Thomas Hart

CS 312

Network Routing Project

1) Code:

```
# Backtracks through prev array made by Dijkstra’s to get the shortest path. O(v)
# worst case
def getShortestPath(self, destIndex):
    self.dest = destIndex
    path_edges = []
    total_length = 0
    currentNode = self.network.nodes[destIndex]
    prevNode = self.network.nodes[self.prev[currentNode.node_id]]
    while currentNode.node_id != self.source:
        for i in range(3):
            if prevNode.neighbors[i].dest == currentNode:
                edge = prevNode.neighbors[i]
                path_edges.append((edge.src.loc, edge.dest.loc,
'{:.0f}'.format(edge.length)))
                total_length += edge.length
                currentNode = prevNode
                prevNode = self.network.nodes[self.prev[currentNode.node_id]]
    return {'cost': total_length, 'path': path_edges}

# Runs Dijkstra’s algorithm. O(VlogV) time for Heap and O(V^2) for Array
def computeShortestPaths(self, srcIndex, use_heap=False):
    self.dist = []
    self.prev = []
    self.source = srcIndex
    self.use_heap = use_heap
    t1 = time.time()

    for i in range(len(self.network.nodes)):        # O(V) to make prev + dist
        self.dist.append(math.inf)
        self.prev.append(-1)
        if i == srcIndex:
            self.dist[i] = 0

    H = self.makeQueue()                            # Heap, array: O(V)
    while self.queueSize > 0:
        u = self.network.nodes[self.deleteMin(H)]   # H:O(logV), A:O(V)
        for i in range(3):
            v = u.neighbors[i].dest
            if self.dist[v.node_id] > self.dist[u.node_id] + u.neighbors[i].length:
                self.dist[v.node_id] = self.dist[u.node_id] + u.neighbors[i].length
                self.prev[v.node_id] = u.node_id
                self.decreaseKey(H, v.node_id)      # H:O(logV), A:O(1)

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

# Makes the priority queue, implementing the heap or array depending on # use_heap
def makeQueue(self):
    if not self.use_heap:                 # Array: O(V) set all to -1
        H = []
        self.queueSize = len(self.dist)
        for i in range(len(self.dist)):
            H.append(-1)
            if self.dist[i] == 0:
                H[i] = 0

    else:                                   # Heap: O(V) to make i array
        self.indices = []
        for i in range(len(self.dist)):
            self.indices.append(-1)

        H = []
        H.append([self.source, 0])
        self.indices[self.source] = 0
        self.queueSize = 1

    return H

# Deletes the smallest distance value in priority queue. Returns its node_id/index
def deleteMin(self, H):
    if not self.use_heap:  # Array deleteMin O(V)
        min = math.inf
        index = -1
        for i in range(len(H)):
            if H[i] == -1:
                continue
            if H[i] == 0:
                H[i] = -1
                return i
            if H[i] < min:
                min = H[i]
                index = i
        H[index] = -1

    else:                    # Heap deleteMin O(logV)
        index = H[0][0]
        H[0] = H[self.queueSize - 1]
        H.pop(self.queueSize - 1)
        if len(H) != 0: self.shiftDown(H)   # shiftDown: O(logV)

    self.queueSize -= 1
    return index

# If a closer distance has been found, decrease distance key in priority queue
def decreaseKey(self, H, nodeId):

    if not self.use_heap:   # Array decreaseKey O(1)
        H[nodeId] = self.dist[nodeId]

    else:                   # Heap decreaseKey AND insert O(logV)
        if self.indices[nodeId] == -1:
            self.insert(H, nodeId, self.dist[nodeId])
        else:
            H[self.indices[nodeId]][1] = self.dist[nodeId]
            self.shiftUp(H, self.indices[nodeId])

# Inserts a node that doesn't exist into the priority queue(heap)
def insert(self, H, index, dist): # Heap insert O(logV)
    H.append([index, dist])
    self.queueSize += 1
    self.shiftUp(H, self.queueSize - 1) # shiftUp O(logV)

# These functions find the indices we need for the heap. All O(1)
def getParent(self, childIndex):
    return (childIndex - 1) // 2

def getLeftChild(self, parentIndex):
```

```python
    return (2 * parentIndex) + 1

def getRightChild(self, parentIndex):
    return (2 * parentIndex) + 2

# Function needed for deleteMin. Pop the top node, replace it with the last node in
# the array we are using as the heap, then shift it down until it is sorted
def shiftDown(self, H):         # O(logV) worst case
    ni = 0
    li = self.getLeftChild(ni)
    ri = self.getRightChild(ni)
    node = H[ni]

    if li < len(H): l = H[li]   # left and right index assignments
    else: l = [-1, math.inf]    # if the indices don't exist in the
    if ri < len(H): r = H[ri]   # priority queue, set to inf
    else: r = [-1, math.inf]

    while node[1] > l[1] or node[1] > r[1]:
        if l[1] < r[1]:
            H[ni], H[li] = H[li], H[ni]
            self.indices[node[0]] = li
            self.indices[l[0]] = ni
            ni = li
        else:
            H[ni], H[ri] = H[ri], H[ni]
            self.indices[node[0]] = ri
            self.indices[r[0]] = ni
            ni = ri
        li = self.getLeftChild(ni)
        ri = self.getRightChild(ni)
        node = H[ni]
        if li < len(H): l = H[li]
        else: l = [-1, math.inf]
        if ri < len(H): r = H[ri]
        else: r = [-1, math.inf]

# Function needed for insert or decreaseKey. Takes given index of a node that
# has been adjusted and shifts it up according to its smaller distance value
def shiftUp(self, H, index):      # O(logV) worst case
    ni = index
    pi = self.getParent(ni)
    node = H[ni]
    p = H[pi]
    while node[1] < p[1]:
        H[ni], H[pi] = H[pi], H[ni]
        self.indices[node[0]] = pi
        self.indices[p[0]] = ni
        ni = pi
        pi = self.getParent(ni)
        node = H[ni]
        p = H[pi]
```

2) Code shows both implementations of Dijkstra's algorithm. For the **array implementation**, all spaces for the priority queue are created at **makeQueue** (which is **O(V)**), so the insert function just changes the value at the given index. **decreaseKey and insert** are basically the same here, so I use the decreaseKey function for both, which is **O(1)**. The **deleteMin** function for the array looks through the whole priority queue to find the min, which is **O(V)**.

For the **heap**, **makeQueue** just inserts the source node into the priority queue as a tuple: [node_id, distance value]. This is **O(1)**. **DeleteMin** pops index 0 off the heap and replaces index 0 with the last node in the heap. It shifts this node down until it is sorted correctly. This is **O(logV)**. **DecreaseKey and insert** are also **both O(logV)** and are very similar. Insert places a new node into the queue and shifts it up until it is sorted. DecreaseKey changes the value of an existing node and shifts it up in the tree until it is sorted. IT DOES NOT HAVE TO LOOK THROUGH THE WHOLE TREE TO FIND THE RIGHT NODE. A separate array holds the indices for the nodes in the heap, so we can get the right node and change its distance value in constant time.

3) Space and time complexity

This draws the shortest path edges after the algorithm is run. In the worst case, it could iterate through all the nodes tracing back to the source node from the destination node, so it is O(V) time and O(1) space.

a) getShortestPath: # Whole thing O(V) worst case, O(1) space complexity

    While loop traces through prev array from destIndex to source Index: # O(V)

This is Dijkstra's algorithm. It is O(V) space complexity for both because there are a few arrays that need to be filled with values for each implementation that are V elements long. The array implementation of the priority queue is simpler but much slower than the heap implementation when getting into a large number of nodes. I'll talk about the specifics under the specific functions.

b) computeShortestPath: # Whole thing O(V^2) for array, O(VlogV) for heap

```
Loop to make dist and prev arrays     # O(V) time and space for heap and array
H=makeQueue                           # O(V) time and space for heap and array
While loop, loops V times:
        U=deleteMin                   # O(|V| x |V|) array, O(|V| x |logV|) heap
        Compare edges                 # Constant O(1) for array and heap
        If smaller distance found:
                Change dist values    # Constant O(1) for array and heap
                DecreaseKey           # O(|V| x |1|) array, O(|V| x |logV|) heap
```

Note: insert and decreaseKey do the same thing for array, so they are synonymous. For the heap, insert is embedded into my decreaseKey function. If the node does not exist in the array, it inserts it. If it does exist, it changes the value. In both cases, the node in question is shifted up through the heap until it is sorted appropriately.

This makes the array or heap priority queue depending on the self.use_heap Boolean. The array priority queue is just an array V elements long with -1 values to represent that the corresponding nodes are not in the queue yet or have been removed. The source node is set to a distance of zero to start with. For the heap implementation, I had the queue starting at 1 element long, and I added them as I went through the algorithm. However, this still takes O(V) space and time to begin with because the indices array needs to be created to keep track of which node indices in the distance array correspond to the indices in the heap. makeQueue is then O(V) time and space for both implementations.

```
c)  makeQueue                    # Whole thing O(V) time and space for heap and array

        if not self.use_heap:

            for loop sets V elements to -1 in H          # O(V) time and space

            set source index to 0 in H                   # O(1)


        else:

            for loops creates indices array              # O(V) time and space

            append source node to heap H                 # O(1)

            self.queueSize += 1                          # O(1)

        return H
```

This function returns the index of the node with the minimum distance value in the priority queue and deletes it from the queue. The array implementation requires us to iterate through the whole array, so that is O(V) time and O(1) space. The heap pops the top (minimum) element in constant time. The last element in the heap takes its place and shifts down through the heap into its correct spot in O(logV) time and O(1) space. Heap is made much faster than the array here.

```
d)  deleteMin                    # O(V) array time, O(logV) heap time, both O(1) space

        if not self.use_heap:

            iterate through priority queue to find min   # O(V)

            set it to -1 (erased) and return node_id


        else:

            pop first element, which is min              # O(1)

            set last element to first and shift it down  # O(logV)

            return node_id of that element
```

This changes the value in the priority queue to match the shorter distance found and placed into the distance array. It is constant O(1) time for the array implementation and O(1) space. For the heap, it changes the value but also must shift the node up to keep it sorted, which is O(logV) time. As discussed above, the indices array makes it so we don't have to iterate through the whole heap to change the value. We can find it using the array in constant time. This function also calls the insert function in the heap implementation if the node in question doesn't exist in the priority queue yet. This is also O(logV) time and O(1) space complexity.

```
e) decreaseKey                    # O(1) array time, O(logV) heap time, both O(1) space
        if not self.use_heap
            priority queue at index = dist array at index   # O(1)
        else:
            if not in priority queue: insert                # O(logV)
            else: decrease value and shiftUp at given index # O(logV)
```

This inserts a node into the heap if it doesn't already exist. It places it at the end of the array representing the heap and then shifts up until it is properly sorted in O(logV) time, O(1) space.

```
f) insert                 # O(logV) time, O(1) space, only used in heap implementation
        append node to heap              # O(1)
        self.queueSize += 1              # O(1)
        shiftUp heap at given index      # O(logV)
```

These are required in order to find the correct heap indices for our heap implementation of the priority queue.

g) The functions getLeftChild, getRightChild, and getParent are all O(1) time and space

This takes the top node that was placed after deleteMin and shifts it down through the heap until it is sorted properly. If one or both of the child nodes has a distance value smaller than the node, it swaps with the child that has the smallest distance value until the node either has no children or it's value is smaller than both of it's children. O(logV) time worst case. O(1) space.

```
h) shiftDown                      O(logV) time and O(1) space complexity for heap
        set current index and its children indices    # O(1)
        while node's left or right child dist value < node dist value:   # O(logV) loops
            swap node with child that has the smallest dist value   # O(1) work
```

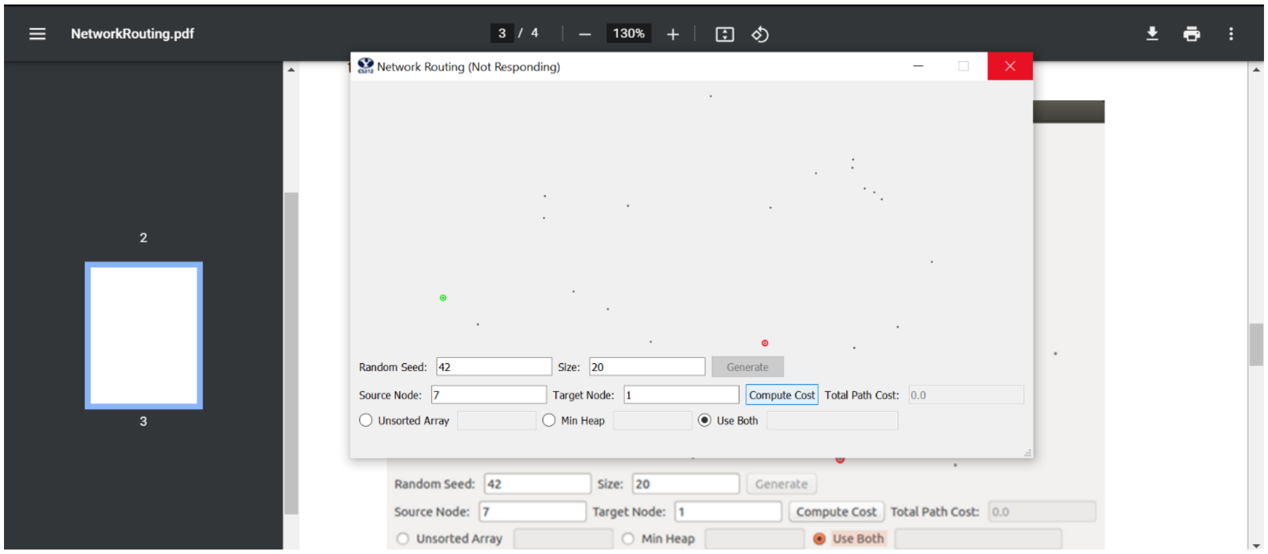
An index is given so the function knows which index in the heap to set the current node to. Swaps the current node with its parent while its distance value is smaller than its parent. Whole thing is also O(logV) time worst case and O(1) space complexity.

i) shiftUp

```
set current index and its parent index            # O(1)
while node's parent dist value > node's dist value:             # O(logV) loops
    swap node with parent                                      # O(1) work
```

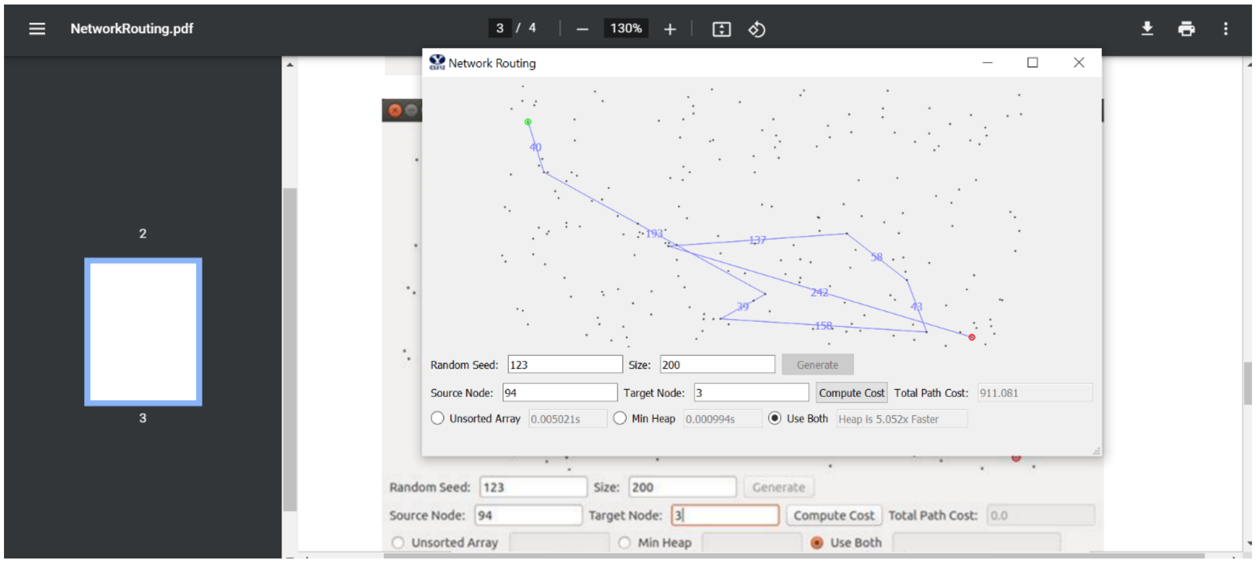
4) Screenshots

a. This one didn't have a path



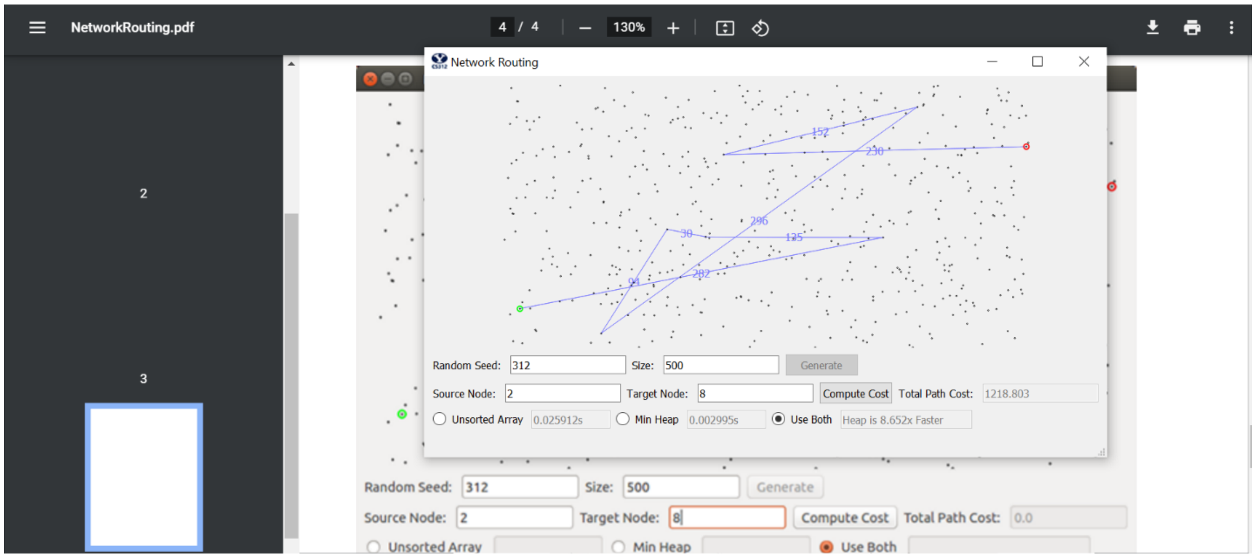
Random Seed: 42 Size: 20
Source Node: 7 Target Node: 1 Total Path Cost: 0.0
Unsorted Array Min Heap [✓ Use Both]

b.



Random Seed: 123 Size: 200
Source Node: 94 Target Node: 3 Total Path Cost: 911.081
Unsorted Array 0.005021s Min Heap 0.000994s [✓ Use Both] Heap is 5.052x Faster

c.



Random Seed: 312 Size: 500 Generate

Source Node: 2 Target Node: 8 Compute Cost Total Path Cost: 1218.803

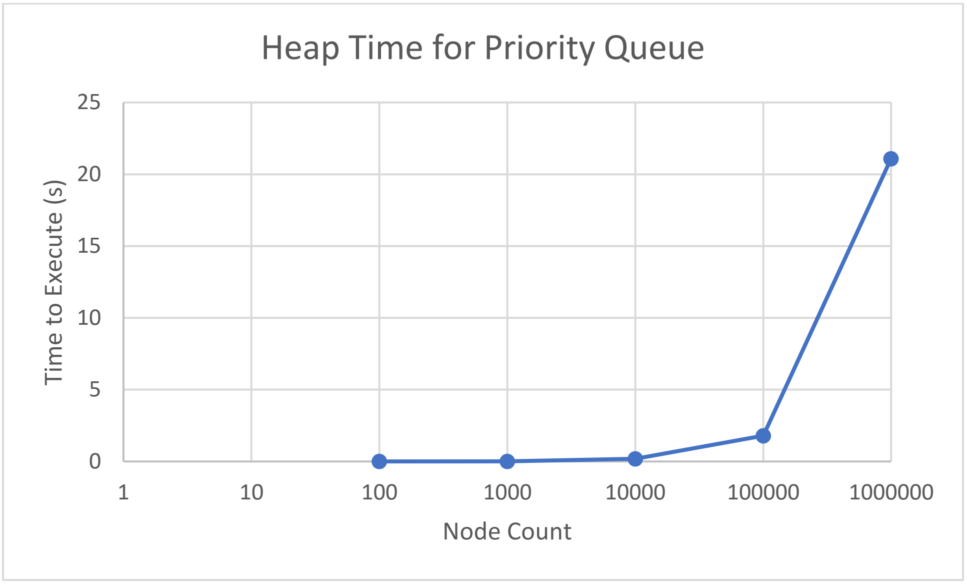
Unsorted Array 0.025912s Min Heap 0.002995s [✓ Use Both] Heap is 8.652x Faster

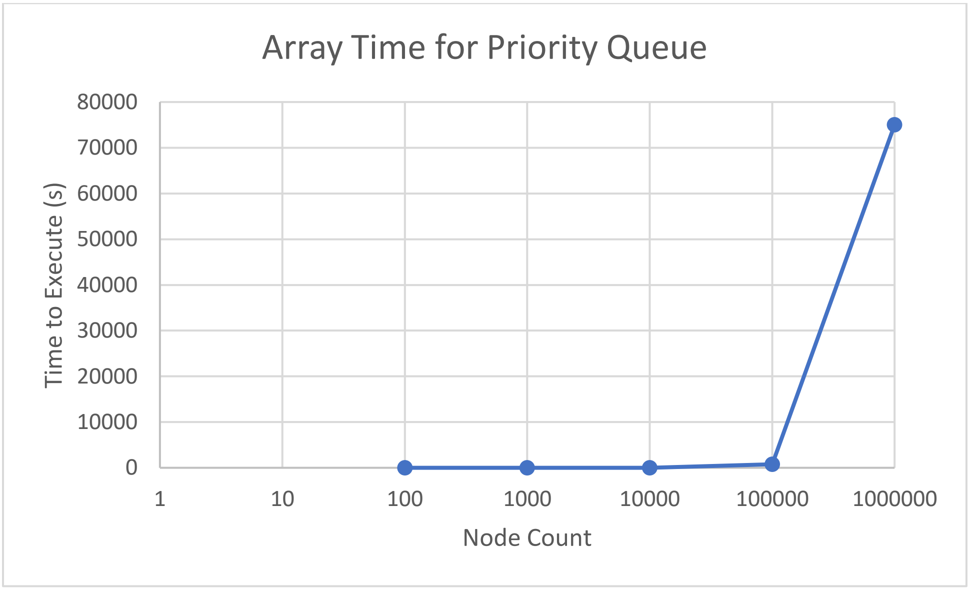
5) Empirical data:

	Heap					
Nodes	Time (seconds)					Avg
100	0.001987	0.000994	0.000946	0.000995	0.002991	0.001583
1000	0.006992	0.00698	0.005982	0.006984	0.005986	0.006585
10000	0.178522	0.17952	0.180517	0.178523	0.180517	0.17952
100000	1.768267	1.563864	1.521007	1.78742	2.334911	1.795094
1000000	20.05434	21.96507	20.23087	22.24149	20.96495	21.09134

	Array					
Nodes	Time (seconds)					Avg
100	0.001991	0.003987	0.003002	0.000999	0.000965	0.002189
1000	0.057837	0.055848	0.055837	0.055855	0.055854	0.056246
10000	8.40047	9.01787	9.784774	7.198721	9.744924	8.829352
100000	843.2309	877.7434	617.7226	787.8818	808.0335	786.9224
1000000					Guess:	75000

Note: I used a base 10 logarithmic scale for the graphs.





The heap implementation went much faster than the array implementation. The heap had a reasonable average time of about 22 seconds for 1 million nodes. I estimated that the 1 million node solution for the array implementation would take about 100 times longer than the average time it took to solve the 100,000 node solution. It was tricky making this estimation though. The time it took to do 10,000 nodes, for example, took about 150 times longer than 1,000 nodes on average. Then from 10,000 nodes to 100,000 nodes, it only took about 90 times longer to do on average. I figured a rough estimate would be around 100 times longer than that for 1 million nodes, but because the algorithm is O(V^2), it might be even longer than that. Regardless, we can easily tell from the data that the difference in implementations makes a massive difference. The O(VlogV) heap priority queue creates a solution within about 20 seconds whereas the O(V^2) array priority queue would solve the same problem in around 24 hours. This is due to our O(logV) deleteMin function that allows us to find the node with the minimum distance instantly and then adjust the graph in O(logV) time. The deleteMin function in the array implementation requires you to sift through the whole array to find the minimum element. Even when it finds the minimum element, it doesn’t know that until it has been compared to every other node in the array, so sifting through V nodes each time we pop something off the priority queue will understandably take a very long time comparatively.