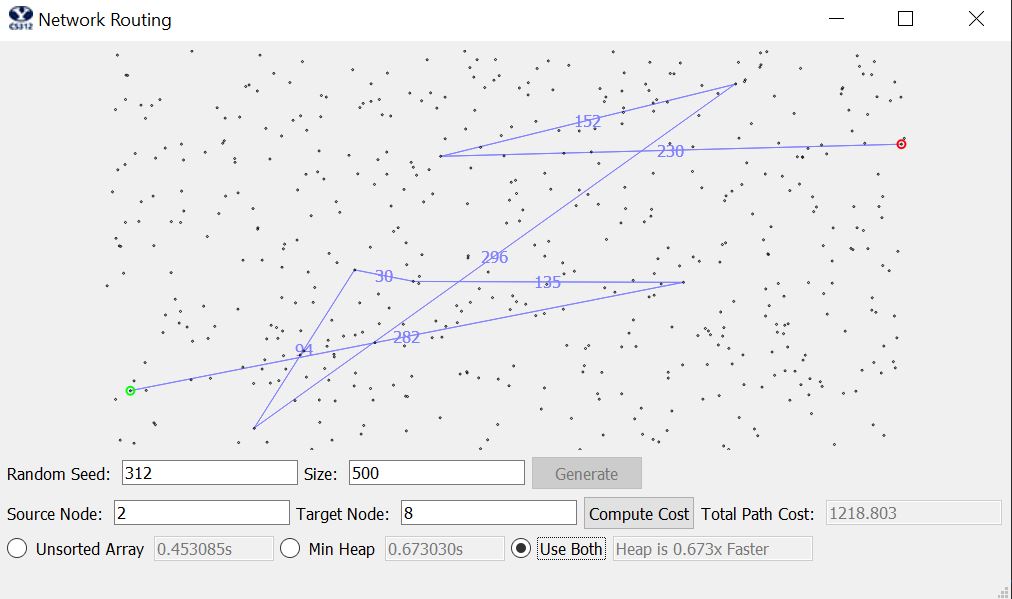
class NetworkRoutingSolver:  
 def \_\_init\_\_( self ):  
 pass  
  
 def initializeNetwork( self, network ):  
 assert( type(network) == CS312Graph )  
 self.network = network  
  
 def getShortestPath( self, dest\_index ):  
 self.dest = dest\_index  
  
 # *TODO: RETURN THE SHORTEST PATH FOR destIndex* # *INSTEAD OF THE DUMMY SET OF EDGES BELOW* # *IT'S JUST AN EXAMPLE OF THE FORMAT YOU'LL* # *NEED TO USE* path\_edges = []  
 total\_length = 0  
  
 # Setting total length and edges to return for the destination index  
 # Takes O(n) time and O(n) space to build an array  
 self.current\_index = self.queue.nodes[self.dest]  
 while True:  
 if self.current\_index.node\_id == self.source:  
 break  
 current\_loc = self.current\_index.loc  
 previous = self.queue.get\_previous(self.current\_index)  
 if previous is None:  
 break  
 weight = self.queue.get\_previous\_weight(self.current\_index)  
 path\_edges.append((current\_loc, previous.loc, '{:.0f}'.format(weight)))  
 self.current\_index = self.queue.get\_previous(self.current\_index)  
 total\_length = total\_length + weight  
  
 return {'cost': total\_length, 'path': path\_edges}  
  
 def computeShortestPaths( self, srcIndex, use\_heap=False ):  
 self.source = srcIndex  
 t1 = time.time()  
  
 # To check if user wants to use binary heap or unsorted array  
 if use\_heap:  
 self.queue = BinHeap(self.network, srcIndex)  
 else:  
 self.queue = Unsorted\_Array(self.network, srcIndex)  
  
 # Implementation of Dijkstra's Algorithm  
 # The overall algorithm would take O((V+E)logV) time for the heap implementation and O(V^2) time for the array.  
 # For both implementations, the space complexity would be O(V) since they both use array to store the weights  
 # of every node.  
 while not self.queue.isEmpty():  
 u = self.queue.delMin()  
 if u == -1:  
 break  
 if u.node\_id != 0:  
 self.queue.building\_queue(u)  
 adjacents = u.neighbors  
 for i in range(0, len(adjacents)):  
 if self.queue.get\_weight(adjacents[i].dest) > self.queue.get\_weight(u) + adjacents[i].length:  
 self.queue.set\_weight(adjacents[i].dest, self.queue.get\_weight(u) + adjacents[i].length)  
 self.queue.set\_previous(adjacents[i].dest, u)  
 t2 = time.time()  
 return t2-t1  
  
# Implementation of unsorted array  
class Unsorted\_Array:  
 def \_\_init\_\_(self, graph, src):  
 self.graph = graph  
 self.nodes = graph.getNodes()  
 self.node\_weight = [sys.maxsize - 1] \* len(self.nodes)  
 self.previous\_node = [None] \* len(self.nodes)  
 self.deletedNodes = []  
 self.unsorted\_array = []  
 self.unsorted\_array.append(CS312GraphEdge(self.nodes[src], self.nodes[src], 0))  
 self.node\_weight[src] = 0  
 self.previous\_node[src] = self.nodes[src]  
 self.building\_queue(self.nodes[src])  
  
 # Making a queue takes O(n) time for array  
 def building\_queue(self, node):  
 for i in range(0,len(node.neighbors)):  
 if node.neighbors[i].dest.node\_id not in self.deletedNodes:  
 self.unsorted\_array.append(node.neighbors[i])  
  
 # Takes O(n) time and space complexity  
 def delMin(self):  
 temp = sys.maxsize  
 this\_node = -1  
 index = -1  
 for i in range(len(self.unsorted\_array)):  
 if self.node\_weight[self.unsorted\_array[i].dest.node\_id] < temp and \  
 self.unsorted\_array[i].dest.node\_id not in self.deletedNodes:  
 temp = self.node\_weight[self.unsorted\_array[i].dest.node\_id]  
 this\_node = self.unsorted\_array[i].dest.node\_id  
 index = i  
  
 if this\_node == -1:  
 return -1  
 else:  
 retVal = self.unsorted\_array[index].dest  
 self.deletedNodes.append(retVal.node\_id)  
 self.unsorted\_array.pop(index)  
 return retVal  
  
  
 # return false if the queue is not empty  
 def isEmpty(self):  
 if len(self.unsorted\_array) > 0:  
 return False  
 return True  
  
 # O(n) Helper functions  
 def set\_previous(self, node, prev\_node):  
 self.previous\_node[node.node\_id] = prev\_node.node\_id  
  
 def set\_weight(self, node, distance):  
 self.node\_weight[node.node\_id] = distance  
  
 def get\_previous(self, node):  
 if self.previous\_node[node.node\_id] is not None:  
 return self.nodes[self.previous\_node[node.node\_id]]  
  
 def get\_weight(self, node):  
 return self.node\_weight[node.node\_id]  
  
 def get\_previous\_weight(self, node):  
 if self.previous\_node[node.node\_id] is not None:  
 return self.node\_weight[node.node\_id] - self.node\_weight[self.previous\_node[node.node\_id]]  
 return 0  
  
  
  
# Implementation of binary heap data structure  
class BinHeap:  
 def \_\_init\_\_(self, graph, srcIndex):  
 self.nodes = graph.nodes  
 self.currentSize = 0  
 self.heapList = []  
 self.insert(CS312GraphEdge(self.nodes[srcIndex], self.nodes[srcIndex], 0))  
  
 self.node\_weight = [sys.maxsize - 1] \* len(self.nodes)  
 self.previous\_node = [None] \* len(self.nodes)  
  
 self.deletedNodes = list()  
 self.node\_weight[srcIndex] = 0  
 self.previous\_node[srcIndex] = self.nodes[srcIndex]  
  
 self.building\_queue(self.nodes[srcIndex])  
  
 # Making a heap takes O(nlog n) time and O(n) space complexity because inserting a node in the middle may require  
 # O(n) operations to shift the rest  
 def building\_queue(self, node):  
 for i in range(len(node.neighbors)):  
 if node.neighbors[i].dest.node\_id not in self.deletedNodes:  
 self.insert(node.neighbors[i])  
 self.bubbleUp(self.currentSize - 1)  
  
 def insert(self, u):  
 self.heapList.append(u)  
 self.currentSize += 1  
  
 # This part takes O(log n) time and O(n) space to maintain the heap property  
 # Dividing the index by 2 gives the parent node  
 def bubbleUp(self, i):  
 while i // 2 > 0:  
 if self.node\_weight[self.heapList[i].dest.node\_id] < self.node\_weight[self.heapList[i // 2].dest.node\_id]:  
 temp = self.heapList[i // 2]  
 self.heapList[i // 2] = self.heapList[i]  
 self.heapList[i] = temp  
 i = i // 2  
  
 # The smallest should go to the top of the tree to maintain the heap property  
 # Takes O(log n) time and O(n) space complexity  
 def delMin(self):  
 if self.isEmpty():  
 return -1  
 retVal = self.heapList[0].dest  
 self.heapList[0] = self.heapList[-1]  
 self.deletedNodes.append(retVal.node\_id)  
 self.currentSize = self.currentSize - 1  
 self.heapList.pop()  
 self.trickle\_down(0)  
 return retVal  
  
 # This part takes O(log n) time and O(n) space complexity  
 def trickle\_down(self, i):  
 if not self.isEmpty():  
 while i \* 2 <= self.currentSize:  
 minimumChild = self.minChild(i)  
 if minimumChild == -1:  
 break  
 if self.node\_weight[self.heapList[i].dest.node\_id] > self.node\_weight[self.heapList[minimumChild].dest.node\_id]:  
 temp = self.heapList[i]  
 self.heapList[i] = self.heapList[minimumChild]  
 self.heapList[minimumChild] = temp  
 i = minimumChild  
  
 def minChild(self, i):  
 if i \* 2 + 1 >= self.currentSize:  
 return -1  
 elif i \* 2 + 2 >= self.currentSize:  
 return i \* 2 + 1  
 else:  
 if self.node\_weight[self.heapList[i\*2+1].dest.node\_id] < self.node\_weight[self.heapList[i\*2+2].dest.node\_id]:  
 return i\*2+1  
 else:  
 return i\*2+2  
  
 # return false if the queue is not empty  
 def isEmpty(self):  
 if self.currentSize > 0:  
 return False  
 return True  
  
 # O(n) Helper functions  
 def get\_weight(self, node):  
 return self.node\_weight[node.node\_id]  
  
 def get\_previous(self, node):  
 if self.previous\_node[node.node\_id] is not None:  
 return self.nodes[self.previous\_node[node.node\_id]]  
  
 def get\_previous\_weight(self, node):  
 if self.previous\_node[node.node\_id] is not None:  
 return self.node\_weight[node.node\_id] - self.node\_weight[self.previous\_node[node.node\_id]]  
 return 0  
  
 def set\_weight(self, node, distance):  
 self.node\_weight[node.node\_id] = distance  
  
 def set\_previous(self, node, prev\_node):  
 self.previous\_node[node.node\_id] = prev\_node.node\_id

3. The overall time complexity for binary heap would be O((V+E)logV) because there is one making queue, and V times of deleting the minimum, and E times of decreasing key and each takes O(nlogn), O(logn), and O(logn). For space complexity, it would be O(n) since it’s using arrays to store the needed data.

The overall time complexity for unsorted array would be O(V^2 + E). We know each nodes has 3 neighbors, so it can be denoted as O(V^2). Like the heap implementation, it takes one making queue, V times of deleting the minimum, and E times of decreasing the key, which take O(n), O(n), O(1) times each. Also, its space complexity is O(n) since it uses arrays to store and keep track of the data.

4. A - was not reachable

B - 

C - 

5. 1 2 3 4 5 Average

100 Heap 0.13 0.06 0.15 0.07 0.17 0.12

Array 0.02 0.02 0.03 0.02 0.02 0.02

1000 Heap 1.19 1.77 1.48 1.82 2.06 1.67

Array 2.69 2.34 2.46 2.6 2.62 2.54

10000 Heap 68.6 70.2 68.01 67.08 68.42 68.46

Array couldn’t figure out

It seems the code takes particularly long to find the shortest paths and I couldn’t really create the whole table.