Emily Morrow

1. Dijkstra's algorithm

```
def Dijkstra(self, use_heap):
    dist = dict()
    prev = dict()

    graph = self.network
    vertices = self.network.nodes

    # fill dist and prev
    for v in vertices:
        dist[v.node_id] = float("inf")
        prev[v.node_id] = None

    # set the distance of the source node to 0
    dist[self.source] = 0

    if use_heap:
        queue = HeapPriorityQueue()
    else:
        queue = ArrayPriorityQueue()

    queue.makeQueue(vertices)
    queue.decreaseKey(0, self.source)

    while queue.size() > 0:
        # get the element with shortest distance from the priority queue
        u = queue.deleteMin()

        # check each edge of u to see if there is a shorter path through u
        for edge in u.vertex.neighbors:
            if (dist[edge.dest.node_id] > dist[edge.src.node_id] + edge.length):
                # set the new distance value to the shorter distance value
                dist[edge.dest.node_id] = dist[edge.src.node_id] + edge.length
                # set previous node of destination node to source node
                prev[edge.dest.node_id] = edge.src
                # change the distance value of the node in the priority queue
                queue.decreaseKey(dist[edge.dest.node_id], edge.dest.node_id)
    # set prev so that getShortestPath() has access to it
    self.prev = prev
```

2. Both priority queues array explain complexity (O(1) O(1) O(|V|)) and heap (O(log|V|))

```
class HeapPriorityQueue(PriorityQueue):
    # min heap represented as an array
    heap = []
    indexMap = dict()

    def __init__(self):
        # add an dummy element at index 0 to make array operations easier
        self.heap.append(-1)


    # move i up in the heap until it's smaller than it's child nodes
    def bubbleUp(self, i):
        # while there is still a parent node
        while i // 2 > 0:
            parentI = i // 2
            # if the child dist is smaller than the parent dist then swap the nodes
```

```
            if self.heap[i].dist < self.heap[parentI].dist:
                self.indexMap[self.heap[i].id] = parentI
                self.indexMap[self.heap[parentI].id] = i
                tmp = self.heap[parentI]
                self.heap[parentI] = self.heap[i]
                self.heap[i] = tmp
            i = parentI


    # move i down in the heap until it's larger than it's parent nodes
    def siftDown(self, i):
        while (i * 2) <= len(self.heap) - 1:
            # find the minimum child index
            if i * 2 + 1 > len(self.heap) - 1:
                childI =  i * 2
            else:
                if self.heap[i * 2].dist < self.heap[i * 2 + 1].dist:
                    childI = i * 2
                else:
                    childI = i * 2 + 1
            # if parent dist is bigger than child dist
            if self.heap[i].dist > self.heap[childI].dist:
                # update index in the index map for nodes being swapped
                self.indexMap[self.heap[i].id] = childI
                self.indexMap[self.heap[childI].id] = i

                # swap parent and child nodes
                tmp = self.heap[i]
                self.heap[i] = self.heap[childI]
                self.heap[childI] = tmp
            i = childI


    # fill the queue with the vertices
    def makeQueue(self, vertices):
        for v in vertices:
            node = Node(v.node_id, float("inf"), v)
            self.heap.append(node)
            self.indexMap[node.id] = len(self.heap) - 1


    # place the new element at the bottom of the tree and bubble up
    # (if it's smaller than the parent swap the two and repeat
    def insert(self, dist, vertex):
        self.heap.append(Node(vertex.node_id, dist, vertex))
        self.indexMap[vertex.node_id] = len(self.heap) - 1
        self.bubbleUp(len(self.heap)-1)


    # find node to decrease and bubble it up
    def decreaseKey(self, key, value):
        i = self.indexMap[value]
        self.heap[i].dist = key
        self.bubbleUp(i)


    # return the root value and adjust the tree
    def deleteMin(self):
        retval = self.heap[1]
        self.heap[1] = self.heap[len(self.heap) - 1]
        # delete the min value from index map and update bottom
```

```
        del self.indexMap[retval.id]
        self.indexMap[self.heap[1].id] = 1

        self.heap.pop()
        self.siftDown(1)
        return retval


    def size(self):
        return len(self.heap) - 1
```

in insert everything has a lower time complexity except the call to bubbleUp. bubbleUp traverses up the tree of vertices V and runs in O(log(|V|)) time. In decreaseKey everything runs in lower time complexity except the call to bubbleUp which takes O(log(|V|)) time. In deleteMin the operation that will take the most time is siftDown. SiftDown traverses down the binary tree from parent to smallest child and also takes O(log(|V|)) time.

```
class ArrayPriorityQueue(PriorityQueue):
    nodes = []

    # add vertices to the queue
    def makeQueue(self, vertices):
        for v in vertices:
            node = Node(v.node_id, float("inf"), v)
            self.nodes.append(node)

    # insert node to the queue and sort it from least to greatest distance
    def insert(self, dist, vertex):
        node = self.Node(vertex.node_id, dist, vertex)
        self.nodes.append(node)
        self.nodes.sort(key=lambda x: x.dist, reverse=False)


    # find node to decrease, update it, and sort the array
    def decreaseKey(self, key, value):
        for node in self.nodes:
            if node.id == value:
                node.dist = key
        self.nodes.sort(key=lambda x: x.dist, reverse=False)


    # delete the node with the smallest dist from the array and return it
    def deleteMin(self):
        minNode = self.nodes.pop(0)
        return minNode


    def size(self):
        return len(self.nodes)
```

In insert all the operations are done in constant time so time complexity is O(1). In delete min all of the operations are also done in constant time so the time complexity is O(1). DecreaseKey has a for loop to find the node that needs to be changed which means it will run through every vertex in the nodes array which would run in O(|V|) time.

Emily Morrow

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

```
def Dijkstra(self, use_heap):
    dist = dict()
    prev = dict()

    graph = self.network
    vertices = self.network.nodes

    # fill dist and prev
    for v in vertices:
        dist[v.node_id] = float("inf")
        prev[v.node_id] = None

    # set the distance of the source node to 0
    dist[self.source] = 0

    if use_heap:
        queue = HeapPriorityQueue()
    else:
        queue = ArrayPriorityQueue()

    queue.makeQueue(vertices)
    queue.decreaseKey(0, self.source)

    while queue.size() > 0:
        # get the element with shortest distance from the priority queue
        u = queue.deleteMin()

        # check each edge of u to see if there is a shorter path through u
        for edge in u.vertex.neighbors:
            if (dist[edge.dest.node_id] > dist[edge.src.node_id] + edge.length):
                # set the new distance value to the shorter distance value
                dist[edge.dest.node_id] = dist[edge.src.node_id] + edge.length
                # set previous node of destination node to source node
                prev[edge.dest.node_id] = edge.src
                # change the distance value of the node in the priority queue
                queue.decreaseKey(dist[edge.dest.node_id], edge.dest.node_id)
    # set prev so that getShortestPath() has access to it
    self.prev = prev
```

In the Dijkstra algorithm it starts by filling the dist and prev structures with values. This will run in O(|V|). The next part of the algorithm that will take more time is the while loop at the end.

```
while queue.size() > 0:
    # get the element with shortest distance from the priority queue
    u = queue.deleteMin()

    # check each edge of u to see if there is a shorter path through u
    for edge in u.vertex.neighbors:
        if (dist[edge.dest.node_id] > dist[edge.src.node_id] + edge.length):
            # set the new distance value to the shorter distance value
            dist[edge.dest.node_id] = dist[edge.src.node_id] + edge.length
            # set previous node of destination node to source node
            prev[edge.dest.node_id] = edge.src
            # change the distance value of the node in the priority queue
            queue.decreaseKey(dist[edge.dest.node_id], edge.dest.node_id)
```

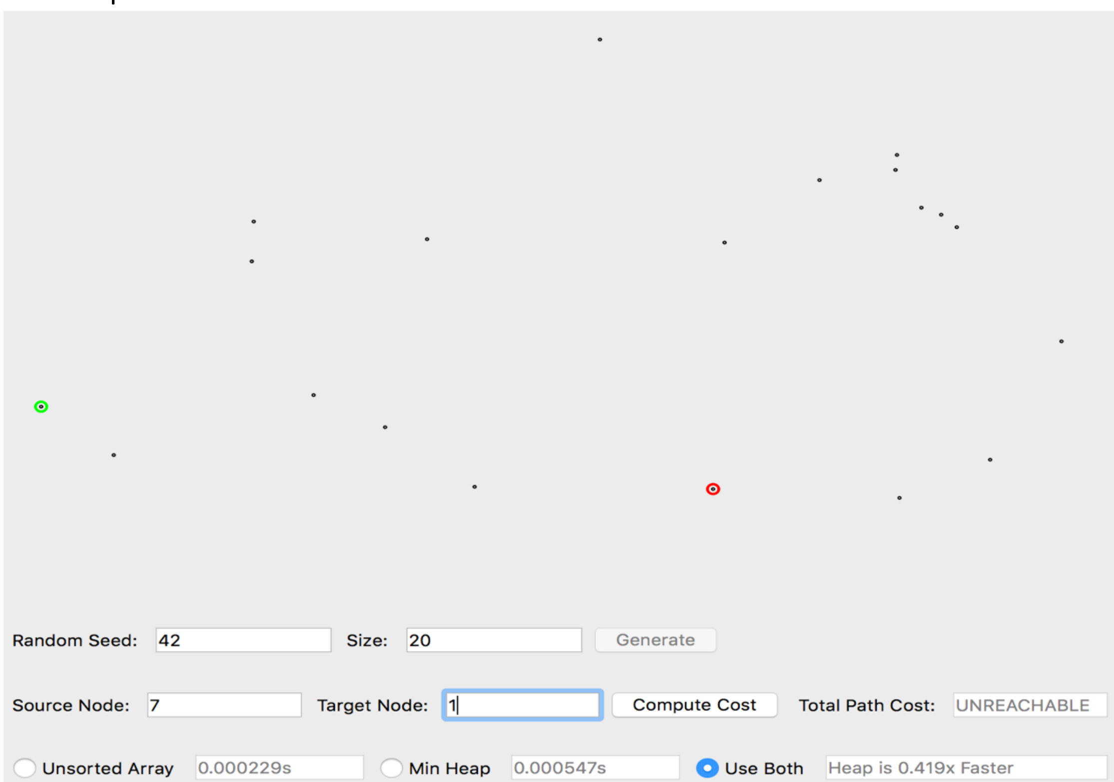
Emily Morrow

In this while loop it will loop |V| times which is the number of vertices. For each iteration of the loop it calls deleteMin on the queue which will either be O(log(|V|)) or O(1) depending on the queue it then loops through the current vertex's neighboring edges and in this for loop decrease key gets called which will either run in O(log(|V|)) or O(|V|) depending on the queue.

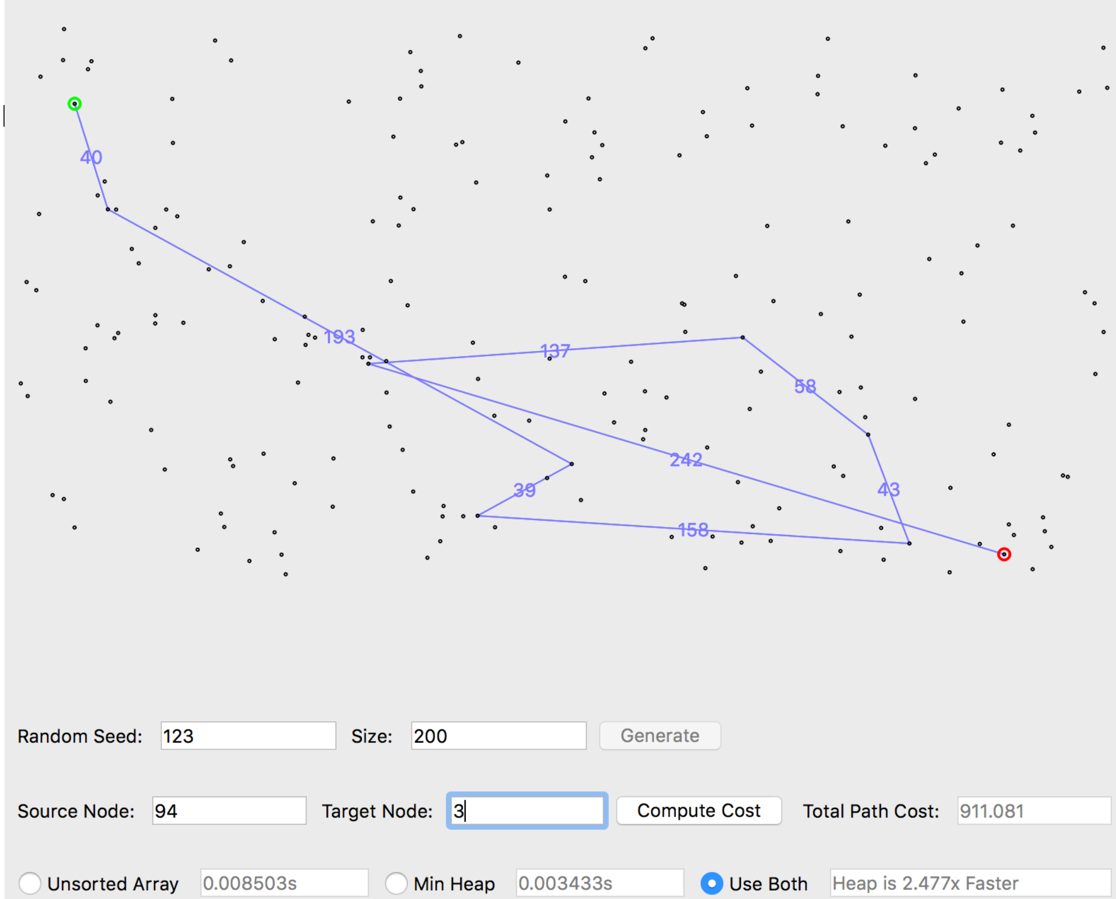
So if we're just looking at the heap implementation the inner loop will run in log|V| + e*log|V|. We ignore log|V| since e*log|V| will grow faster. Then we take into account the outer loop and we get V*elog|V| where e are just the edges that belong to a vertex. This can also be written as Elog|V| where E is all of the edges in the graph. O(|E|log(|V|))

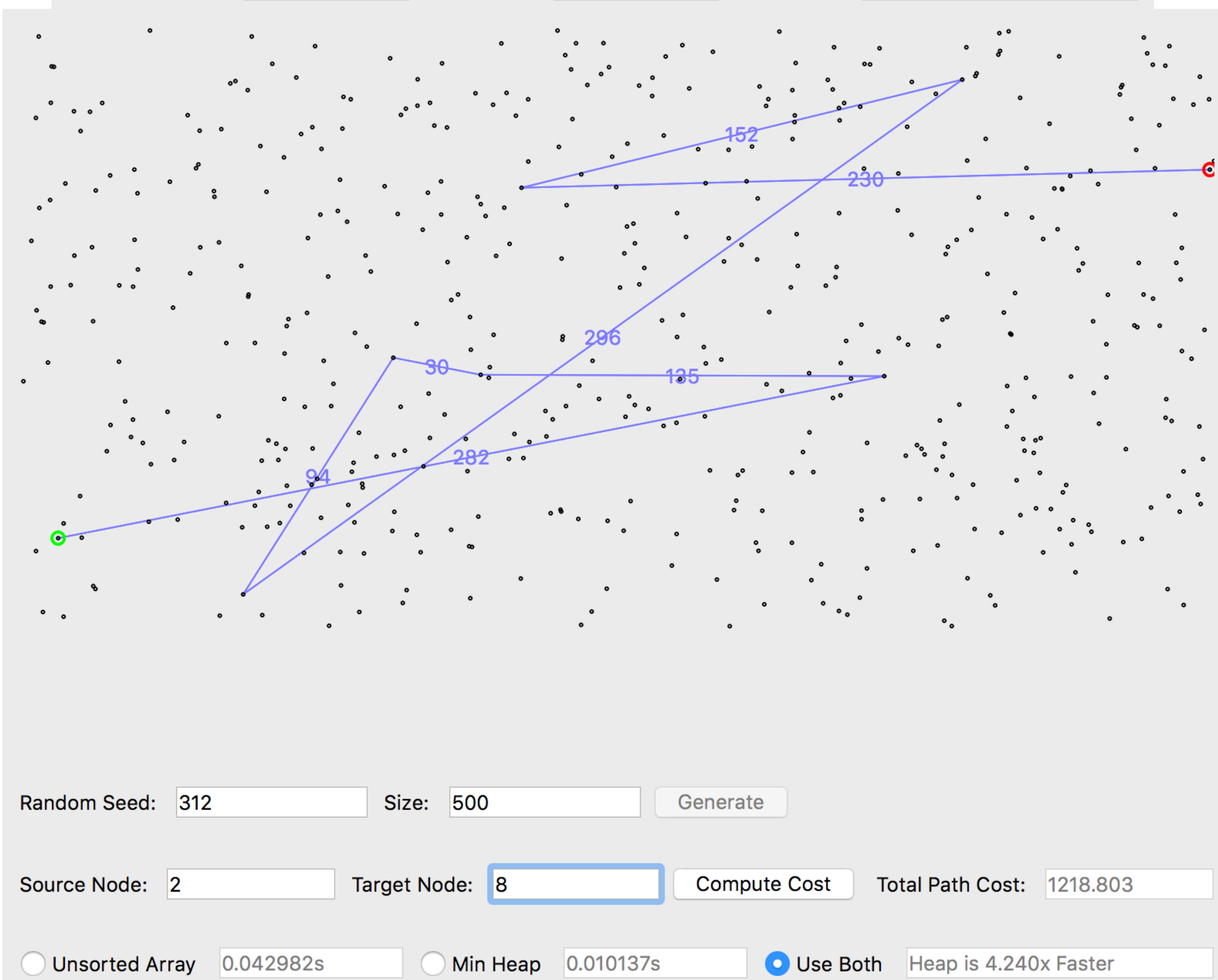
For the array implementation the inner loop will run in e*|V|. When we add the outer loop then it will run in V*e|V| which can be expresses as E|V| where E represents all of the edges in the graph. O(|E||V|)

4. Show pics



Emil


40
193
137
58
242
43
39
158
Random Seed: 123
Size: 200
Generate
Source Node: 94
Target Node: 3
Compute Cost
Total Path Cost: 911.081
Unsorted Array 0.008503s
Min Heap 0.003433s
Use Both
Heap is 2.477x Faster


152
230
296
30
125
94
282
Random Seed: 312
Size: 500
Generate
Source Node: 2
Target Node: 8
Compute Cost
Total Path Cost: 1218.803
Unsorted Array 0.042982s
Min Heap 0.010137s
Use Both
Heap is 4.240x Faster

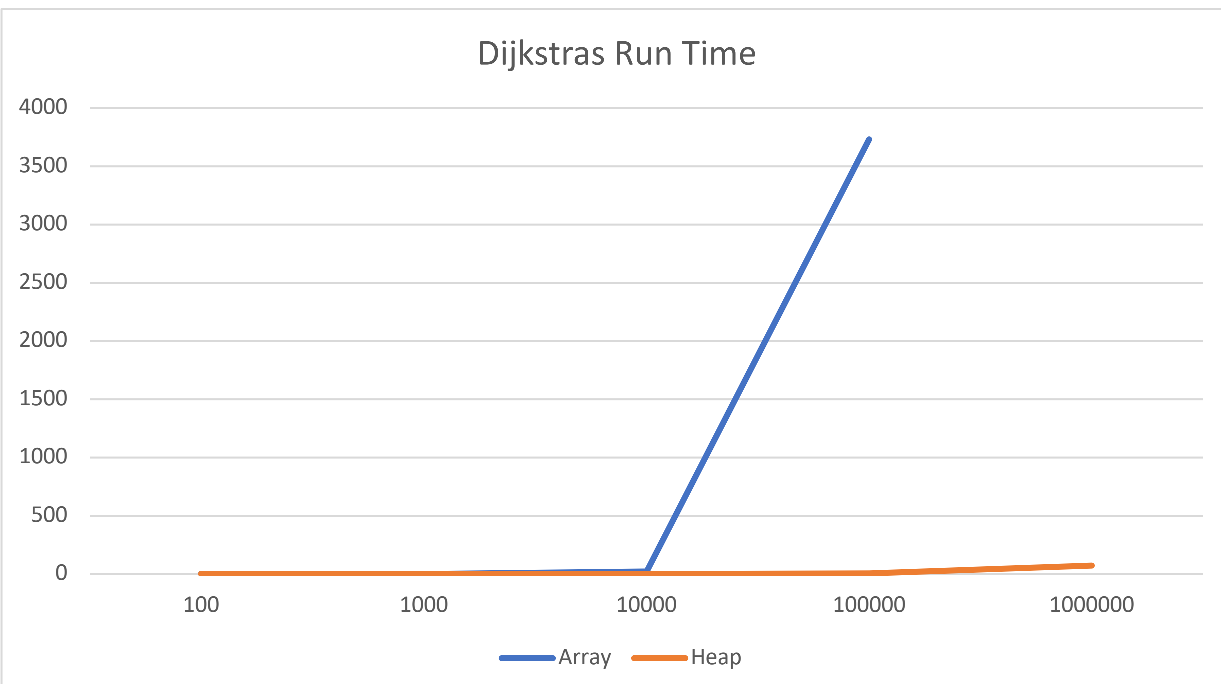
Emily Morrow

5. Graph results, show table of raw data, include your estimate of array 1000000

Raw data

	array					average
100	0.002701	0.002692	0.0024	0.002657	0.002619	0.0026138
1000	0.17113	0.166144	0.171924	0.171814	0.171311	0.1704646
10000	20.679385	20.575158	19.574728	20.519201	19.678774	20.2054492
100000	3721.6198	3795.25918	3730.39109	3725.20499	3776.14984	3749.72498
1000000						
	heap					
100	0.001803	0.00195	0.001907	0.001709	0.002003	0.0018744
1000	0.023109	0.022811	0.022886	0.025815	0.02293	0.0235102
10000	0.388843	0.370448	0.374728	0.374585	0.364342	0.3745892
100000	5.541471	5.805447	5.034962	5.23788	5.200407	5.3640334
1000000	71.051185	69.372794	72.149092	69.98372	70.653321	70.6420224

Graph



The opproximate time it would take to find the shortest path with 1000000 nodes with the array algorithm would be 300000 seconds.

In my results it's clear that the Heap data structure resulted in a much better run time than the array run time. This is because the decreaseKey function runs faster with the heap.