```
 1 #!/usr/bin/python3
 2 import sys
 3
 4 from CS312Graph import *
 5 from module import PQ_Dict, PQ_Heap
 6 import time
 7 import math
 8
 9
10
11 class NetworkRoutingSolver:
12     def __init__( self):
13         pass
14
15     def initializeNetwork( self, network ):
16         assert( type(network) == CS312Graph )
17         self.network = network
18
19     def getShortestPath( self, destIndex ):
20         self.dest = destIndex
21
22         path_edges = []
23         total_length = 0
24         nodeID = destIndex
25
26         # returns no path when there is no path
   from source to destination
27         if self.prev[nodeID] == None:
28             return {'cost': math.inf, 'path': []}
29
30         # tracks back from destination to source
   using the 'prev' map
31         while nodeID != self.source:
32             for edge in self.network.nodes[self.
   prev[nodeID]].neighbors:
33                 if edge.dest.node_id == nodeID:
34                     path_edges.append((edge.src.loc
   , edge.dest.loc, '{:.0f}'.format(edge.length)))
35                     total_length += edge.length
36                     nodeID = self.prev[nodeID]
37                     break
```

```
38
39          return {'cost': total_length, 'path':
    path_edges}
40
41
42      def computeShortestPaths( self, srcIndex,
    use_heap=False ):
43          self.source = srcIndex
44          t1 = time.time()
45
46          self.dijkstras(srcIndex, use_heap)
47
48          t2 = time.time()
49          return (t2-t1)
50
51      def dijkstras(self, startNodeIndex, use_heap):
52          dist = {}
53          prev = {}
54
55          # initializes dist and prev with
    appropriate values for all nodes and starting node
56          for node in self.network.nodes:
57              dist[node.node_id] = sys.maxsize
58              prev[node.node_id] = None
59          dist[startNodeIndex] = 0
60
61          # creates priority queue with array or dict
     based on user specifications
62          if(use_heap):
63              priority_queue = PQ_Heap()
64          else:
65              priority_queue = PQ_Dict()
66          priority_queue.makeQueue(dist.keys(),
    startNodeIndex)
67
68          # iterate through every node
69          while not priority_queue.isEmpty():
70              # delete node with smallest priority
    from priority queues
71              newStartNodeID = priority_queue.
    deleteMin()
```

```
72              # loop through every edge with
    newStart node as the source
73              for neighbor in self.network.nodes[
    newStartNodeID].neighbors:
74                  endNodeIndex = neighbor.dest.
    node_id
75                  edgeLength = neighbor.length
76                  # compare new found path distance
    with current path distance and update if it's
    short
77                  if dist[endNodeIndex] > dist[
    newStartNodeID] + edgeLength:
78                      dist[endNodeIndex] = dist[
    newStartNodeID] + edgeLength
79                      prev[endNodeIndex] =
    newStartNodeID
80                      priority_queue.decreaseKey(
    endNodeIndex, dist[newStartNodeID] + edgeLength)
81
82          self.dist = dist
83          self.prev = prev
84
85
86
87
88
89
90
91
92
```

```
 1 import sys
 2
 3 class PQ_Dict:
 4     def __init__(self):
 5         self.nodeID_to_distance = {}
 6
 7     # Time Complexity: O(1)
 8     # Space Complexity: O(n)
 9     #Adds a new element to the set
10     def insert(self, nodeID, distance):
11         self.nodeID_to_distance[nodeID] = distance
12
13     # Time Complexity: O(n)
14     # Space Complexity: O(n)
15     # Build a priority queue out of the given
   elements, with default start values
16     def makeQueue(self, node_set, startNode_id):
17         for node in node_set:
18             self.nodeID_to_distance[node] = sys.
   maxsize;
19         self.nodeID_to_distance[startNode_id] = 0;
20
21     # Time Complexity: O(n)
22     # Space Complexity: O(1)
23     # Return the element with the smallest key, and
    remove it from the set
24     def deleteMin(self):
25         min_key = min(self.nodeID_to_distance, key=
   self.nodeID_to_distance.get)
26         self.nodeID_to_distance.pop(min_key)
27         return min_key
28
29     # Time Complexity: O(1)
30     # Space Complexity: O(1)
31     # Accommodates the decrease in key value of a
   particular element
32     def decreaseKey(self, key, newDistance):
33         self.nodeID_to_distance[key] = newDistance
34
35     def isEmpty(self):
36         return len(self.nodeID_to_distance) == 0
```

```
37
38
39 class PQ_Heap:
40     def __init__(self):
41         self.heap_tree_list = []
42         self.nodeID_to_priority = {}
43         self.nodeID_to_position = {}
44
45     # Time Complexity: O(log(n))
46     # Space Complexity: O(1)
47     # Adds a new element to the set
48     def insert(self, nodeID, distance):
49         # add new node to the end of the heap
50         self.heap_tree_list.append(nodeID)
51         self.nodeID_to_priority[nodeID] = distance
52         self.nodeID_to_position[nodeID] = len(self.
   heap_tree_list) - 1
53
54         # moves node to the right place
55         self.bubble_up(nodeID)
56
57     # Time Complexity: O(n)
58     # Space Complexity: O(n)
59     # Build a priority queue out of the given
   elements, with default start values
60     def makeQueue(self, node_set, startNode_ID):
61         self.heap_tree_list.append(startNode_ID)
62         self.nodeID_to_priority[startNode_ID] = 0
63         self.nodeID_to_position[startNode_ID] = 0
64         for node in node_set:
65             if not node==startNode_ID:
66                 self.heap_tree_list.append(node)
67                 self.nodeID_to_priority[node] = sys
   .maxsize
68                 self.nodeID_to_position[node] = len
   (self.heap_tree_list) - 1
69
70     # Time Complexity: O(log(n))
71     # Space Complexity: O(1)
72     # Return the element with the smallest key, and
    remove it from the set
```

```
 73     def deleteMin(self):
 74         minID = self.heap_tree_list[0]
 75         lastNodeID = self.heap_tree_list[-1]
 76 
 77         #swap places in heap tree
 78         self.heap_tree_list[0] = lastNodeID
 79         self.heap_tree_list.pop()
 80 
 81         #if there is nothing left in queue after
    popping last element, return
 82         if len(self.heap_tree_list) == 0:
 83             return minID
 84 
 85         #update the postions
 86         self.nodeID_to_position[lastNodeID] = 0
 87         del self.nodeID_to_position[minID]
 88 
 89         #bubble down the node that got pushed to
    the top
 90         self.bubble_down(lastNodeID)
 91 
 92         return minID
 93 
 94     # Time Complexity: O(log(n))
 95     # Space Complexity: 0(1)
 96     # Accommodates the decrease in key value of a
    particular element
 97     def decreaseKey(self, key, newDistance):
 98         self.nodeID_to_priority[key] = newDistance
 99 
100         # moves node to the right place
101         self.bubble_up(key)
102 
103     def isEmpty(self):
104         return len(self.heap_tree_list) == 0
105 
106     # swaps a node and it's parent if the parent's
     priority is smaller than the node's priority
107     def bubble_up(self, nodeID):
108         parentID = self.find_parent(nodeID)
109         while parentID is not None and self.
```

```
109 nodeID_to_priority[parentID] > self.
    nodeID_to_priority[nodeID]:
110             self.swap(nodeID, parentID)
111             parentID = self.find_parent(nodeID)
112 
113     # swaps a node and it's child if the child's
    priority is smaller than the parent's priority
114     def bubble_down(self, nodeID):
115         childID = self.find_lowest_priority_child(
    nodeID)
116         while childID is not None and self.
    nodeID_to_priority[childID] < self.
    nodeID_to_priority[nodeID]:
117             self.swap(nodeID, childID)
118             childID = self.
    find_lowest_priority_child(nodeID)
119 
120     # returns the id of a node's parent
121     def find_parent(self, nodeID):
122         if self.nodeID_to_position[nodeID] == 0:
123             return None
124         parentIndex = (self.nodeID_to_position[
    nodeID] - 1) // 2
125         return self.heap_tree_list[parentIndex]
126 
127     # returns the id of a node's child with the
    lowest priority
128     def find_lowest_priority_child(self, nodeID):
129         right_child_index = (self.
    nodeID_to_position[nodeID] + 1) * 2
130         left_child_index = (self.
    nodeID_to_position[nodeID] + 1) * 2 - 1
131 
132         #no children
133         if left_child_index > len(self.
    heap_tree_list) - 1:
134             return None
135         #2 children
136         elif right_child_index <= len(self.
    heap_tree_list) - 1:
137             right_nodeID = self.heap_tree_list[
```

```
137 right_child_index]
138             left_nodeID = self.heap_tree_list[
    left_child_index]
139 
140             if self.nodeID_to_priority[left_nodeID
    ] < self.nodeID_to_priority[right_nodeID]:
141                 return left_nodeID
142             else:
143                 return right_nodeID
144         #1 child
145         else:
146             return self.heap_tree_list[
    left_child_index]
147 
148     # swaps two nodes in the heap tree
149     def swap(self, node1_id, node2_id):
150         node1_position = self.nodeID_to_position[
    node1_id]
151         node2_position = self.nodeID_to_position[
    node2_id]
152 
153         self.nodeID_to_position[node1_id] =
    node2_position
154         self.nodeID_to_position[node2_id] =
    node1_position
155 
156         self.heap_tree_list[node1_position] =
    node2_id
157         self.heap_tree_list[node2_position] =
    node1_id
158 
159 
160 
161 
162 
```