Project 3 - Network Routing

Austin McKamey
CS312
Oct 18, 2022

1. Network Routing Solver

```
from ArrayQueue import ArrayQueue
from CS312Graph import *
import time
import math

from HeapQueue import HeapQueue
from MyQueue import MyQueue


class NetworkRoutingSolver:
   def __init__(self):
       self.source = None
       self.dest = None
       self.network = None
       self.queue = MyQueue
       self.previous = None

   #  Initializes member variable network to the graph made by the GUI
   def initializeNetwork(self, network):
       assert(type(network) == CS312Graph)
       self.network = network

   #  Called after computeShortestPaths, using the previous array computed there to
   #  return shortest path and cost
   def getShortestPath(self, destIndex):
       self.dest = destIndex
       edges = []
       total_length = 0
       #  Iterates backwards from destination node until it reaches the source node
       while self.source != destIndex:
           prevIndex = self.previous[destIndex]
           if prevIndex is None:
               #  Shortest path is not found, meaning the destination node is
unreachable
               return {'cost': float('inf'), 'path': edges}
           prev_node = self.network.nodes[prevIndex]
           edge = None
           for i in range(len(prev_node.neighbors)):
               if prev_node.neighbors[i].dest.node_id == destIndex:
                   edge = prev_node.neighbors[i]
           edges.append((edge.src.loc, edge.dest.loc,
```

```
'{:.0f}'.format(edge.length)))
            total_length += edge.length
            destIndex = prevIndex
        return {'cost': total_length, 'path': edges}

    #  Computes the shortest path from the source node to every node in the graph
using
    #  Dijkstra's algorithm, and creates previous array to trace that path
    def computeShortestPaths(self, srcIndex, use_heap=False):
        self.source = srcIndex
        t1 = time.time()
        dist = []
        prev = []
        #  Dijkstra's
        for i in range(len(self.network.nodes)):
            dist.append(math.inf)
            prev.append(None)
        dist[srcIndex] = 0
        #  Defines which implementation of priority queue to use, based off flags
set
        if not use_heap:
            self.queue = ArrayQueue()
        else:
            self.queue = HeapQueue()
        H = self.queue.makeQueue(dist)
        while len(H.listOfNodes) > 0:
            u = H.deleteMin(dist)
            nodeU = self.network.nodes[u]
            for i in range(len(nodeU.neighbors)):
                v = nodeU.neighbors[i].dest.node_id
                if dist[v] > dist[u] + nodeU.neighbors[i].length:
                    dist[v] = dist[u] + nodeU.neighbors[i].length
                    prev[v] = u
                    H.decreaseKey(v, dist)
        self.previous = prev
        t2 = time.time()
        return t2 - t1
```

2. Time complexity of insert, decreaseKey, deleteMin
    a. Array
        i. insert - uses only the Python list.append function, which is O(1)
        ii. decreaseKey - passes, resulting in O(1)
        iii. deleteMin - iterates through all nodes, resulting in O(|V|)
    b. Heap

i. insert - time is worst case of bubbleUp, which can only iterate as many times as there are levels in the tree, giving us O(log|V|)

ii. decreaseKey - also assumes worst case of bubbleUp, resulting in O(log|V|)

iii. deleteMin - time is worst case of siftDown, which also can only iterate as many times as there are levels in tree, which is O(log|V|)

```
import math

from MyQueue import MyQueue


class ArrayQueue(MyQueue):

   def __init__(self):
       self.listOfNodes = []

   def insert(self, node, distances):
       self.listOfNodes.append(node)

   def makeQueue(self, nodes):
       for i in range(len(nodes)):
           self.insert(i, nodes)
       return self

   def deleteMin(self, distances):
       minimum = math.inf
       toReturn = None
       toRemove = None
       for i in range(len(self.listOfNodes)):
           if distances[self.listOfNodes[i]] <= minimum:
               minimum = distances[self.listOfNodes[i]]
               toReturn = self.listOfNodes[i]
               toRemove = i
       del self.listOfNodes[toRemove]
       return toReturn

   def decreaseKey(self, node, distances):
       pass

import math

from MyQueue import MyQueue


class HeapQueue(MyQueue):

   def __init__(self):
```

```
        self.pointerList = []
        self.listOfNodes = []
        self.currentSize = 0

    def bubbleUp(self, i, distances):
        while i != 0:
            f = math.floor((i-1)/2)
            if distances[self.listOfNodes[i]] < distances[self.listOfNodes[f]]:
                temp = self.listOfNodes[f]
                self.listOfNodes[f] = self.listOfNodes[i]
                self.listOfNodes[i] = temp
                self.pointerList[self.listOfNodes[f]] = f
                self.pointerList[self.listOfNodes[i]] = i
            i = f

    def siftDown(self, i, distances):
        while (i*2 + 1) < self.currentSize:
            smallestChild = self.minChild(i, distances)
            if distances[self.listOfNodes[i]] >
distances[self.listOfNodes[smallestChild]]:
                temp = self.listOfNodes[i]
                self.listOfNodes[i] = self.listOfNodes[smallestChild]
                self.listOfNodes[smallestChild] = temp
                self.pointerList[self.listOfNodes[i]] = i
                self.pointerList[self.listOfNodes[smallestChild]] = smallestChild
                i = smallestChild
            else:
                return

    def minChild(self, i, distances):
        first = i*2 + 1
        last = i*2 + 2
        if last > self.currentSize - 1:
            return first
        else:
            if distances[self.listOfNodes[first]] <
distances[self.listOfNodes[last]]:
                return first
            else:
                return last

    def insert(self, node, distances):
        self.listOfNodes.append(node)
        self.pointerList.append(node)
        self.currentSize += 1
        self.bubbleUp(self.currentSize - 1, distances)
```

```python
    def makeQueue(self, nodes):
        for i in range(len(nodes)):
            self.insert(i, nodes)
        return self

    def deleteMin(self, distances):
        toReturn = self.listOfNodes[0]
        self.listOfNodes[0] = self.listOfNodes[self.currentSize - 1]
        self.pointerList[self.listOfNodes[0]] = 0
        self.pointerList[toReturn] = -1
        del self.listOfNodes[self.currentSize - 1]
        self.currentSize -= 1
        self.siftDown(0, distances)
        return toReturn

    def decreaseKey(self, node, distances):
        index = self.pointerList[node]
        self.bubbleUp(index, distances)
```

3. Time and space complexity of both implementations of the algorithm
    a. Time
        i. Dijkstra with Array - O(|V|^2)

```python
for i in range(len(self.network.nodes)):                 #  O(|V|)
    dist.append(math.inf)                                 #  O(1)
    prev.append(None)                                     #  O(1)
dist[srcIndex] = 0                                        #  O(1)
self.queue = ArrayQueue()
H = self.queue.makeQueue(dist)                            #  O(|V|)
while len(H.listOfNodes) > 0:                            #  O(|V|)
    u = H.deleteMin(dist)                                 #  O(|V|)
    nodeU = self.network.nodes[u]
    for i in range(len(nodeU.neighbors)):                #  O(|E|)
        v = nodeU.neighbors[i].dest.node_id               #  O(1)
        if dist[v] > dist[u] + nodeU.neighbors[i].length: #  O(1)
            dist[v] = dist[u] + nodeU.neighbors[i].length #  O(1)
            prev[v] = u                                   #  O(1)
            H.decreaseKey(v, dist)                        #  O(1)
self.previous = prev
```

        ii. Dijkstra with Heap - O(|V|log|V|)

```python
for i in range(len(self.network.nodes)):                 #  O(|V|)
    dist.append(math.inf)                                 #  O(1)
    prev.append(None)                                     #  O(1)
dist[srcIndex] = 0                                        #  O(1)
self.queue = ArrayQueue()
```

```
H = self.queue.makeQueue(dist)                                  #  O(|V|log|V|)
while len(H.listOfNodes) > 0:                                   #  O(|V|)
    u = H.deleteMin(dist)                                       #  O(log|V|)
    nodeU = self.network.nodes[u]
    for i in range(len(nodeU.neighbors)):                       #  O(|E|)
        v = nodeU.neighbors[i].dest.node_id                     #  O(1)
        if dist[v] > dist[u] + nodeU.neighbors[i].length:       #  O(1)
            dist[v] = dist[u] + nodeU.neighbors[i].length       #  O(1)
            prev[v] = u                                         #  O(1)
            H.decreaseKey(v, dist)                              #  O(log|V|)
self.previous = prev
```

b. Space

i. Dijkstra with Array - O(|V|)

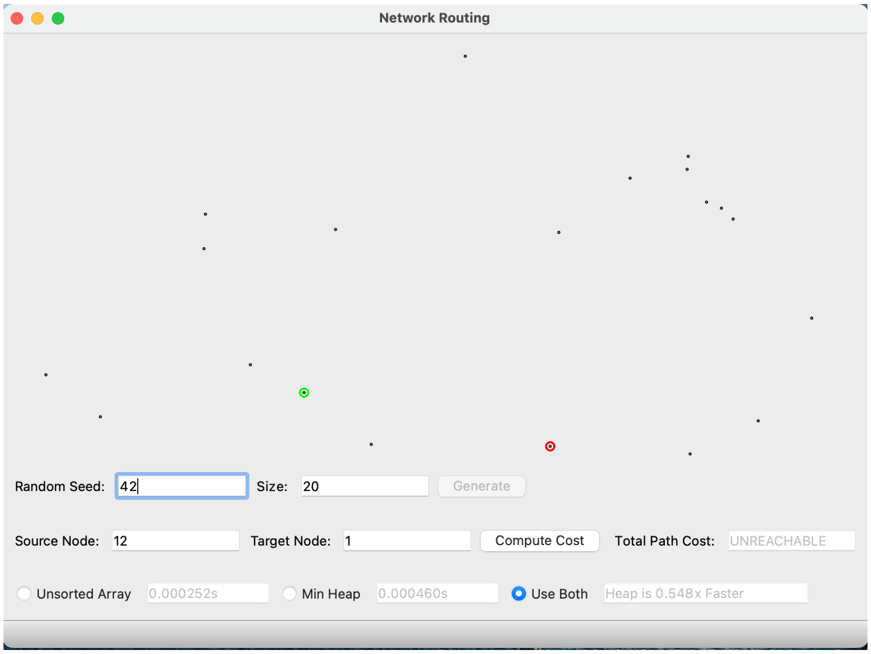
```
for i in range(len(self.network.nodes)):
    dist.append(math.inf)                                       #  O(|V|)
    prev.append(None)                                           #  O(|V|)
dist[srcIndex] = 0
self.queue = ArrayQueue()
H = self.queue.makeQueue(dist)                                  #  O(|V|)
while len(H.listOfNodes) > 0:
    u = H.deleteMin(dist)                                       #  O(1)
    nodeU = self.network.nodes[u]
    for i in range(len(nodeU.neighbors)):                       #  O(1)
        v = nodeU.neighbors[i].dest.node_id
        if dist[v] > dist[u] + nodeU.neighbors[i].length:
            dist[v] = dist[u] + nodeU.neighbors[i].length
            prev[v] = u
            H.decreaseKey(v, dist)
self.previous = prev
```

ii. Dijkstra with Heap - O(|V|)

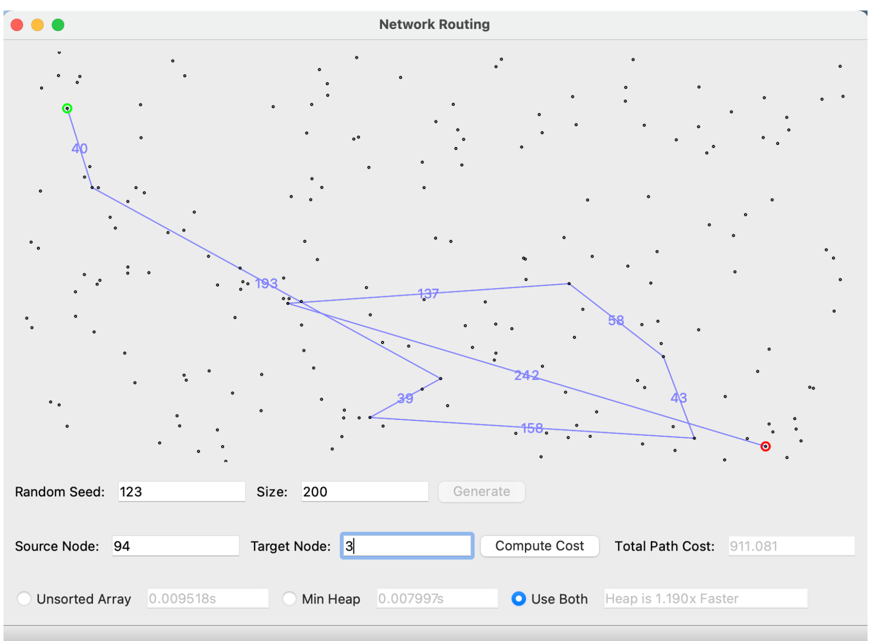
```
for i in range(len(self.network.nodes)):
    dist.append(math.inf)                                       #  O(|V|)
    prev.append(None)                                           #  O(|V|)
dist[srcIndex] = 0
self.queue = ArrayQueue()
H = self.queue.makeQueue(dist)                                  #  O(|V|)
while len(H.listOfNodes) > 0:
    u = H.deleteMin(dist)                                       #  O(1)
    nodeU = self.network.nodes[u]
    for i in range(len(nodeU.neighbors)):                       #  O(1)
        v = nodeU.neighbors[i].dest.node_id
        if dist[v] > dist[u] + nodeU.neighbors[i].length:
            dist[v] = dist[u] + nodeU.neighbors[i].length
            prev[v] = u
            H.decreaseKey(v, dist)
```

```
self.previous = prev
```

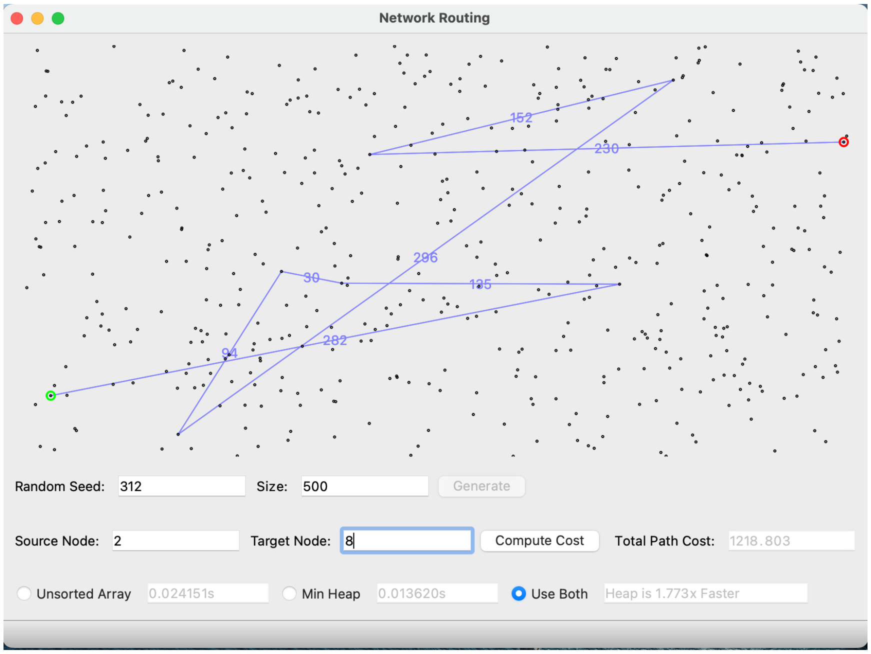
4. Shortest Path
   a. Random seed 42 - Size 2



   b. Random seed 123 - Size 200

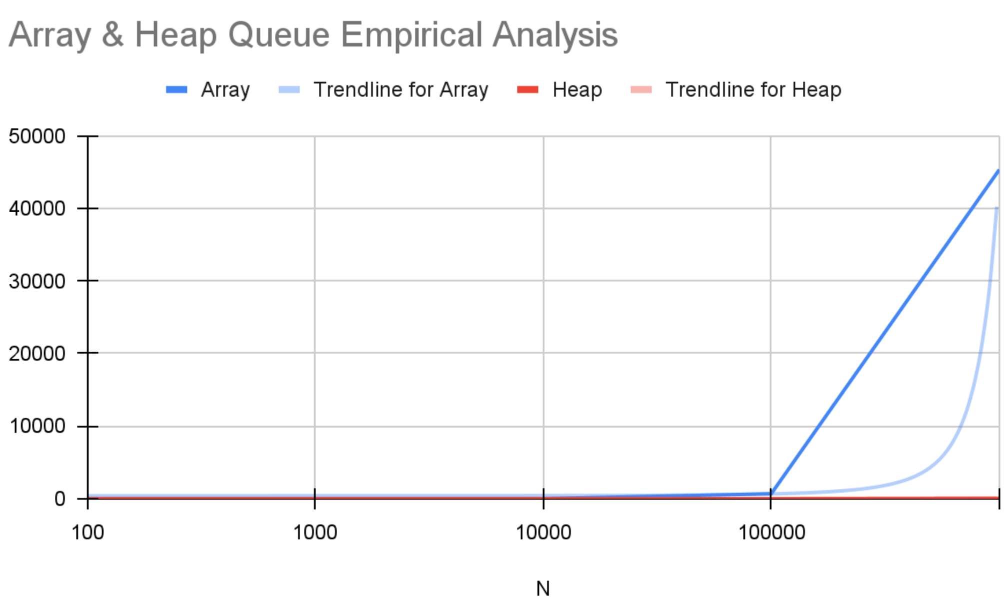


   c. Random seed 312 - Size 500



5. Empirical Analysis

	100	1000	10000	100000	1000000
ArrayQueue	0.0019178	0.1145494	4.5701056	692.4217778	45350.03516
HeapQueue	0.0028968	0.0334182	0.2364614	4.2842304	55.3750078



In order to estimate the array time for 1,000,000 nodes, I used the time complexity found in 3a for Dijkstra's algorithm with the array implementation. This gave me 45,350 minutes, or about 12.5 hours. The results showed that for very small sizes of networks, the array implementation was actually faster than the heap implementation. As the size increased, however, the array implementation time increased drastically, resulting in the heap implementation being 819 times faster for a size of 1,000,000. This makes sense, when we consider the time complexity to insert, deleteMin, and decreaseKey for the array and heap. For smaller sizes, the linear time to deleteMin for the array is trivial. However, when we have to deleteMin 1,000,000 times, it is dramatically slower than the logarithmic time to deleteMind for the heap. It is so dramatic, in fact, that the constant time to insert and decreaseKey in the array makes up almost no ground whatsoever, even when compared to the logarithmic time for the heap implementation.