Trevor Ashby

Network Routing

|  |
| --- |
| 1. Code Submission |
| \*\*See appendix |

|  |  |
| --- | --- |
| 1. Priority Queue’s Complexity | |
| Array | Heap |
| **Insert():** I do not use an insert function with the array implementation, but adding something to the array is O(1) complexity because appending to the end of an array takes constant time.  **Deletemin\_Array():** The complexity is O(n) because every node is traversed to determine which has the lowest value. | **Insert\_heap():** Appending to the end of an array takes O(1) time. Afterwards, checking to make sure that the array is sorted is O(logn) due to visiting a node at each level during the “bubbling up”stage to rebalance the tree.  **DecreaseKey():** Mine has worst case scenario Complexity O(n). This is because I implemented my pointerArray in an inverse way \*unintentionally\*. However, this is the incorrect implementation. To fix this, I would have the indices of the pointerArray be related to the nodeID’s, which values would then point to the position of the nodeID in the BinaryHeap array. This would allow me to use the nodeID for a lookup of O(1) time. This would greatly improve speeds.  **Deletemin\_Heap():** Popping off values from the arrays is O(1) time. However, once the last value becomes gets put onto the front of the array (or top of the tree depending on how you look at it), the “sifting down” will take O(logn) due to having to revisit a node at each layer. Thus, rebalancing the tree.  **Swap():** Complexity O(1)  **MakeQueue():** O(1)  **Sift\_down():** O(logn) |

|  |  |
| --- | --- |
| 1. Time and Space Complexity | |
| Array | Heap |
| **Time Complexity:** O(n^2). This is because we run “n” times for Dijkstra’s algorithm for each node. Then each time we run deletemin is called, which is also O(n) complexity, therefore we have O(n^2) complexity.  **Space Complexity:** O(n) Because for each array that is being stored, “n” space is required. But due to the max rule, we reduce to O(n). | **Time Complexity:** The time complexity should be O(nlogn). Because we run for “n” times (in Dijkstra’s, but for each run we also use deletemin which \*should be\* O(logn). Therefore giving us O(nlogn). However, because my implementation is not correct, our worst case scenario will be O(n^2).  **Space Complexity:** O(n) Because for each array that is being stored, “n” space is required. But due to the max rule, we reduce to O(n). |

|  |  |
| --- | --- |
| 1. Screenshots | |
| 1. I am not including a screenshot of this because there is no path between the two points. | |
| 1. The GUI was not working for me when I did “use both”. It would not report the Min Heap time, so I am including pictures of both individually so that time comparisons can be made. | |
|  |  |
| 1. The GUI was not working for me when I did “use both”. It would not report the Min Heap time, so I am including pictures of both individually so that time comparisons can be made. | |
|  |  |

|  |
| --- |
| 1. Empirical Analysis |
| |  |  |  |  | | --- | --- | --- | --- | | Node Count | Array Time | Heap Time | | | 100 | 0.003987 | 0.001994 | | | 100 | 0.003950 | 0.003986 | | | 100 | 0.003985 | 0.004989 | | | 100 | 0.004984 | 0.004984 | | | 100 | 0.000997 | 0.00202 | | | Average: |  |  | | | 1000 | 0.138628 | 0.03590 | | | 1000 | 0.104744 | 0.035903 | | | 1000 | 0.127666 | 0.035904 | | | 1000 | 0.103722 | 0.033934 | | | 1000 | 0.103704 | 0.038879 | | | Average: |  |  | | | 10,000 | 12.257217 | 2.387613 | | | 10,000 | 11.144193 | 3.111657 | | | 10,000 | 10.987611 | 2.690777 | | | 10,000 | 11.250908 | 2.596055 | | | 10,000 | 11.976964 | 2.983012 | | | Average: | | |  | | |

|  |
| --- |
| Appendix |
| from CS312Graph import \* import time import numpy as np import collections  class NetworkRoutingSolver:  def \_\_init\_\_( self):  pass   def initializeNetwork( self, network ):  assert( type(network) == CS312Graph )  self.network = network  self.dist = [np.inf] \* len(self.network.nodes)  self.prev = [None] \* len(self.network.nodes)  self.binaryHeap = []  self.pointerArray = []   def getShortestPath( self, destIndex ):  self.dest = destIndex  path\_edges = []  total\_length = 0   node = self.network.nodes[self.dest]  while node.node\_id != self.source:  prevNode = node  node = self.network.nodes[self.prev[node.node\_id]] # <--- this will not work for HEAP because here is where "prev" is used  for edge in node.neighbors:  if edge.dest.node\_id == prevNode.node\_id:  path\_edges.append( (edge.src.loc, edge.dest.loc, '{:.0f}'.format(edge.length)) )  total\_length += edge.length   return {'cost':total\_length, 'path':path\_edges}   #######################  # Time Complexity: O(1)  #######################  def makequeue(self):  # populate heap array   # SET VALUES TO DIST AND PREV  startNode = self.network.nodes[self.source]  self.insert\_heap(0, startNode)  for edge in startNode.neighbors:  self.insert\_heap(edge.length, edge.dest)   return   #######################  # Time Complexity: O(logn)  #######################  def insert\_heap(self, distance, nodeIn):  index = len(self.binaryHeap)  self.binaryHeap.append(distance)  self.pointerArray.append(nodeIn.node\_id)  while (index != 0):  p = self.parent(index)  if (self.binaryHeap[p] > self.binaryHeap[index]):  self.swap(p, index)  index = p   def parent(self, i):  return (i - 1)//2   def left\_child(self, i):  return 2\*i + 1   def right\_child(self, i):  return 2\*i + 2   #######################  # Time Complexity: O(1)  #######################  def swap(self, i, j):  self.binaryHeap[i], self.binaryHeap[j] = self.binaryHeap[j], self.binaryHeap[i]  self.pointerArray[i], self.pointerArray[j] = self.pointerArray[j], self.pointerArray[i]   #######################  # Time Complexity: O(n)  #######################  def deletemin\_array(self, tempNodes):  # return lowest value of array  lowestDistance = np.inf  lowestIndex = -1  nodeId = -1   for i in range(len(tempNodes)):  if lowestDistance > self.dist[tempNodes[i].node\_id]:  lowestDistance = self.dist[tempNodes[i].node\_id]  lowestIndex = i  nodeId = tempNodes[i].node\_id  if lowestIndex != -1:  del tempNodes[lowestIndex]   return nodeId   #######################  # Time Complexity: O(n) should be O(logn  #######################  def decreaseKey(self, nodeIdToDecrease, updateDistance):  for i in range(len(self.pointerArray)): # <---------------- CAN BE SPED UP HERE  if nodeIdToDecrease == self.pointerArray[i]:  self.binaryHeap[i] = updateDistance  # change the value in heap to updateDistance  # check if parent is bigger, if so, swap.  index = i  while (index != 0):  p = self.parent(index)  if (self.binaryHeap[p] > self.binaryHeap[index]):  self.swap(p, index)  index = p  break  return   #########################  # Time Complexity: O(logn)  #########################  def sift\_down(self):  # check right or left and replace with whichever one is smaller  # continue until either null child or at bottom  # if none less, then end  index = 0  changed = True  while(changed):  # if both null, break  if (self.right\_child(index) > len(self.binaryHeap)-1 and self.left\_child(index) > len(self.binaryHeap)-1):  break  # null right, check left. if less, swap  elif (self.right\_child(index) > len(self.binaryHeap)-1):  if (self.binaryHeap[self.left\_child(index)] < self.binaryHeap[index]):  self.swap(self.left\_child(index), index)  index = self.left\_child(index)  else:  break  # if left < right, check left. if less, swap  elif (self.binaryHeap[self.left\_child(index)] < self.binaryHeap[self.right\_child(index)]):  if (self.binaryHeap[self.left\_child(index)] < self.binaryHeap[index]):  self.swap(self.left\_child(index), index)  index = self.left\_child(index)  else:  break  # if left > right, check right. if less, swap  elif (self.binaryHeap[self.left\_child(index)] > self.binaryHeap[self.right\_child(index)]):  if (self.binaryHeap[self.right\_child(index)] < self.binaryHeap[index]):  self.swap(self.right\_child(index), index)  index = self.right\_child(index)  else:  break  changed = False  return   ########################  # Time Complexity: O(logn)  ########################  def deletemin\_heap(self):  # pop the top off of both arrays  if (len(self.binaryHeap) == 0):  return  else:   distance = self.binaryHeap.pop(0)  i = self.pointerArray.pop(0)  node = self.network.nodes[i]   # set the last value to the first  if (len(self.binaryHeap) != 0):  bHtoAppend = self.binaryHeap.pop(len(self.binaryHeap)-1)  self.binaryHeap.insert(0, bHtoAppend)  pAtoAppend = self.pointerArray.pop(len(self.pointerArray)-1)  self.pointerArray.insert(0, pAtoAppend)   # sift down until ordered properly  self.sift\_down()   return node, distance   # used to populate arrays  ###############################  # Time Complexity Array: O(n^2)  # Time Complexity Heap: worst case O(n^2) should be O(nlogn)  ###############################  def computeShortestPaths( self, srcIndex, use\_heap=False ):  # if not use heap, do priority queue  self.source = srcIndex  if use\_heap == False:  t1 = time.time()   # set start distance as 0 and prev as None  self.dist[self.source] = 0  i = self.source  self.prev[self.source] = None   tempNodes = self.network.nodes[:]  del tempNodes[self.source]   for indx in range(len(self.network.nodes)):  node = self.network.nodes[i] # begin with starting node  for edge in node.neighbors: # for all edges(u, v) in E:  alt = self.dist[i] + edge.length  if (self.dist[edge.dest.node\_id] > alt):  self.dist[edge.dest.node\_id] = alt  self.prev[edge.dest.node\_id] = edge.src.node\_id  i = self.deletemin\_array(tempNodes)  t2 = time.time()  else:  t1 = time.time()   # else use heap  self.source = srcIndex  # set start distance as 0 and prev as None  self.dist[self.source] = 0  self.prev[self.source] = None   self.makequeue() # using dist-values as keys  while len(self.binaryHeap) != 0:  node, distance = self.deletemin\_heap()   for edge in node.neighbors:  alt = self.dist[edge.src.node\_id] + edge.length  if (self.dist[edge.dest.node\_id] == np.inf):  self.insert\_heap( alt, edge.dest)  #self.dist[edge.dest.node\_id] = edge.length  self.prev[edge.dest.node\_id] = node.node\_id  old = self.dist[edge.dest.node\_id]  if (old > alt):  self.dist[edge.dest.node\_id] = alt  self.prev[edge.dest.node\_id] = edge.src.node\_id  self.decreaseKey(edge.dest.node\_id, self.dist[edge.src.node\_id] + edge.length)  t2 = time.time()  return (t2-t1) |