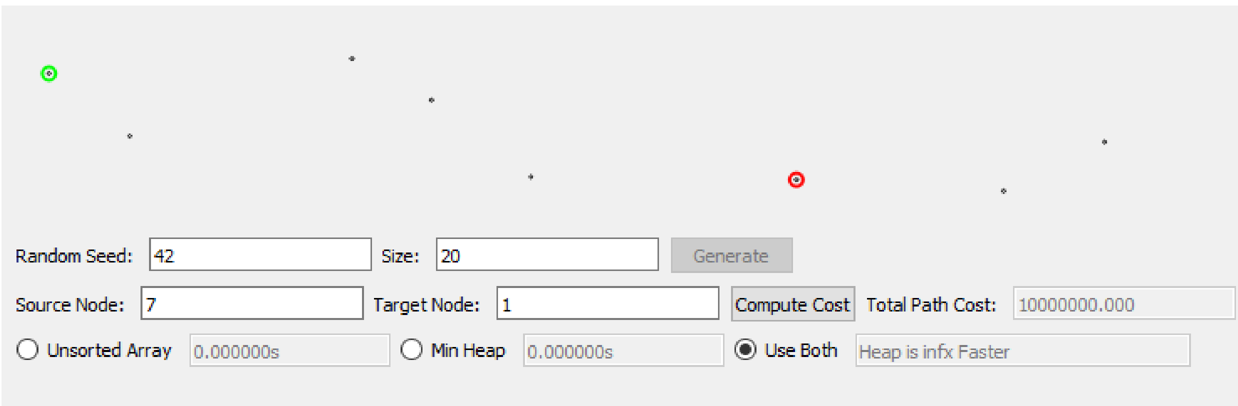
Thomas Neyman
CS 4412 - Advanced Algorithms
March 1 2022
Project 3

Dijkstra’s Shortest Path

Shortest Path Specific Examples:

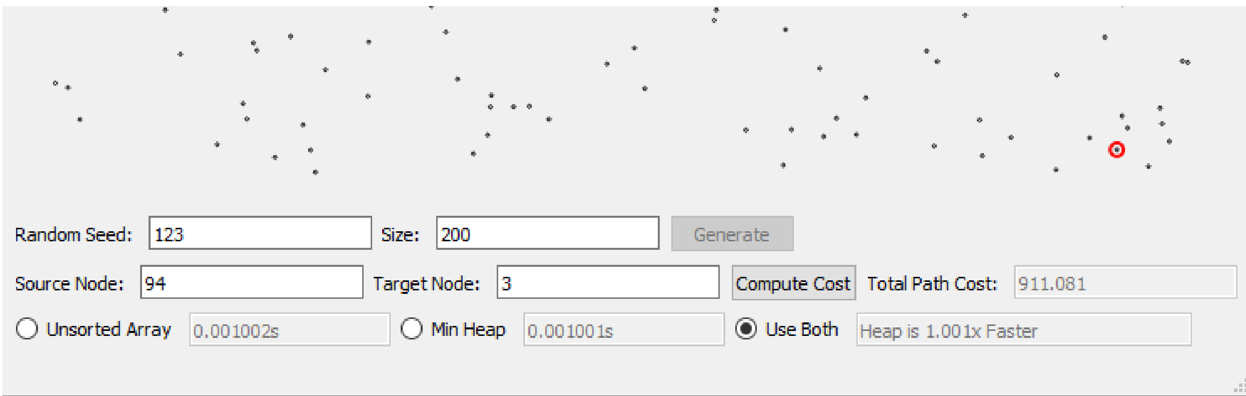
Random Seed 42 - Size 20, using node 7 to node 1:



Node	Distance from Source
1	10000000
2	507
3	308
4	441
5	462
6	98
7	0
8	258
9	430
10	725
11	665
12	449
13	388
14	650
15	338
16	338
17	268
18	162
19	346
20	355

From the shown table of values, we can see that the distance from Node 7 to Node 1 is 10,000,000 (My implementation of infinity) so there is no shortest path that goes from 7 to 1.

Random Seed 123 - Size 200, using Node 94 to Node 3:

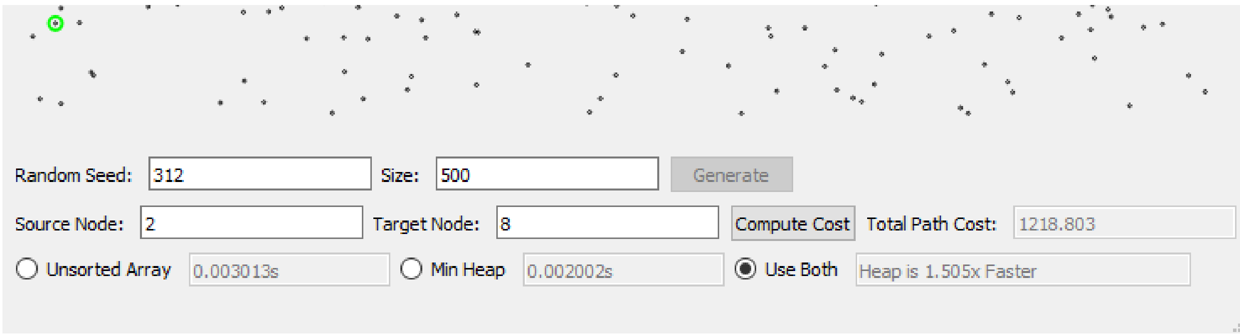


89	661
90	548
91	808
92	675
93	478
94	0
95	487
96	706
97	625
98	593
99	602
100	524

Node	Distance from Source
1	702
2	634
3	911
4	651
5	10000000
6	603
7	562
8	666
9	761

From the table of values we can see the starting Node 94 has a shortest path of 911.0.81 to the Destination Node 3.

For Random Seed 312 - Size 500, using Node 2 to Node 8:



Node	Distance from Source
1	672
2	0
3	1266
4	1443
5	728
6	684
7	1082
8	1218
9	816
10	898
11	1167
12	1087

From the table of values we can see that the shortest path length from Node 2 to Node 8 is 1218.803

Time and Space Complexity:

```
        # subsection 1
        n = len(self.network.nodes)
        distances = []
        heap = CS4412Heap()
```

Initializing the heap data structure as well as creating the list to hold distance values all come down to O(1) time operations as well as O(1) space complexity meaning the sizes are constant.

```
        # subsection 2
        for i in range(n):
            distances.append(10000000)
            heap.array.append(heap.createNode(i, distances[i]))
            heap.pos.append(i)
```

A for loop will run for *V* times where *V* is the amount of vertices in the given network. The amount of vertices will be denoted as n because that is the input of our algorithm. Inside the loop the distances list is appended to n times, and the heap is appended to n times twice. This is 3 separate n operations in the loop, resulting in a O(3n) time complexity which just simplifies to O(n). The space complexity is now made up of 3 separate lists of size n, therefore the space is now O(3n) as well.

```
        # subsection 3
        heap.pos[srcIndex] = srcIndex
        distances[srcIndex] = 0
        heap.decreaseKey(srcIndex, distances[srcIndex])
        heap.size = n
```

All of the operations here are constant time except for the decreaseKey() function. The decreaseKey function complexity is described below. The space complexity does not change after this subsection. The data structure's values are only altered, not changed in size.

```
    def decreaseKey(self, v, dist):
        i = self.pos[v]
        self.array[i][1] = dist
        # this is an O(Log n ) loop
        while i > 0 and self.array[i][1] < self.array[(i - 1) // 2][1]:
            # swap the node and its parent
            self.pos[self.array[i][0]] = (i - 1) // 2
            self.pos[self.array[(i-1) // 2][0]] = i
            self.swapNode(i, (i - 1) // 2)
            # move to parent index
            i = (i - 1) // 2
```

Getting the index of the position list is constant. Setting the distance value of the node is constant. When the while loop begins, it will iterate for O(log n) times. This is because the heap is trickling down its contents to the front, not going through every single value inside the heap. Swapping nodes inside the loop is constant as well as setting values of nodes. So the total time complexity of this function is O(log n) while the space complexity remains unchanged.

```
        # subsection 4
        while heap.isEmpty() == False:
            newHeapNode = heap.deleteMin()
            current = newHeapNode[0]

            for edge in self.network.nodes[current].neighbors:
                destination = edge.dest

                if heap.isInHeap(destination.node_id) and distances[current] !=
10000000 and edge.length + distances[current] < distances[destination.node_id]:
                    distances[destination.node_id] = edge.length +
distances[current]
                    heap.decreaseKey(destination.node_id,
distances[destination.node_id])
```

The while loop is going to iterate for $V + E$ amount of times because it needs to account for every vertex and each one of its edges. This gives a time complexity of O(V+E) so far. Deleting the min value is a constant time function (shown below) and setting the current node value is constant. The space complexity to this point will be 1 less in the heap array for each iteration, but will remain O(n).

The next stage is the nested for loop. Normally a nested loop would create an exponential time complexity, but the for loop only utilizes the isInHeap() function which is constant as well as the decreaseKey() function which has already been shown to be O(log n) time. This means that in total, the nested loops are compounded by O($V + E$) * O(Log V). E and V are both the same degree in terms of big O, so either can be used, but there will usually be a greater number of edges, so we use E when describing the time complexity as O(E Log n) where n is the size of the input (V).

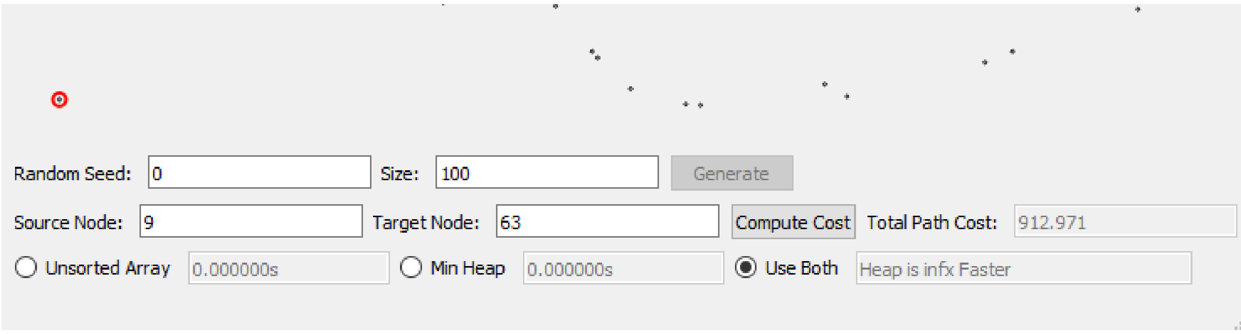
By the end of the while loop, the heap's array is completely deleted, leaving only 2 data structures (The list of distances and the list of positions in the heap) giving a space complexity of O(2n) or just O(n).

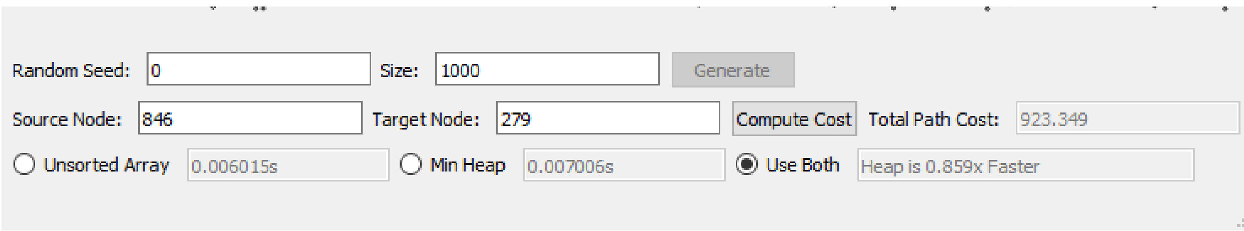
```python
    def deleteMin(self):
        if self.isEmpty():
            return

        root = self.array[0]
        lastNode = self.array[self.size - 1]
        self.array[0] = lastNode
        self.pos[lastNode[0]] = 0
        self.pos[root[0]] = self.size - 1
        self.size -= 1
        self.minHeapify(0)
        return root
```
Proof that the deleteMin() function runs at a constant time. Each operation is an assignment or access call of some sort, all taking O(1) time. The minHeapify() function is also only comprised of constant time operations.
```python
    def minHeapify(self, index):
        shortest = index
        left = 2*index + 1
        right = 2*index + 2
        if left < self.size and self.array[left][1] < self.array[shortest][1]:
            shortest = left
        if right < self.size and self.array[right][1] < self.array[shortest][1]:
            shortest = right
        if shortest != index:
            self.pos[self.array[shortest][0]] = index
            self.pos[self.array[index][0]] = shortest
            self.swapNode(shortest, index)
            self.minHeapify(shortest)
```

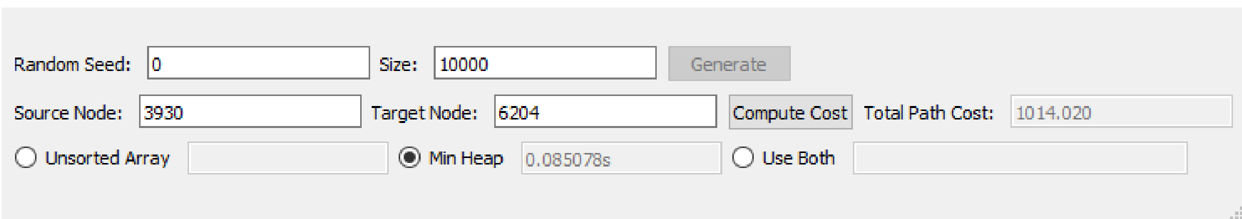
Time Comparison for Different Node Amounts:

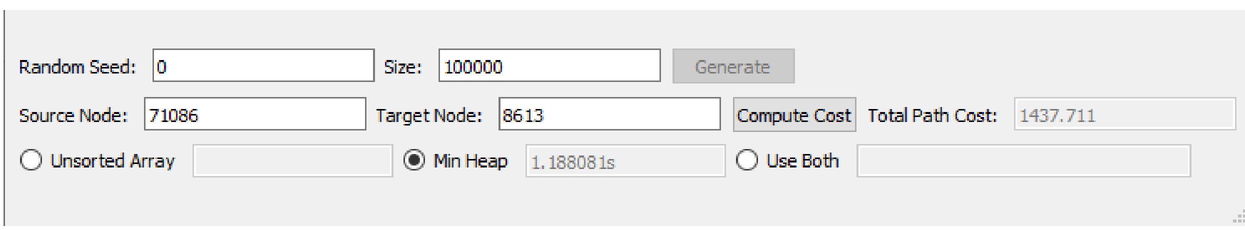
Times for heap implementation:

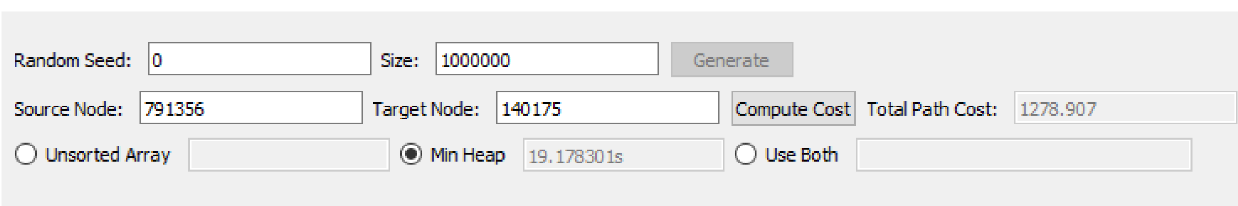
100:	0 seconds
1000:	0.007 seconds
10,000:	0.085 seconds
100,000:	1.188 seconds
1,000,000:	19.178 seconds

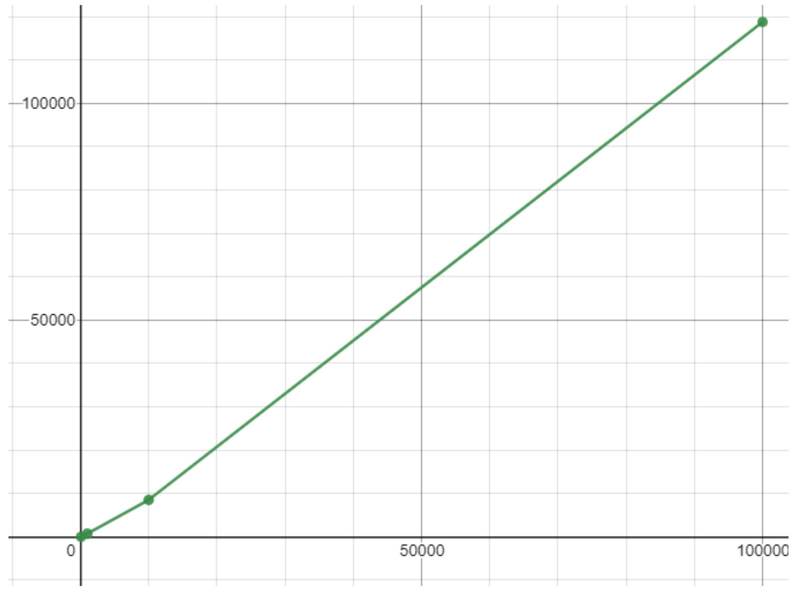

Random Seed: 0
Size: 100
Generate
Source Node: 9
Target Node: 63
Compute Cost
Total Path Cost: 912.971
Unsorted Array 0.000000s
Min Heap 0.000000s
Use Both
Heap is infx Faster


Random Seed: 0
Size: 1000
Generate
Source Node: 846
Target Node: 279
Compute Cost
Total Path Cost: 923.349
Unsorted Array 0.006015s
Min Heap 0.007006s
Use Both
Heap is 0.859x Faster

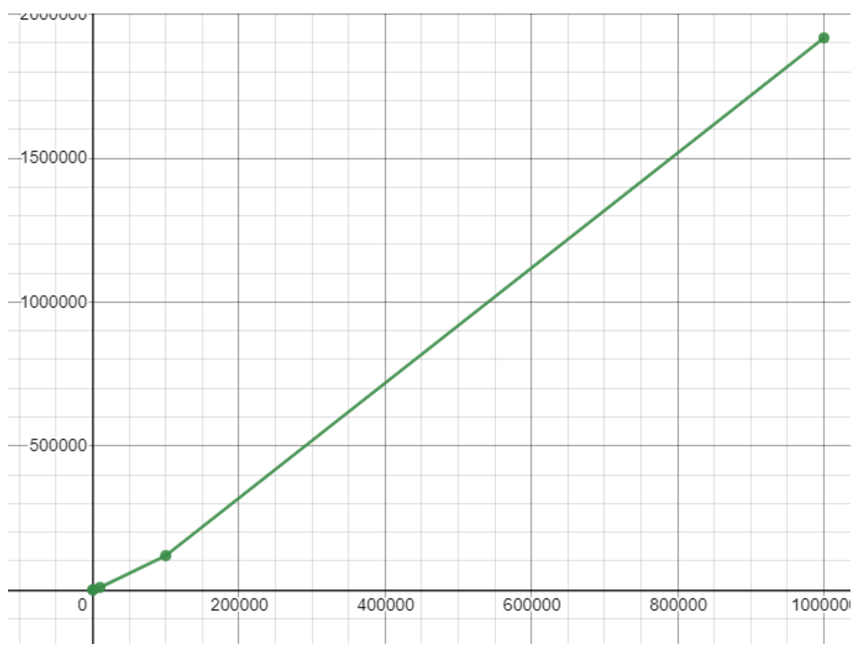

Random Seed: 0
Size: 10000
Generate
Source Node: 3930
Target Node: 6204
Compute Cost
Total Path Cost: 1014.020
Unsorted Array
Min Heap 0.085078s
Use Both


Random Seed: 0
Size: 100000
Generate
Source Node: 71086
Target Node: 8613
Compute Cost
Total Path Cost: 1437.711
Unsorted Array
Min Heap 1.188081s
Use Both


Random Seed: 0
Size: 1000000
Generate
Source Node: 791356
Target Node: 140175
Compute Cost
Total Path Cost: 1278.907
Unsorted Array
Min Heap 19.178301s
Use Both



When graphing the time to the number of nodes processed, it almost appears as though the algorithm is running at a linear speed. (y values increased by a factor of 100,000 in order to see the value changes against such high x values)



Accounting for going up to 1,000,000 it's easier to see the rate at which the algorithm processes the input. Once the input size reaches 1,000,000 it takes about 20 seconds. 20 upscaled by 100,000 to 2,000,000 is about double the size of the input length. This is closer to the O(n log n) time that is expected.