Jeff Reimschussel

Project 3—Network Routing

Code implementation

```
# Define abstract methods priority queues must implement
class PriorityQueue(abc.ABC):
    def __init__(self):
        self.list = []
        self.list_idx = {}

    @abc.abstractmethod
    def makeQueue(self, nodes):
        pass

    @abc.abstractmethod
    def insert(self, entry):
        pass

    @abc.abstractmethod
    def deleteMin(self):
        pass

    @abc.abstractmethod
    def decreaseKey(self, changed):
        pass

    def empty(self):
        return 0 == len(self.list)


# Priority queues contain this data holder object
class PriorityQueueEntry:
    def __init__(self, id, dist, prev):
        self.id = id
        self.prev = prev
        self.dist = dist

    def __str__(self):
        return '(id:%s prev:%s dist:%s)' % (self.id, self.prev, self.dist)


class UnsortedListPriorityQueue(PriorityQueue):
    def __init__(self):
        self.list = {}
```

```
    def makeQueue(self, nodes):
        # Returns the initial list of vertices entered into the queue
        queue = []
        for node in nodes:
            self.insert(PriorityQueueEntry(node.node_id, float('inf'), None))
            queue.append(self.list[node.node_id])
        return queue

    def insert(self, entry):
        self.list[entry.id] = entry

    def deleteMin(self):
        # Search for the minimum value still in the queue
        min = next(iter(self.list))
        min_entry = self.list[min]
        for k in self.list.keys():
            entry = self.list[k]
            if min_entry.dist > entry.dist:
                min = k
                min_entry = entry
        del self.list[min]
        return min_entry

    def decreaseKey(self, changed):
        # Nothing to update except the node value
        self.list[changed.id].dist = changed.dist
        self.list[changed.id].prev = changed.prev


class BinaryMinHeapPriorityQueue(PriorityQueue):
    def makeQueue(self, nodes):
        # Returns the initial list of vertices entered into the queue
        queue = []
        for node in nodes:
            self.insert(PriorityQueueEntry(node.node_id, float('inf'), None))
            queue.append(self.list[node.node_id])
        return queue

    def insert(self, entry):
        idx = len(self.list)
        self.list.append(entry)

        # Update map to enable constant lookup
        self.list_idx[entry.id] = idx
        self.bubbleUp(idx)
```

```
    def deleteMin(self):
        min = PriorityQueueEntry(self.list[0].id,
                                 self.list[0].dist,
                                 self.list[0].prev)
        # Swap minimum with leaf in heap, bubble down to right priority
        back = len(self.list) - 1
        self.swap(0, back)
        del self.list[back]
        self.bubbleDown(0)
        return min

    def decreaseKey(self, changed):
        idx = self.list_idx.get(changed.id)
        self.list[idx] = changed
        self.bubbleUp(idx)

    def swap(self, parent, child):
        # Swaps two nodes
        temp = self.list[parent]
        parent_id = temp.id
        child_id = self.list[child].id
        self.list[parent] = self.list[child]
        self.list[child] = temp
        # Update map to enable constant lookup
        self.list_idx[parent_id] = child
        self.list_idx[child_id] = parent

    def bubbleUp(self, child):
        # The parent of any child (assuming starting at index 1) is child // 2
        parent = (child + 1)//2 - 1

        if parent >= 0 and self.list[parent].dist > self.list[child].dist:
            self.swap(parent, child)
            # Check for continuing to bubble up
            self.bubbleUp(parent)

    def bubbleDown(self, parent):
        # The child of any node (assuming starting at index 1) is parent * 2
        child = parent * 2 + 1

        # Get smaller of two children
        if (len(self.list) > (child + 1) and
            self.list[child + 1].dist < self.list[child].dist):
```

```
            child += 1

        if (len(self.list) > child and
            self.list[parent].dist > self.list[child].dist):
            self.swap(parent, child)
            # Check for continuing to bubble down
            self.bubbleDown(child)

def dijkstra(self, use_heap):
        shortest_paths = {}
        queue = UnsortedListPriorityQueue()
        if use_heap:
            queue = BinaryMinHeapPriorityQueue()
        shortest_paths = queue.makeQueue(self.network.nodes)
        start = PriorityQueueEntry(self.source, 0, self.source)
        queue.decreaseKey(start)

        while not queue.empty():
            next = queue.deleteMin()
            shortest_paths[next.id] = next

            for edge in self.network.nodes[next.id].neighbors:
                id = edge.dest.node_id
                if next.dist + edge.length < shortest_paths[id].dist:
                    shortest_paths[id].dist = next.dist + edge.length
                    shortest_paths[id].prev = next.id
                    queue.decreaseKey(shortest_paths[id])

        return shortest_paths
```

Time and Space Complexity Analysis

In each of the operations, this is assuming that the queue entries are only storing information for each vertex in the graph, not all of the edges. This is why |V| is used to represent the number of entries in the queues; it is the same as the number of vertices in a graph.

insert

The unsorted array has a time complexity of O(1) for insert; the function simply inserts a new entry into the queue at the entry's id.

The space complexity is also 1-to-1 for each entry in the queue; there's no new space required when inserting multiple entries beyond a constant space. Therefore the space complexity is only dependent on the number of entries into the queue, or O(|V|).

In order to ensure a O(1) for lookup in the binary minimum heap priority queue, the function inserts both an entry into the main list, and the entry's index in the main list into a map. After appending to the end of the main list, however, the priority queue may need to "bubble up" the entry if it's found to have a higher priority than the entry's parent. The binary minimum heap maintains a structure of every parent entry having at most 2 children. This means that the number of possible swaps an entry could make when swapping a parent with a child is always cut in half after each swap. Therefore the time complexity for completing the bubbling up portion of inserting a new entry is O(log|V|). Because this is the most expensive aspect of inserting new entries, the overall time complexity for insert in the binary minimum heap priority queue is at worst O(log|V|).

This space complexity is also dependent on the number of entries in this implementation. Each entry requires space in the list, and a map entry to be able to locate that entry. Therefore the space complexity can be reduced to O(|V|).

deleteMin

The unsorted array does not have a good way to locate the entry within the queue with the highest priority. In order to find the highest priority to remove, a search of all entries still in the queue has to be made. After locating the highest priority entry, deleting it takes O(1) time, because there is no reordering of the queue after deleting an entry. This makes searching the queue the most expensive aspect of deleteMin, meaning deleteMin has a time complexity of O(|V|).

There's no new space needed at a call of deleteMin, so the space complexity is O(1).

Because the entry with the highest priority is always the root of the heap, locating the entry to delete is straightforward. In order to maintain the heap priority structure, a leaf node in the heap is swapped with the root and the original root entry is removed. Next, the leaf node in the root index must be "bubbled down" the heap to the right place to maintain the binary minimum heap structure. Bubbling

down is very similar to bubbling up; a parent node is compared with the higher priority of its two children, swapped if the child has a higher priority, and then the original parent is again compared with the new children. Entries compared are only between parent and one of its two children, meaning the most possible swaps and rechecks that can take place is O(log|V|). Because bubbling down is the most expensive part of deleteMin, the time complexity is O(log|V|).

Still no new space stored in this implementation. O(1).

decreaseKey

Because the unsorted list has O(1) lookup time, and because there's no restructuring for adjusting entry priorities within the queue, decreaseKey has a time complexity of O(1)—the entry to change is directly changed.

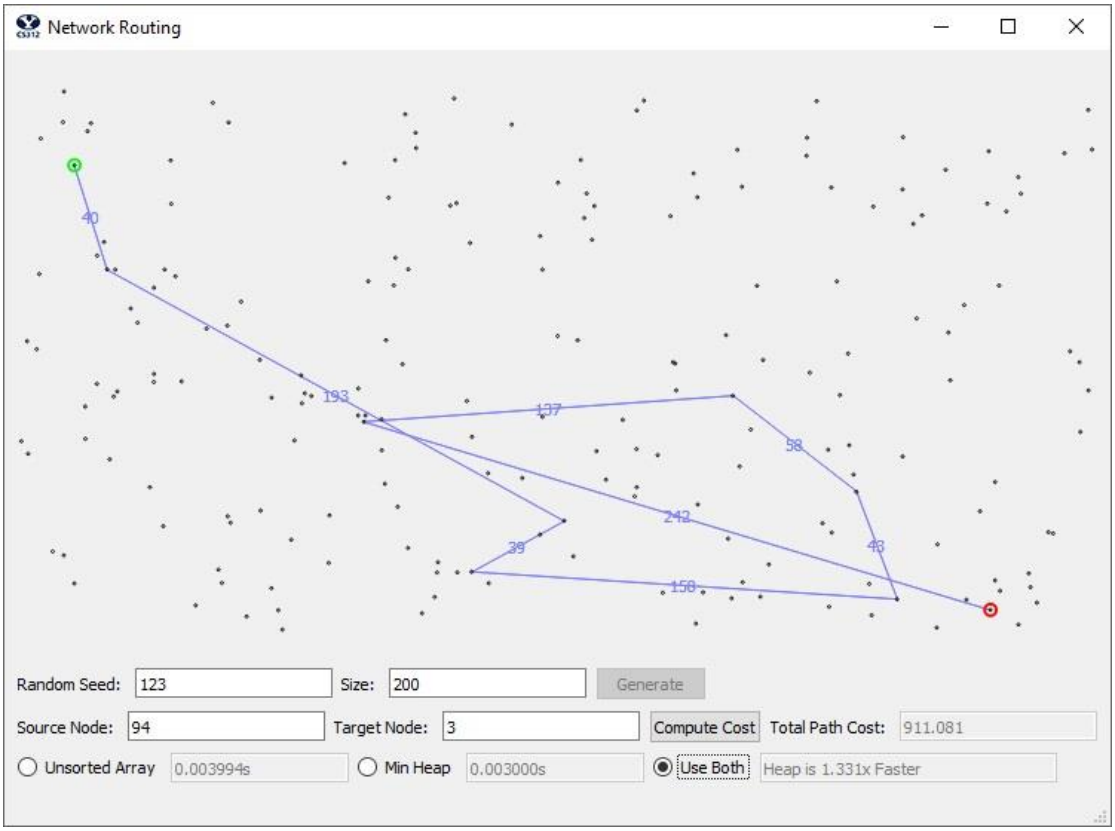
There is no new space stored in this implementation for decreaseKey. O(1).

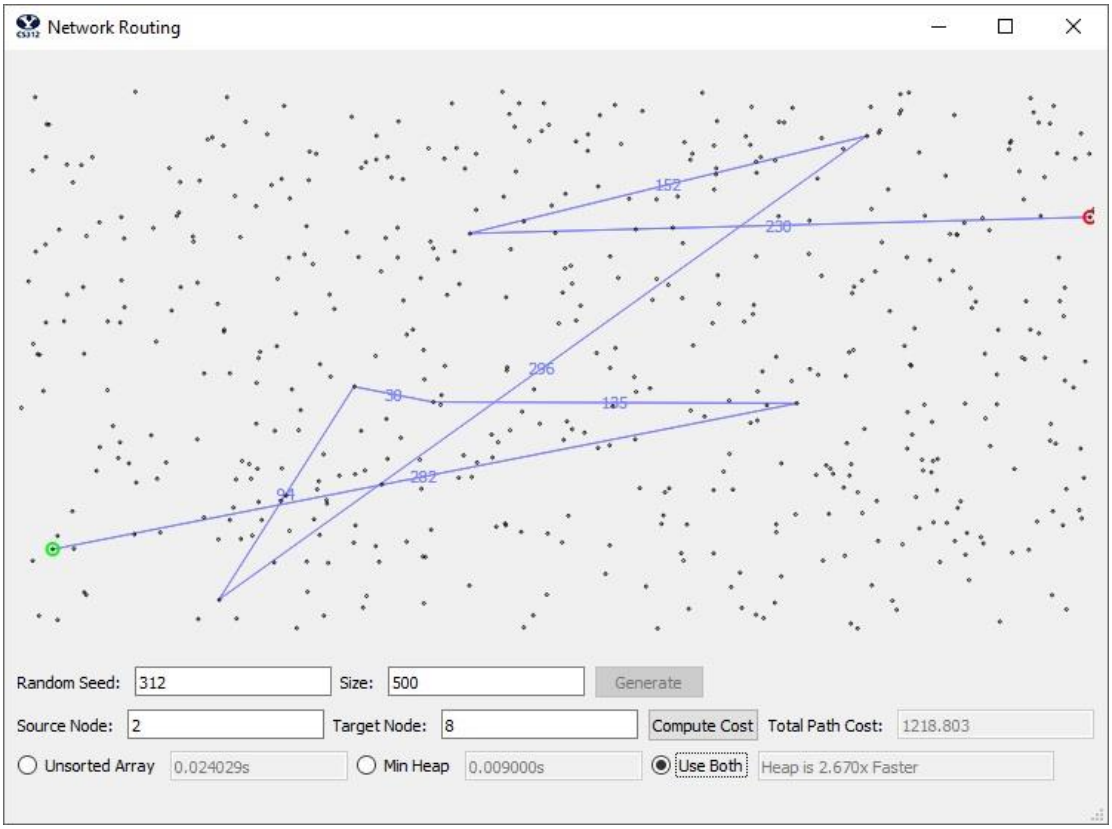
The heap implementation does have O(1) lookup time, but it does need to restructure to maintain the binary minimum heap structure. After adjusting an entry priority directly, the heap must check if that value now needs to bubble up. There's no need to check if it must bubble down because the only adjustments to entries within a queue are to improve an entry's priority, never to lower it. Because bubbling up is also the most expensive part to decreaseKey, the time complexity is also O(log|V|).

Still no new space stored for decreaseKey. O(1).

After looking through each of the operations in both priority queue implementations, it is apparent that the overall time complexity is dependent primarily on the number of entries. The worst case for the unsorted array comes from its deleteMin, with a O(|V|). The worst case for the binary minimum heap comes from bubbling up or down entries, which is used in its insert, deleteMin, and decreaseKey. Therefore the time complexity is O(log|V|).

Screenshots

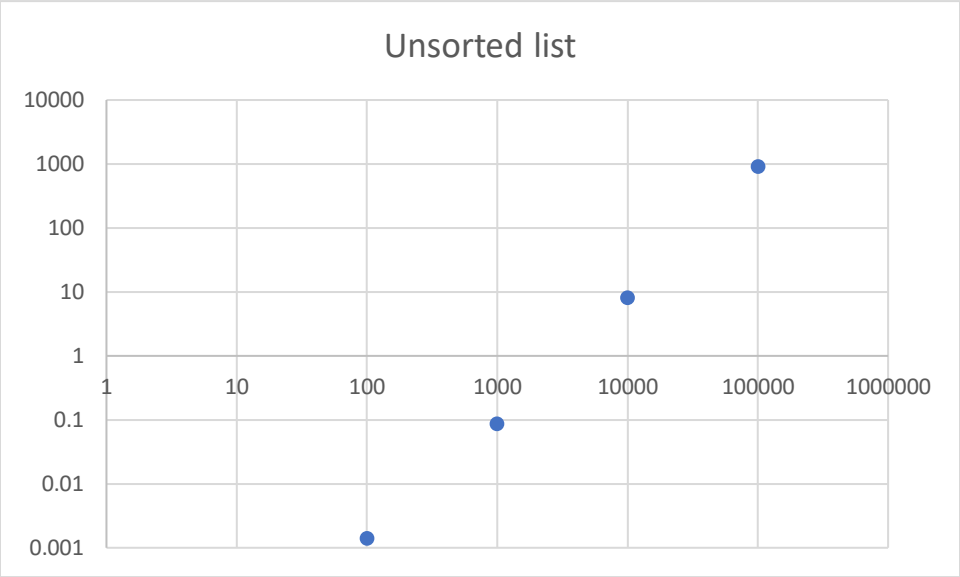

Network Routing
40
193
137
58
242
39
48
150
Random Seed: 123
Size: 200
Generate
Source Node: 94
Target Node: 3
Compute Cost
Total Path Cost: 911.081
Unsorted Array
0.003994s
Min Heap
0.003000s
Use Both
Heap is 1.331x Faster


Network Routing
152
230
296
30
135
282
94
Random Seed: 312
Size: 500
Generate
Source Node: 2
Target Node: 8
Compute Cost
Total Path Cost: 1218.803
Unsorted Array
0.024029s
Min Heap
0.009000s
Use Both
Heap is 2.670x Faster

Time complexity tests

Unsorted List

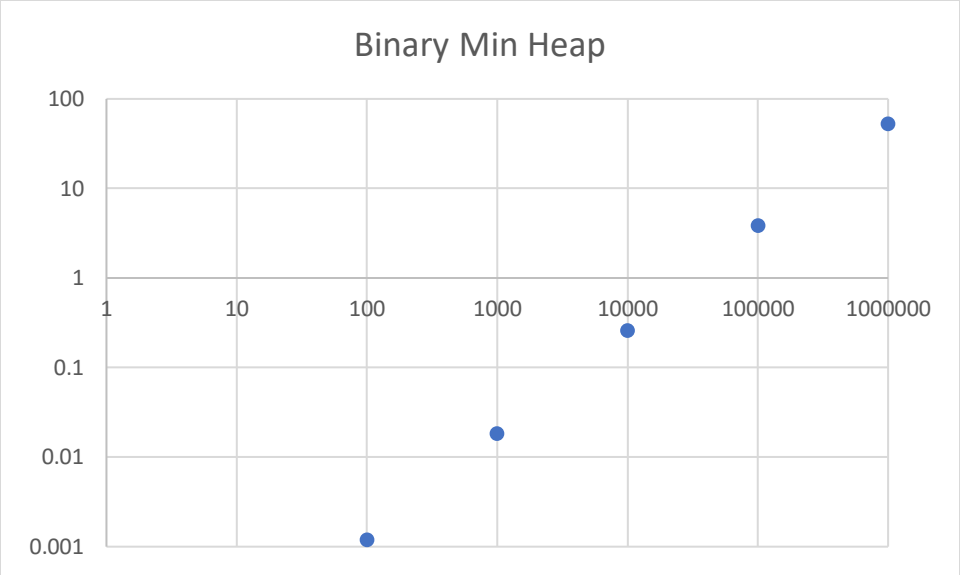
Node count	Test run					
	1	2	3	4	5	Average
100	0.002013	0.001983	0.000993	0.000998	0.001028	0.001403
1000	0.086020	0.087991	0.089026	0.089013	0.086017	.0876134
10000	8.248960	8.276991	8.283018	8.317017	8.199953	8.2651878
100000	886.789167	890.266031	920.024997	961.650983	915.116964	915.7696284



When projecting out an expected run time for computing Dijkstra's on 1,000,000 nodes using an unsorted list, this graph suggests a runtime of around 10,000 seconds. Every time the amount of points is multiplied by 10, the runtime is also multiplied by roughly 10, suggesting a linear time complexity. This is exactly what was expected when computing the biggest time complexity of this priority queue; that is, that it runs in O(|V|).

Binary Min. Heap

Node count	Test run (in seconds)					
	1	2	3	4	5	Average
100	0.000999	0.001002	0.000998	0.001996	0.000997	0.0011984
1000	0.018996	0.017994	0.017997	0.018022	0.019000	0.0184018
10000	0.269024	0.254000	0.252000	0.253008	0.259011	0.2574086
100000	3.837003	3.786991	3.841024	4.075006	3.784002	3.8648052
1000000	50.436939	50.338431	52.849001	55.172690	52.939029	52.3456226



As expected, the runtime for the binary minimum heap implantation was not a linear relationship with the number of vertices in the graph. As the number of vertices increased by a factor of ten, the factor by which runtime also increased was decreasing. This suggests a logarithmic relationship rather than a linear one, which is to be expected from the code analysis.