Matthew R. Christensen

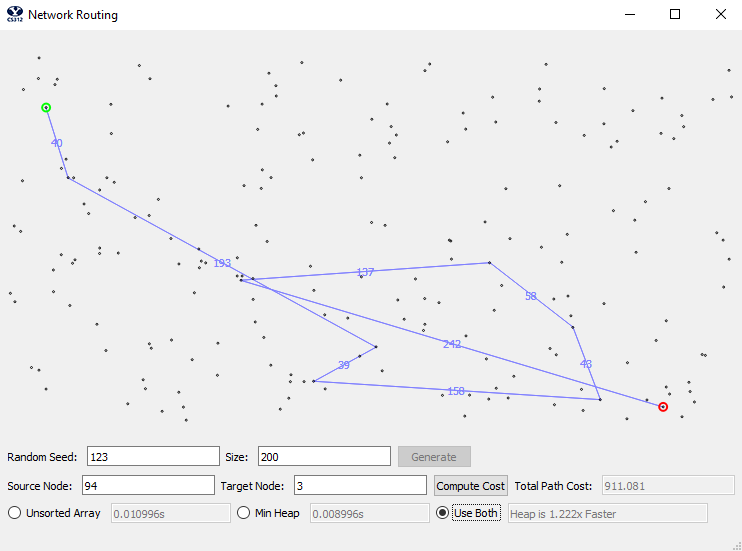
CS 312 (001) – Dr Martinez, Tony R

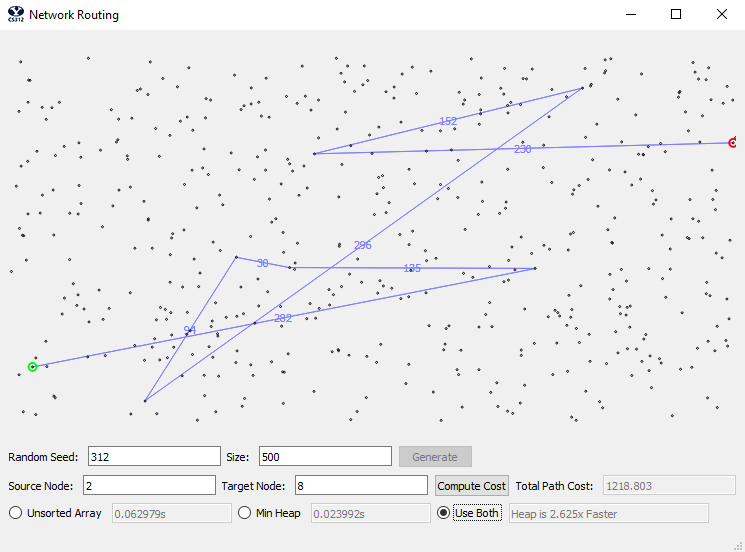
February 25, 2021

Network Routing Algorithm Report

**Screenshots**







**Time and Space Complexity**

We first will talk about the time and space complexity of the program generally followed by a specific review of the two implementations using different data structures: the unsorted array implementation vs the heap implementation of the priority queue.

First, it is important to note that the theoretical time that we want to hit for the slack implementation is O(n2) while the goal of the heap implementation is O((|E| + n) log n). It can be noted that the binary heap implementation should be better than the array implementation when |E| < n2 / log n. Additionally, n here represents the number of vertices in the given graph problem. This O(|V|) is equivalent to O(n) for the purposes of this paper.

The algorithm used for graph exploration, in both implementations, is Dijkstra's Algorithm. The pseudo code is as follows:

*H = makequeue(V)*

*While H is not empty:*

*U = deletemin(H)*

*For all edges (u, v) of in E:*

*If dist(v) > dist(u) + l(u,v):*

*dist(v) = dist(u) + l(u,v)*

*prev(v) = u*

*decreasekey(H,v)*

Creating the queue, the delete minimum function, and the decrease key function all depend on their data structure for their respective time and space complexity. However, we know that the while loop with run at most n times as it must go through the entirety of the queue, and that each point with have at most 3 edges, for a O(n(E)) = O(3n)

However, the other part of this algorithm involves tracing the queue backwards to find and report the shortest path. This part of the algorithm is not dependent on the data structure used in implementation and thusly will be discussed in this section of the paper. The pseudo code is as follows:

*current\_node = final\_node*

*while current\_node is not none:*

*previous\_node = getPrevNode(current\_node)*

*for edge in previous\_node.edges:*

*if edge.dest is current\_node:*

*shortest\_past.append(current\_node)*

*current\_node = previous\_node*

We can see that in the worst case scenario we have the shortest path take every node, and thus have to visit and store n number of nodes, for a time and space complexity of O(n) (because the edges reduce down due to not being significant compared to O(n)).

**Unsorted Array Implementation Analysis**

Now we will analyze the unsorted array implementation of the algorithm. First, the create queue function:

*For node in graph:*

*Array.append(node)*

We will have a time and space complexity of O(n), where n represents the number of vertices (|V|), because of the need to initially iterate and store all of the vertices in the queue.

Next, the delete minimum function:

*For node in queue:*

*If node.dist < min:*

*min = node*

*return queue.pop(min)*

Because the array is unsorted, we need to iterate through the entire array to guarantee that we found the smallest node distance. Thus our time complexity is O(n) while our space complexity is O(1), because of the lack of need to store anything meaningful. We will improve on this complexity in our heap due to storing nodes in an order that we can traverse to increase our time efficiency.

Next, we have the decrease function call. Because we don’t retain any sense of ordering within our array data structure implementation, we do not need to implement such a function (and thus skip this function for the array implementation).

This proves our case that the unsorted array should be of O(n2). Because we know that getting the sortest path is of O(n), and that Dikstras algorithm is of O(3n(n + n)), we know that the totally order of complexity for the stack implementation is O(n + 3n(n + n)) = O(n + 3n(2n)) = O(n + 6n2) = O(n2)

**Heap Priority Queue Implementation Analysis**

Now we will analyze the algorithm theoretically with the heap priority queue implementation.

First, the create queue function:

*For node in graph:*

*Queue.set\_node(node)*

*Percolate\_up(node)*

Because we go through each node, and percolating is a function of O(log n) time complexity, we get a time complexity of O(n log n). While we only have to store a max of n node, for a space complexity of O(n)

Next, the delete minimum function:

//ran out of time, see code below

O(log n)

Next, we have the decrease function call.

//ran out of time, see code below

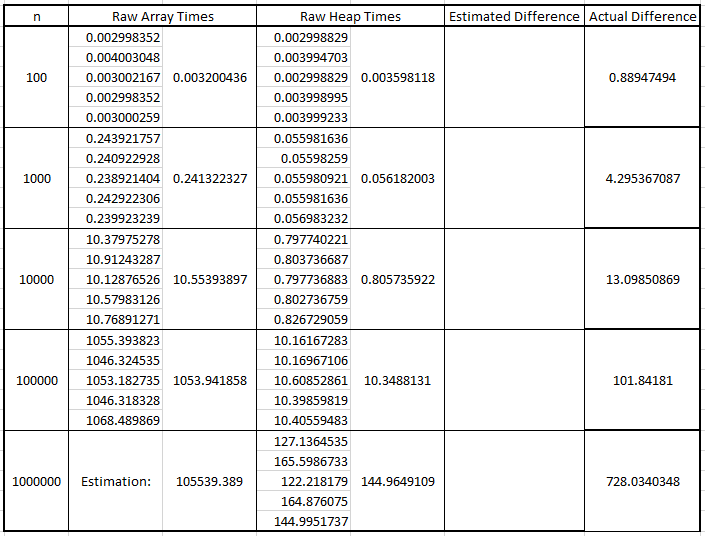
O(log n)

Thus we see that our heap implementation of the algorithm does indeed fit our theoretical complexity of O(n log n), as O(n + 3n(log n + log n + log n) = O(n + 3n(3log n) =

O(n + 6n log n) = O(n log n).

**Empirical Algorithm Result Analysis**

Now that we have defined the complexity of the algorithm, including both implementation of the priority queue (unsorted array and heap), we can compare what we expect to see against real world results. We let n be powers of 10, where *n* = {100, 1000, 10000, 100000, 1000000} indicates the number of points, or vertices, in the given execution. We then run our algorithm, using both the unsorted array and heap implementation of the priority queue, for every value of n to create the results seen below:



Please note that the results for the unsorted array for the n value of 1,000,000 are estimated using a constant k. The constant k is solved for using the following equation:



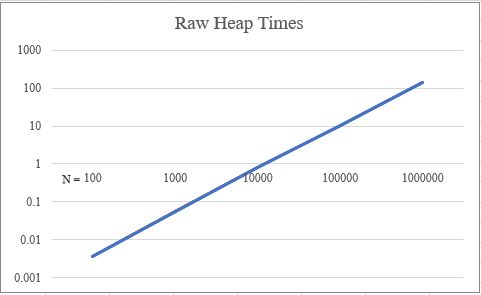
We can simply solve for *k* by finding what our real practical results are divided by the theoretical result of n2. We can them simply solve for k, and by doing so find:

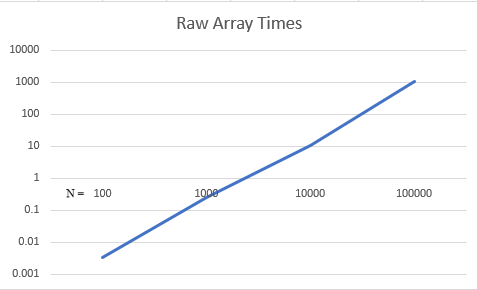


This process gives us a *k* of 1.05539389E-07. By multiplying n2 by our k value for an n value of 1,000,000 to achieve out estimate of 1055.39389 seconds:

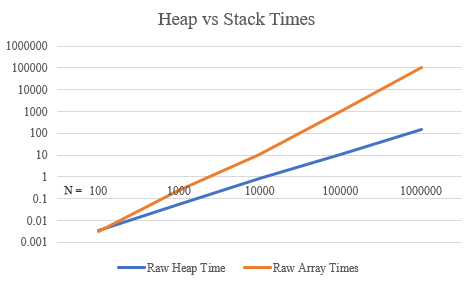


Plotting the times of the array and heap computation times, on a logarithmic scale, we see:





This confirms our theoretical analysis of O(n log n) and O(n2) respectively because of the straight nature of the graphs. Plotting the results of the stack and heap implementations against each we can see the difference between the logarithmic and exponential grown:



**Conclusion**

We can thus conclude that our algorithm is correctly running at an approximate time complexity of n log n and space complexity for the heap implementation while the unsorted array implementation does indeed run at a time complexity of n2. Furthermore, we have defined the constants k’s that each algorithm runs in time with.

**Appendix**

*from CS312Graph import \**

*import time*

*import math*

*class NetworkRoutingSolver:*

*def \_\_init\_\_( self):*

*pass*

*def initializeNetwork( self, network ):*

*assert( type(network) == CS312Graph )*

*self.network = network*

*self.results = {}*

*def getShortestPath( self, destIndex ):*

*print("getShortestPath")*

*self.dest = destIndex*

*path\_edges = []*

*total\_length = 0*

*node = self.network.nodes[self.dest]*

*while self.results[node.node\_id]["prev"] is not None: # Traverse the graph backwards*

*previous\_node = self.network.nodes[self.results[node.node\_id]['prev']]*

*for neighbor in previous\_node.neighbors:*

*if neighbor.dest is node:*

*total\_length = total\_length + neighbor.length*

*path\_edges.append((neighbor.src.loc, neighbor.dest.loc, '{:.0f}'.format(neighbor.length)))*

*node = previous\_node*

*return {'cost': total\_length, 'path':path\_edges}*

*def computeShortestPaths(self, src\_index, use\_heap=False):*

*print("computeShortestPaths")*

*t1 = time.time()*

*if use\_heap:*

*queue = HeapPriorityQueue(self.network, src\_index)*

*else:*

*queue = UnsortedArrayPriorityQueue(self.network, src\_index)*

*for node in self.network.nodes:*

*self.results[node.node\_id] = {'dist': math.inf, 'prev': None}*

*self.results[src\_index]['dist'] = 0*

*print("Started queue")*

*while queue.is\_not\_empty():*

*print("Queue length: ", len(queue))*

*u = queue.delete\_min()*

*edges = self.network.nodes[u['id']].neighbors*

*for edge in edges:*

*v = self.results[edge.dest.node\_id]*

*# v2 = edge.dest*

*if v['dist'] > u['dist'] + edge.length:*

*v['dist'] = u['dist'] + edge.length*

*v['prev'] = u['id']*

*queue.decrease\_key(edge.dest.node\_id)*

*queue.update\_node(edge.dest.node\_id, v["dist"])*

*print("Finished queue")*

*t2 = time.time()*

*print(t2-t1)*

*return t2-t1*

*class UnsortedArrayPriorityQueue:*

*def \_\_init\_\_(self, graph, source\_index):*

*print("Start init for array pq")*

*self.num\_nodes = len(graph.nodes)*

*self.queue = {}*

*for index in range(self.num\_nodes):*

*if index == source\_index:*

*self.queue[graph.nodes[index].node\_id] = {'dist': 0}*

*else:*

*self.queue[graph.nodes[index].node\_id] = {'dist': math.inf}*

*print("Finish init for array pq")*

*def delete\_min(self):*

*print("started delete")*

*smallest\_index = -1*

*smallest\_distance = math.inf*

*for index, node in self.queue.items():*

*if self.queue[index]['dist'] < smallest\_distance:*

*smallest\_distance = self.queue[index]['dist']*

*smallest\_index = index*

*smallest\_node = {'id': smallest\_index, 'dist': smallest\_distance}*

*if smallest\_index is -1:*

*first\_node = self.queue.popitem()*

*print("Finished delete")*

*return {'id': first\_node[0], 'dist': first\_node[1]['dist']}*

*del self.queue[smallest\_index]*

*print("Finished delete")*

*return smallest\_node*

*def update\_node(self, index, distance):*

*self.queue[index]['dist'] = distance*

*def is\_not\_empty(self):*

*if len(self.queue) > 0:*

*return True*

*else:*

*return False*

*def decrease\_key(self, foo):*

*pass*

*class HeapPriorityQueue:*

*def \_\_init\_\_(self, graph, src\_index):*

*self.heap = []*

*for node in graph.nodes:*

*if node.node\_id == src\_index:*

*self.insert\_node(node.node\_id, 0)*

*else:*

*self.insert\_node(node.node\_id, math.inf)*

*def \_\_len\_\_(self):*

*return len(self.heap) - 1*

*def insert\_node(self, node\_id, distance):*

*print("started insert")*

*self.heap.append({'id': node\_id, 'dist': distance})*

*self.percolate\_up(len(self))*

*print("Finished insert")*

*def delete\_min(self):*

*print("Started delete min")*

*return\_node = self.heap[0]*

*self.heap[0] = self.heap[len(self)]*

*self.heap.pop()*

*self.percolate\_down(0)*

*print("Ended delete min")*

*return return\_node*

*def decrease\_key(self, node\_id):*

*self.percolate\_up(node\_id)*

*def percolate\_up(self, index):*

*if index is 0:*

*return*

*parent\_index = index // 2*

*if self.heap[parent\_index]['dist'] > self.heap[index]['dist']:*

*self.swap\_node(index, parent\_index)*

*self.percolate\_up(parent\_index)*

*def percolate\_down(self, parent\_index):*

*print("started percolate\_down")*

*if parent\_index is len(self):*

*return*

*while parent\_index \* 2 <= len(self):*

*mc = self.min\_child(parent\_index)*

*if self.heap[parent\_index]["dist"] > self.heap[mc]["dist"]:*

*self.swap\_node(mc, parent\_index)*

*parent\_index = mc*

*print("finished percolate\_down")*

*def min\_child(self, index):*

*print("started min\_child")*

*if index \* 2 + 1 > len(self):*

*print("finished min\_child")*

*return index \* 2*

*if self.heap[index \* 2]['dist'] < self.heap[index \* 2 + 1]["dist"]:*

*print("finished min\_child")*

*return index \* 2*

*print("finished min\_child")*

*return index \* 2 + 1*

*def update\_node(self, node\_id, distance): # todo check to see if this is being used*

*for node in self.heap:*

*if node['id'] is node\_id:*

*node["dist"] = distance*

*break*

*def swap\_node(self, index\_1, index\_2):*

*node = self.heap[index\_1]*

*self.heap[index\_1] = self.heap[index\_2]*

*self.heap[index\_2] = node*

*def is\_not\_empty(self):*

*if len(self) > 0:*

*return True*

*else:*

*return False*