Atkinson, Jenna – Network Routing

GitHub Link: https://github.com/jennanatkinson/cs312-proj3-network-routing

1. **Correctly implement Dijkstra's algorithm and the functionality discussed above. Include a copy of your (well-documented) code in your submission to the TA.**

DataStructures.py

```
from math import floor
from CS312Graph import *

# Space: O(|V|)
class PQueueArray:
  #Note: uses None as inf distance

  #idIncrement aka num to add to the ids (0 => the real index, 1=> offset by 1
like the GUI)
  #dict pathDict {CS312GraphNode node : [int distance, CS312GraphNode prevNode]}
  #set CS312GraphNode visitedNodes

  # Time: O(|V|)
  def __init__(self, list=None, sourceId:int=None):
    self._idIncrement = 1
    self.pathDict = dict() # Space: O(|V|)
    self.visitedNodes = set() # Space: O(|V|)

    if list is not None:
      self.make_queue(list, sourceId)

  # Time: O(|V|)
  # Set up dictionary of nodes, with max len distances
  def make_queue(self, list, sourceId:int=None):
    for i in range(len(list)):
      if isinstance(list[i], CS312GraphNode):
        # Put the keys in the dictionary, with null distance and prevNode
        if (list[i].node_id == sourceId):
          self.pathDict[list[i]] = [0, None]
        else:
          self.pathDict[list[i]] = [None, None]

  # Add node to set
  # Time: O(1)
  def insert(self, node:CS312GraphNode, dist:int, prevNode:CS312GraphNode):
```

```
    self.decrease_key(node, dist, prevNode)

  # Replace new shortest distance and prevNode for a specific node
  # Time: O(1)
  def decrease_key(self, node:CS312GraphNode, dist:int, prevNode:CS312GraphNode):
    self.set_dist(node, dist)
    self.set_dist_prev_node(node, prevNode)

  #Visit the next unvisited smallest node, return the node and the dist
  # Time: O(|V|)
  def delete_min(self):
    minNode, minDist = None, None
    # Iterate through dict to find min distance
    for key, value in self.pathDict.items():
      if (key not in self.visitedNodes):
        if (value[0] is not None) and (minDist is None or value[0] < minDist):
          minNode = key
          minDist = value[0]
    if minNode is not None:
      self.visitedNodes.add(minNode)
    return minNode, minDist

  # Time: O(1)
  def get_num_visited(self):
    return len(self.visitedNodes)

  # Note: (below) Same helpers as PQueueHeap

  # Time: O(|V|)
  def get_node_by_id(self, nodeId:int):
    for key in self.pathDict:
      if (key.node_id == nodeId):
        return key
    return None

  # Time: O(1)
  def get_dist(self, node:CS312GraphNode):
    return self.pathDict.get(node)[0]

  # Time: O(|V|)
  def get_dist_by_id(self, id:int):
    return self.get_dist(self.get_node_by_id(id))

  # Time: O(1)
  def set_dist(self, node:CS312GraphNode, dist:int):
```

```
    self.pathDict.get(node)[0] = dist

  # Time: O(1)
  def get_dist_prev_node(self, node:CS312GraphNode):
    return self.pathDict.get(node)[1]

  # Time: O(|V|)
  def get_dist_prev_node_by_id(self, id:int):
    return self.get_dist_prev_node(self.get_node_by_id(id))

  # Time: O(1)
  def set_dist_prev_node(self, node:CS312GraphNode, prevNode:CS312GraphNode):
    self.pathDict.get(node)[1] = prevNode

  # Time: O(|V|)
  def __str__(self):
    string = "\nVisited Nodes: "
    if len(self.visitedNodes) != 0:
      for node in self.visitedNodes:
        string += f"{node.node_id+self._idIncrement} "
      string += '\n'
    else:
      string += "*empty*\n"

    table_data = [["NodeKey","[Dist, PrevNode]"]]
    for key, value in self.pathDict.items():
      table_data.append([])
      if isinstance(key, CS312GraphNode):
        table_data[-1].append(key.node_id+self._idIncrement)
      dist, prevNode = value[0], value[1]
      if prevNode is not None and isinstance(prevNode, CS312GraphNode):
        prevNode = prevNode.node_id+self._idIncrement
      if dist is not None:
        table_data[-1].append(f"[{value[0]:.2f}, {prevNode}]")
      else:
        table_data[-1].append(f"[{value[0]}, {prevNode}]")
    for row in table_data:
      string += "{: <7} {: <20}".format(*row) + '\n'

    return string


# Space: O(|V|)
class PQueueHeap:
  #Note: uses inf for distance
```

```
#pathDict => CS312GraphNodes : [dist:int, prev:CS312GraphNodes]
#array of CS312GraphNodes, sorted by the min dist

# Time: O(|V|log|V|)
def __init__(self, list=None, sourceId:int=None):
  self._idIncrement = 1
  self.pathDict = dict() # Space: O(|V|)
  self.nodeQueue = [] # Space: O(|V|)

  if list is not None:
    self.make_queue(list, sourceId)

# Set up dictionary of nodes, with max len distances
# Time: O(|V|log|V|)
def make_queue(self, list, sourceId:int=None):
  for i in range(len(list)):
    if isinstance(list[i], CS312GraphNode):
      # Put the keys in the dictionary, with distance and null prevNode
      dist = float('inf')
      if (list[i].node_id == sourceId):
        dist = 0
      self.insert(list[i], dist, None)

# Time: O(log|V|)
def insert(self, node:CS312GraphNode, dist=float('inf'), prevNode=None):
  # Add node to dict
  if (node != None):
    self.pathDict[node] = [dist, prevNode]
  # Add nodeId as the last element
  self.nodeQueue.append(node)
  # Reorder queue based on that new element
  self._reverseHeapify(len(self.nodeQueue)-1)

# Replace new shortest distance and prevNode for a specific node
# Time: O(log|V|)
def decrease_key(self, node:CS312GraphNode, dist:int, prevNode:CS312GraphNode):
  assert(dist <= self.get_dist(node))
  self.set_dist(node, dist)
  self.set_dist_prev_node(node, prevNode)
  # Reorder queue based on the updated distance
  self._reverseHeapify(self.nodeQueue.index(node))

# Finds the shortest dist node, delete it and return that node and the dist
# Time: O(log|V|)
def delete_min(self):
```

```
    if len(self.nodeQueue) == 0:
      return None, None
    oldRoot = self.nodeQueue[0]

    # Reorganize remaining queue elements
    # Replace root with last element
    self.nodeQueue[0] = self.nodeQueue[-1]
    # Delete last element
    del self.nodeQueue[-1]
    # Heapify new root
    self._heapify(0)

    return oldRoot, self.get_dist(oldRoot)

  # From the given index, check the parent for swaps and loop up the tree
  # Time: O(log|V|)
  def _reverseHeapify(self, startIndex:int):
    i = startIndex
    while i >= 0:
      self._heapify(i) # this call will not recurse, only potential swap parent
with child
      i = floor((i-1)/2) # move up to the parent

  # From the given index, check the left and right children for swaps, recurses
down the tree
  # Time: O(log|V|)
  def _heapify(self, initialIndex:int):
    smallestIndex = initialIndex
    leftIndex = 2*initialIndex + 1
    rightIndex = 2*initialIndex + 2

    # Check if left node dist is smaller
    if (leftIndex < len(self.nodeQueue)
      and self.get_dist(self.nodeQueue[leftIndex]) <
self.get_dist(self.nodeQueue[smallestIndex])):
      smallestIndex = leftIndex

    # Check if right node dist is smaller
    if (rightIndex < len(self.nodeQueue)
      and self.get_dist(self.nodeQueue[rightIndex]) <
self.get_dist(self.nodeQueue[smallestIndex])):
      smallestIndex = rightIndex

    # If smallest is not root, swap and look at sub-tree
    if (smallestIndex != initialIndex):
```

```
      self.nodeQueue[smallestIndex], self.nodeQueue[initialIndex] =
self.nodeQueue[initialIndex], self.nodeQueue[smallestIndex]
      self._heapify(smallestIndex)

  # Time: O(1)
  def get_num_visited(self):
    # totalNodes - nodes left to visit
    return len(self.pathDict) - len(self.nodeQueue)

  # Note: (below) Same helpers as PQueueArray

  # Time: O(|V|)
  def get_node_by_id(self, nodeId:int):
      for key in self.pathDict:
        if (key.node_id == nodeId):
          return key
      return None

  # Time: O(1)
  def get_dist(self, node:CS312GraphNode):
    return self.pathDict.get(node)[0]

  # Time: O(|V|)
  def get_dist_by_id(self, id:int):
    return self.get_dist(self.get_node_by_id(id))

  # Time: O(1)
  def set_dist(self, node:CS312GraphNode, dist:int):
    self.pathDict.get(node)[0] = dist

  # Time: O(1)
  def get_dist_prev_node(self, node:CS312GraphNode):
    return self.pathDict.get(node)[1]

  # Time: O(|V|)
  def get_dist_prev_node_by_id(self, id:int):
    return self.get_dist_prev_node(self.get_node_by_id(id))

  # Time: O(1)
  def set_dist_prev_node(self, node:CS312GraphNode, prevNode:CS312GraphNode):
    self.pathDict.get(node)[1] = prevNode

  # Time: O(|V|)
  def __str__(self):
    string = "\nNodeId Queue: "
```

```
    if len(self.nodeQueue) != 0:
      for node in self.nodeQueue:
        string += f"{node.node_id+self._idIncrement} "
      string += '\n'
    else:
      string += "*empty*\n"

    table_data = [["NodeKey","[Dist, PrevNode]"]]
    for key, value in self.pathDict.items():
      table_data.append([])
      if isinstance(key, CS312GraphNode):
        table_data[-1].append(key.node_id+self._idIncrement)
      dist, prevNode = value[0], value[1]
      if prevNode is not None and isinstance(prevNode, CS312GraphNode):
        prevNode = prevNode.node_id+self._idIncrement
      if dist is not None:
        table_data[-1].append(f"[{value[0]:.2f}, {prevNode}]")
      else:
        table_data[-1].append(f"[{value[0]}, {prevNode}]")
    for row in table_data:
      string += "{: <7} {: <20}".format(*row) + '\n'

    return string
```

NetworkRoutingSolver.py

```
#!/usr/bin/python3
from CS312Graph import *
from DataStructures import *
import time


class NetworkRoutingSolver:
    def __init__(self):
        pass

    def initializeNetwork(self, network):
        assert(type(network) == CS312Graph)
        self.network = network # Space: O(|V|)
        self.queue = None # Space: O(|V|)

    def getShortestPath(self, destIndex):
        # print(self.queue)
        self.dest = self.queue.get_node_by_id(destIndex)
        path_edges = []
        total_length = 0
        # If there is no possible distance to the destination
        if self.queue.get_dist(self.dest) is None or
self.queue.get_dist(self.dest) == float('inf'):
            total_length = float('inf')
        else:
            total_length = self.queue.get_dist(self.dest)
            # Trace the destination node back to the source, working BACKWARDS,
looking up the edges in the network
            currentNode = self.dest
            len = 0
            # print(f"Shortest Path:
({currentNode.node_id+self.queue._idIncrement})", end = '')
            while currentNode.node_id != self.sourceId:
                prevNode = self.queue.get_dist_prev_node(currentNode)
                edgeLen = self.network.getNodeEdge(prevNode.node_id,
currentNode.node_id).length
                # print(f" <--{edgeLen:.2f}--
({prevNode.node_id+self.queue._idIncrement})", end = '')
                path_edges.append((prevNode.loc, currentNode.loc,
'{:.0f}'.format(edgeLen)))
                currentNode = prevNode
                len += edgeLen # to double check this is compounding properly
            # print('\n')
            assert(round(len, 5) == round(total_length, 5))
```

```
        # print(f'Total Cost: {total_length:.2f}')
        # print(f'Path: {path_edges}\n')
        return {'cost':total_length, 'path':path_edges}

    # ArrayTime: O(|V|**2) HeapTime: O(|V|log|V|)
    def computeShortestPaths(self, srcId, use_heap=False):
        self.sourceId = srcId
        t1 = time.time()
        if (use_heap):
            # Time: O(|V|log|V|)
            self.queue = PQueueHeap(self.network.nodes, self.sourceId)
        else:
            # Time: O(|V|)
            self.queue = PQueueArray(self.network.nodes, self.sourceId)
        # print(self.queue)

        # Run while there are unvisited nodes
        # ArrayTime: O(|V|**2) HeapTime: O(|V|log|V|)
        while self.queue.get_num_visited() < len(self.network.nodes):
            # Get the next smallest node/distance that hasn't been visited, add
to visited
            node, dist = self.queue.delete_min() # ArrayTime: O(|V|) HeapTime:
O(log|V|)
            if node is None or dist == float('inf'):
                break # the only nodes that are left are infinity

            # For each edge aka neighbor of the node
            # Time: O(1) bc we have constrained edges to 3
            for i in range(len(node.neighbors)):
                neighborNode = node.neighbors[i].dest
                # Get the shortest distance logged for that edge
                currentEdgeDist = self.queue.get_dist(neighborNode)
                # Calculate what the new distance could be
                newEdgeDist = dist + node.neighbors[i].length

                # If new possible distance is less than current distance, update
                if currentEdgeDist is None or newEdgeDist < currentEdgeDist:
                    #ArrayTime: O(1) HeapTime: O(log|V|)
                    self.queue.decrease_key(neighborNode, newEdgeDist, node)
            # print(self.queue)

        t2 = time.time()
        return (t2-t1)
```

2. **Correctly implement both versions of a priority queue, one using an array with worst case O(1), O(1) and O(|V|) operations and one using a heap with worst case O(log|V|) operations. For each operation (insert, delete-min, and decrease-key) convince us (refer to your included code) that the complexity is what is required here.**

Array

Insert – Time: O(1)

```
# Add node to set
# Time: O(1)
def insert(self, node:CS312GraphNode, dist:int, prevNode:CS312GraphNode):
    self.decrease_key(node, dist, prevNode)
```

Because I used a dictionary to keep track of all my information, I didn't need an insert method, so this just returns decrease-key, aka O(1)

Decrease-key – Time: O(1)

```
# Replace new shortest distance and prevNode for a specific node
# Time: O(1)
def decrease_key(self, node:CS312GraphNode, dist:int, prevNode:CS312GraphNode):
    self.set_dist(node, dist)
    self.set_dist_prev_node(node, prevNode)
```

By using the dictionary to keep all the information, updating the distance and previous node is constant time to set, for a time complexity of O(1).

Delete-min – Time: O(|V|)

```
#Visit the next unvisited smallest node, return the node and the dist
# Time: O(|V|)
def delete_min(self):
    minNode, minDist = None, None
    # Iterate through dict to find min distance
    for key, value in self.pathDict.items():
        if (key not in self.visitedNodes):
            if (value[0] is not None) and (minDist is None or value[0] < minDist):
                minNode = key
                minDist = value[0]
    if minNode is not None:
        self.visitedNodes.add(minNode)
    return minNode, minDist
```

To delete the min, I iterate through every element in the dictionary and then do some constant-time checking to see if that is the smallest distance I can get. Iterating through all the elements gives a time complexity of O(|V|).

Heap

Insert – O(log|V|)

```python
# Time: O(log|V|)
def insert(self, node:CS312GraphNode, dist=float('inf'), prevNode=None):
  # Add node to dict
  if (node != None):
    self.pathDict[node] = [dist, prevNode]
  # Add nodeId as the last element
  self.nodeQueue.append(node)
  # Reorder queue based on that new element
  self._reverseHeapify(len(self.nodeQueue)-1)
```

Insert adds a new node to the dictionary (O(1)) and then appends an element (O(1)) and then reverseHeapify (aka starts at the node and works its way up). By starting at the bottom of the tree, you will only ever deal with ½ of the whole tree. By working your way up, you cut the options in half every time, which gives an O(log|V|) relationship. This gives a total time complexity of O(log|V|).

Decrease-key – O(log|V|)

```python
# Replace new shortest distance and prevNode for a specific node
# Time: O(log|V|)
def decrease_key(self, node:CS312GraphNode, dist:int, prevNode:CS312GraphNode):
  assert(dist <= self.get_dist(node))
  self.set_dist(node, dist)
  self.set_dist_prev_node(node, prevNode)
  # Reorder queue based on the updated distance
  self._reverseHeapify(self.nodeQueue.index(node))
```

Decrease key has similar logic to insert because of the reverseHeapify which is O(log|V|). The other operations deal with setting values in the dictionary, which accessing and modifying it is O(1).

Delete-min – O(log|V|)

```python
# Finds the shortest dist node, delete it and return that node and the dist
# Time: O(log|V|)
def delete_min(self):
  if len(self.nodeQueue) == 0:
    return None, None
  oldRoot = self.nodeQueue[0]

  # Reorganize remaining queue elements
  # Replace root with last element
  self.nodeQueue[0] = self.nodeQueue[-1]
  # Delete last element
  del self.nodeQueue[-1]
  # Heapify new root
  self._heapify(0)

  return oldRoot, self.get_dist(oldRoot)
```

Delete min replaces elements in the queue for O(1) time. Deleting the last element in the array is also O(1) because there is no shifting. Finally, heapify is O(log|V|) because you start at the root and only examine two children. Worse-case, you visit only one child and then recurse. Because we are recursing on half the options every time, we have O(log|V|) time.

3. **Explain the time and space complexity of both implementations of the algorithm by showing and summing up the complexity of each subsection of your code.**

Time Complexity for PQueueArray Misc

Make_queue

```
# Time: O(|V|)
# Set up dictionary of nodes, with max len distances
def make_queue(self, list, sourceId:int=None):
    for i in range(len(list)):
        if isinstance(list[i], CS312GraphNode):
            # Put the keys in the dictionary, with null distance and prevNode
            if (list[i].node_id == sourceId):
                self.pathDict[list[i]] = [0, None]
            else:
                self.pathDict[list[i]] = [None, None]
```

I loop through each member of the list, which is O(|V|),and then add it to my dictionary, O(1), for a total of O(|V|)

Helpers

All the helpers are trivial O(1) accesses/calculations except for the getters that say "**by_id**". Because those are just given an id, I loop through the entire dictionary to find the correct node for O(|V|) time. Example, **get_node_by_id**:

```
# Time: O(|V|)
def get_node_by_id(self, nodeId:int):
    for key in self.pathDict:
        if (key.node_id == nodeId):
            return key
    return None
```

The string function also loops through all the elements for O(|V|).

Space Complexity for PQueueArray

```
# Space: O(|V|)
class PQueueArray:
    #Note: uses None as inf distance

    #idIncrement aka num to add to the ids (0 => the real index, 1=> offset by 1 like the GUI)
    #dict pathDict {CS312GraphNode node : [int distance, CS312GraphNode prevNode]}
    #set CS312GraphNode visitedNodes

    # Time: O(|V|)
    def __init__(self, list=None, sourceId:int=None):
        self._idIncrement = 1
        self.pathDict = dict() # Space: O(|V|)
        self.visitedNodes = set() # Space: O(|V|)
```

For PQueueArray, I have a dictionary and a set of nodes. All of the “node” values are pointers, but even-so, I need to store V of those and the dictionary maps to a len(2) array. That gives me O(|V|) space. The set of visited Nodes could contain pointers to all of the nodes, so it is at worse O(|V|)

Time Complexity for PQueueHeap Misc

Make_queue

```
# Set up dictionary of nodes, with max len distances
# Time: O(|V|log|V|)
def make_queue(self, list, sourceId:int=None):
  for i in range(len(list)):
    if isinstance(list[i], CS312GraphNode):
      # Put the keys in the dictionary, with distance and null prevNode
      dist = float('inf')
      if (list[i].node_id == sourceId):
        dist = 0
      self.insert(list[i], dist, None)
```

Make_queue for PQueueHeap is similar to Array – it loops through all of the elements, O(|V|), but in this, it calls insert for each element, which takes O(log|V|) time to do. The total time complexity is then O(|V|log|V|).

_heapify

```
191 # From the given index, check the left and right children for swaps, recurses down the tree
192 # Time: O(log|V|)
193 def _heapify(self, initialIndex:int):
194   smallestIndex = initialIndex
195   leftIndex = 2*initialIndex + 1
196   rightIndex = 2*initialIndex + 2
197
198   # Check if left node dist is smaller
199   if (leftIndex < len(self.nodeQueue)
200     and self.get_dist(self.nodeQueue[leftIndex]) < self.get_dist(self.nodeQueue[smallestIndex])):
201     smallestIndex = leftIndex
202
203   # Check if right node dist is smaller
204   if (rightIndex < len(self.nodeQueue)
205     and self.get_dist(self.nodeQueue[rightIndex]) < self.get_dist(self.nodeQueue[smallestIndex])):
206     smallestIndex = rightIndex
207
208   # If smallest is not root, swap and look at sub-tree
209   if (smallestIndex != initialIndex):
210     self.nodeQueue[smallestIndex], self.nodeQueue[initialIndex] = self.nodeQueue[initialIndex], self.nodeQueue[
211     self._heapify(smallestIndex)
```

_heapify all has trivial calculations and comparisons, O(1), until line 211 where it is possible it recurses. As explained above, we only recurse one side of the tree at most (and it is very possible we don’t even need to do that), so because we are splitting in half every time, the time complexity is O(log|V|).

_reverseHeapify

```
# From the given index, check the parent for swaps and loop up the tree
# Time: O(log|V|)
def _reverseHeapify(self, startIndex:int):
    i = startIndex
    while i >= 0:
        self._heapify(i) # this call will not recurse, only potential swap parent with child
        i = floor((i-1)/2) # move up to the parent
```

_reverseHeapify calls _heapify, however that is just to avoid duplicate logic. _heapify will only run the trivial calculations, not the recursion because we already know the children are bigger than the parent. With the while loop, we will continue to move up the tree until we get to the root, which is the opposite of the recursion discussed above but has the same logic, so a max time complexity of O(log|V|).

Helpers

The time complexity is the same as the PQueueArray helpers because they are the same.

Space Complexity for PQueueHeap

```
# Space: O(|V|)
class PQueueHeap:
    #Note: uses inf for distance
    #pathDict => CS312GraphNodes : [dist:int, prev:CS312GraphNodes]
    #array of CS312GraphNodes, sorted by the min dist

    # Time: O(|V|log|V|)
    def __init__(self, list=None, sourceId:int=None):
        self._idIncrement = 1
        self.pathDict = dict() # Space: O(|V|)
        self.nodeQueue = [] # Space: O(|V|)
```

The space complexity for this is the same as the PQueueArray – the nodes are pointers, but we still are storing |V| of them in the dictionary and the list.

Time/Space Complexity for Everything else

getShortestPath isn't included in the time data, so I'm going to ignore it here – although you do have to iterate through every edge in the final set and lookup things.

```python
    # ArrayTime: O(|V|**2) HeapTime: O(|V|log|V|)
    def computeShortestPaths(self, srcId, use_heap=False):
        self.sourceId = srcId
        t1 = time.time()
        if (use_heap):
            # Time: O(|V|log|V|)
            self.queue = PQueueHeap(self.network.nodes, self.sourceId)
        else:
            # Time: O(|V|)
            self.queue = PQueueArray(self.network.nodes, self.sourceId)
        # print(self.queue)

        # Run while there are unvisited nodes
        # ArrayTime: O(|V|**2) HeapTime: O(|V|log|V|)
        while self.queue.get_num_visited() < len(self.network.nodes):
            # Get the next smallest node/distance that hasn't been visited, add to visited
            node, dist = self.queue.delete_min() # ArrayTime: O(|V|) HeapTime: O(log|V|)
            if node is None or dist == float('inf'):
                break # the only nodes that are left are infinity

            # For each edge aka neighbor of the node
            # Time: O(1) bc we have constrained edges to 3
            for i in range(len(node.neighbors)):
                neighborNode = node.neighbors[i].dest
                # Get the shortest distance logged for that edge
                currentEdgeDist = self.queue.get_dist(neighborNode)
                # Calculate what the new distance could be
                newEdgeDist = dist + node.neighbors[i].length

                # If new possible distance is less than current distance, update
                if currentEdgeDist is None or newEdgeDist < currentEdgeDist:
                    #ArrayTime: O(1) HeapTime: O(log|V|)
                    self.queue.decrease_key(neighborNode, newEdgeDist, node)
            # print(self.queue)

        t2 = time.time()
        return (t2-t1)
```

computeShortestPaths – Array O(|V|**2)

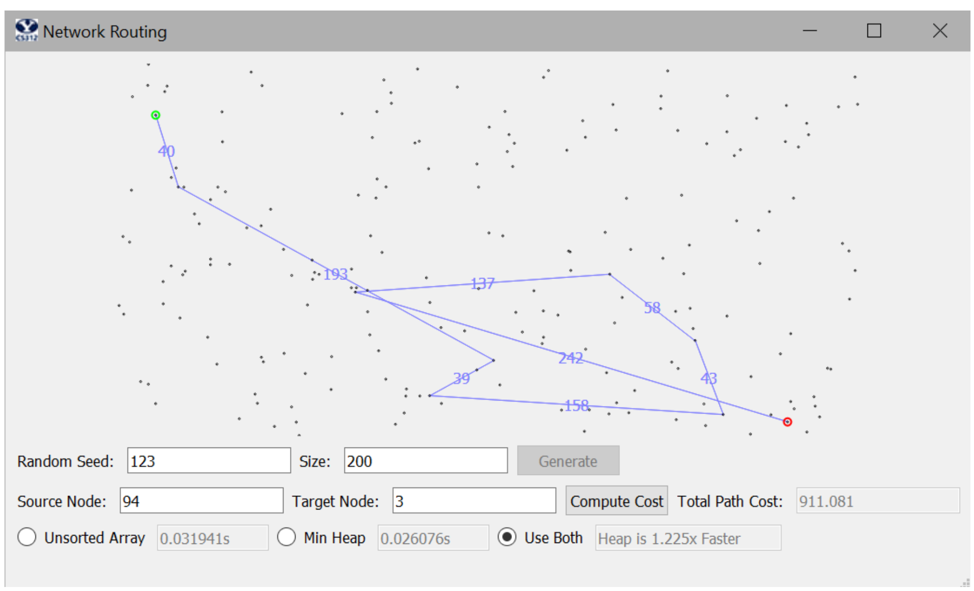
Repeat {{ Delete_min O(|V|) + Decrease_key O(1) }} O(|V|) times for visiting each nodes, equals a **total of O(|V|**2) time**

computeShortestPaths - Heap O(|V|log|V|)

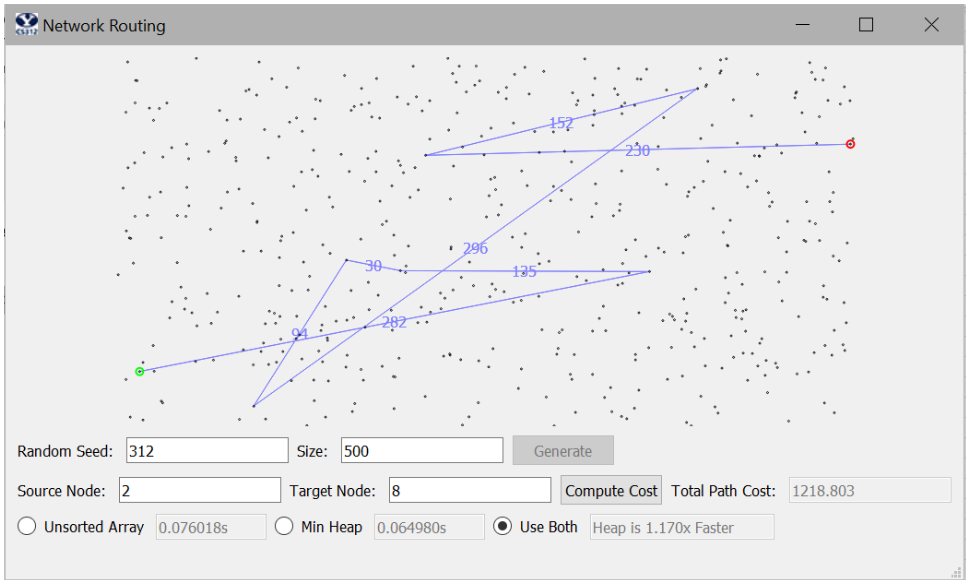
Repeat {{ Delete_min O(log|V|) + Decrease_key O(log|V|) }} O(|V|) times for visiting each nodes, equals a **total of O(|V|log|V|) time**

4. **Submit a screenshot showing the shortest path (if one exists) for each of the three source-destination pairs, as shown in the images below.**
    a. **For Random seed 42 - Size 20, use node 7 (the left-most node) as the source and node 1 (on the bottom toward the right) as the destination, as in the first image below.**

    b. **For Random seed 123 - Size 200, use node 94 (near the upper left) as the source and node 3 (near the lower right) as the destination, as in the second image below.**



    c. **For Random seed 312 - Size 500, use node 2 (near the lower left) as the source and node 8 (near the upper right) as the destination, as in the third image below.**



5. **For different numbers of nodes (100, 1000, 10000, 100000, 1000000), compare the empirical time complexity for Array vs. Heap, and give your best estimate of the difference (for 1000000 nodes, run only the heap version and then estimate how long you might expect your array version to run based on your other results). Discuss the results and give your best explanations of why they turned out as they did.**

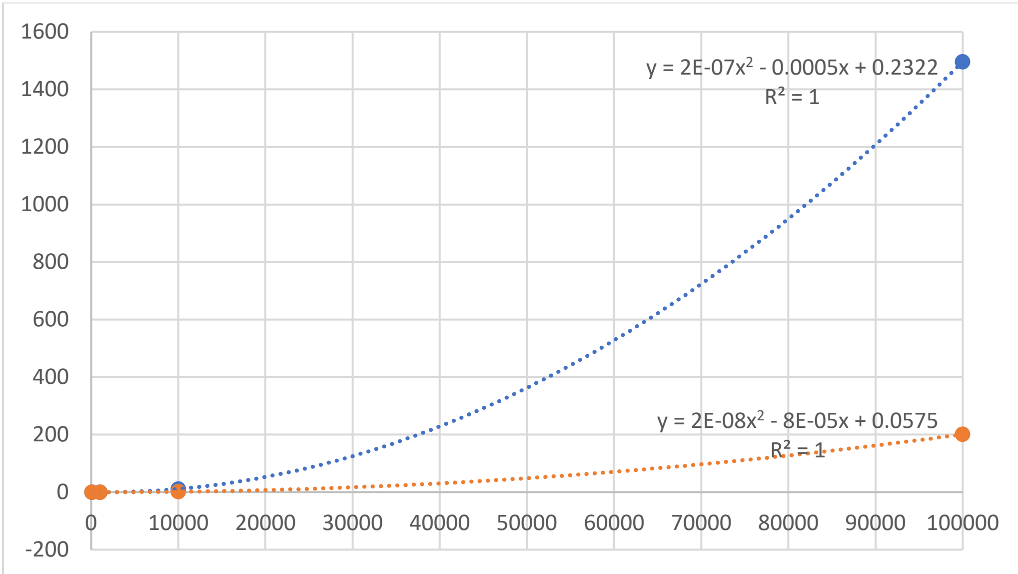
   (#) == seed, the Array and Heap for cell uses the same points

Array (Blue)

N points	Time1 (22)	Time2 (50)	Time3 (2)	Time4 (90)	Time5 (13)	Mean
100	0.004047s	0.003985s	0.004048s	0.003009s	0.003995s	0.0038168s
1000	0.123991s	0.153963s	0.114025s	0.113988s	0.112032s	0.1235998s
10000	10.839992s	12.063032s	11.836996s	9.775024s	10.367998s	10.9766084s
100000	1496.453034s	-	-	-	-	

Heap (Red)

N points	Time1 (22)	Time2 (50)	Time3 (2)	Time4 (90)	Time5 (13)	Mean
100	0.007999s	0.005999s	0.007997s	0.006031s	0.008004s	0.007206s
1000	0.053000s	0.042011s	0.051964s	0.041029s	0.054964s	0.0485936s
10000	1.307003s	1.533997s	1.473999s	1.251010s	1.259000s	1.3650018s
100000	200.769962s	-	-	-	-	



As we expected, the heap performed significantly better than the array on large sets of data. The array performed slightly better than the heap on smaller datasets, simply because it takes a lot more time for the heap to continually sort things. That optimization though is very useful on larger datasets because we don't need to iterate to find the minimum, and shifting things around is pretty fast because of the organization of the heap.

I thought it was surprising how incredibly better the heap was on the largest dataset. Not iterating through all of the items to find the min every loop of the algorithm really helps.