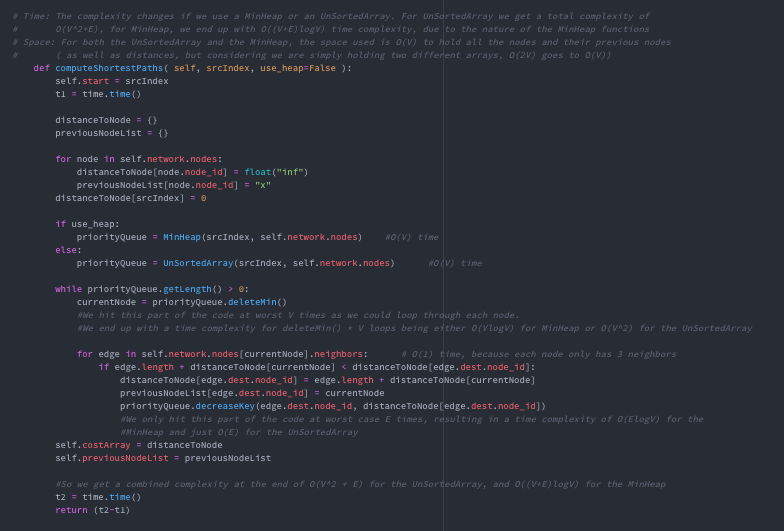
Project 3

Dijkstra’s algorithm:

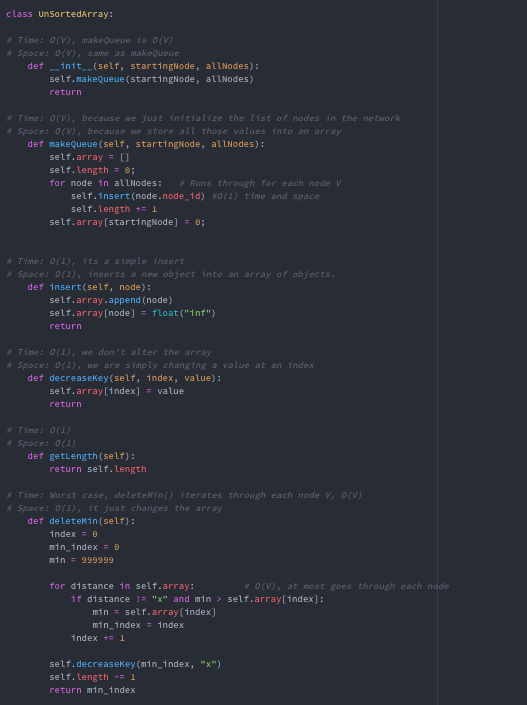


For each node popped from priority queue, check all edges and see if the distance to that edge from the current node is less than the distance stored, if so, add the new distance, and make that the new distance, and update the queue (deleteMin())

Getting the shortest path:



Start from the destination node, and then add for the path each previous node until we get to the starting node. If the previous node is “x”, or the distance is “x”, then that means there is no possible route from the destination node to the source node. Else, just make the path array, and also return the distance to the destination node.

Unsorted Array Implementation:

MakeQueue():

MakeQueue() runs in O(V) time because it needs to iterate through each node in the graph, and calls insert() for each one. Insert() is O(1), so total we get O(V) for setting up the initial array.

Insert():

Insert() runs in O(1) time, because it only appends a value at the end of an array (O(1)) , and changes it to infinity

DeleteMin():

DeleteMin() runs in O(V) time, because it needs to iterate through each element in the queue to find the minimum index, and return that. Each time it is called, it will go through each element in the array

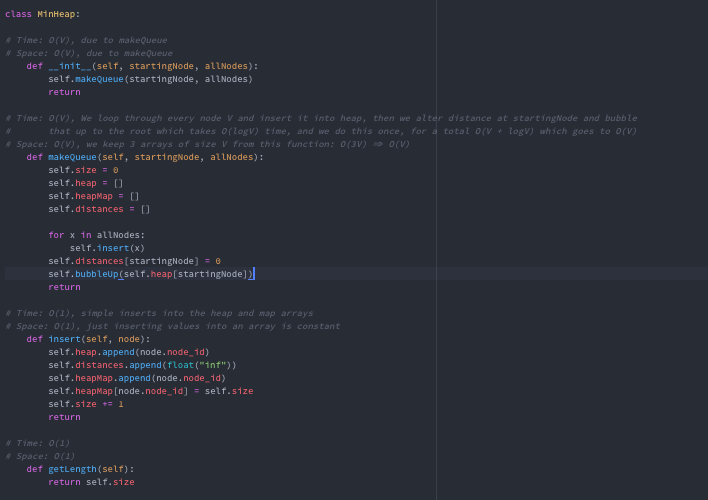
DecreaseKey():

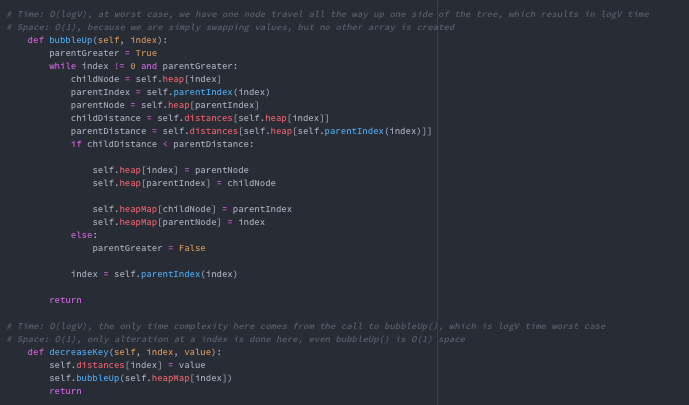
DecreaseKey() is O(1), because it simply goes to an index in the array, and alters it which is constant time with a given index.

Total Complexity:

The total complexity for Dijkstra’s with an unsorted array is O(V^2 + E) as explained in the code comments above my algorithm. Because each call to DeleteMin() is O(V), and Dijkstras at worst case needs to iterate over each node V, we get a complexity of O(V^2), adding on top of that the number of edges of each node with is E, so our final complexity is O(V^2 + E)

Min Heap Implementation:





MakeQueue():

MakeQueue() runs in O(V) time, because it populates 3 arrays by iterating through each node in the graph once, and calling Insert(). After populating the array, it calls bubbleUp() once, which runs in O(logV) time, and bubbles the starting node up to the top of the heap to the root. Because we only call bubbleUp() once after the heap has been initialized, we don’t add that complexity to the total equation, and we end up only bubbling up once, which ends us with a complexity of O(V)

Insert():

Insert() is only O(1) time, because it only appends and alters the arrays with the new elements, which is a collection of O(1) complexity function calls.

DeleteMin():

DeleteMin() runs in O(logV) time. We first swap the first element with the last before deleting it so that the python delete only run in O(1) time as it deletes the last element of the list. Then we trickle down the last element, with at worst case we trickle it down all the way to the bottom of the tree, which takes O(logV) time, as we only would be trickling it down one side of the tree, the side that the minimum child is on each time.

DecreaseKey():

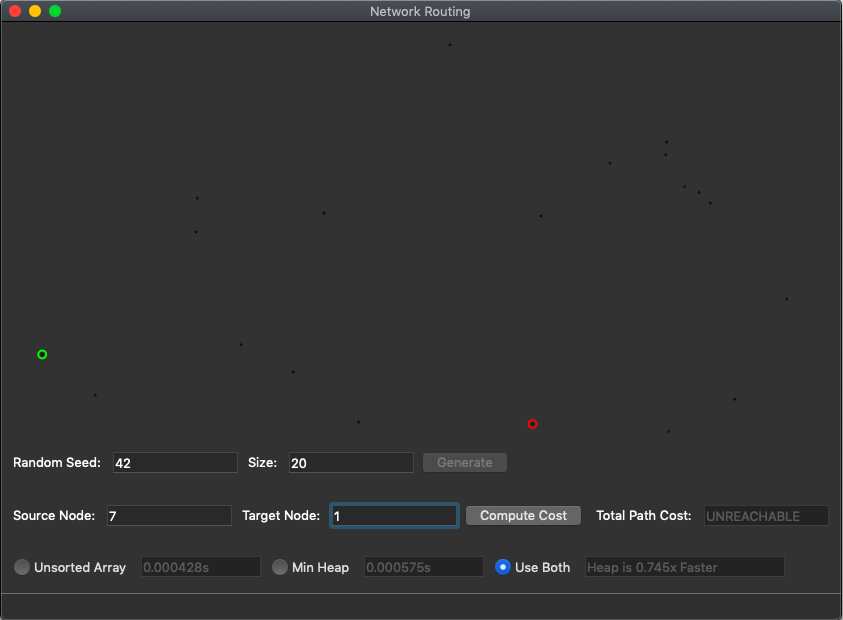
DecreaseKey() runs in O(logV) time. The bulk of the complexity comes from the call to bubbleUp() after the value in the heap is altered, in order to put it in the new position in the heap

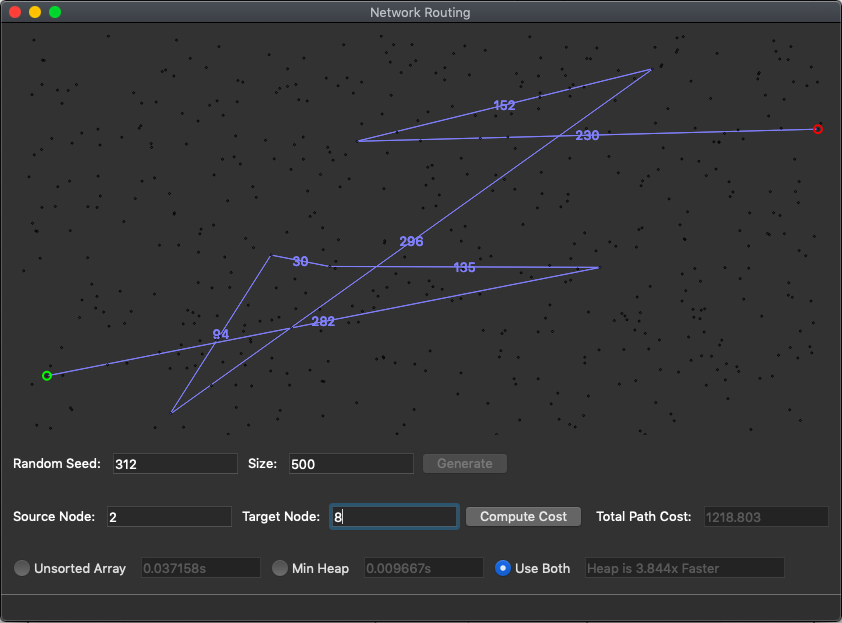
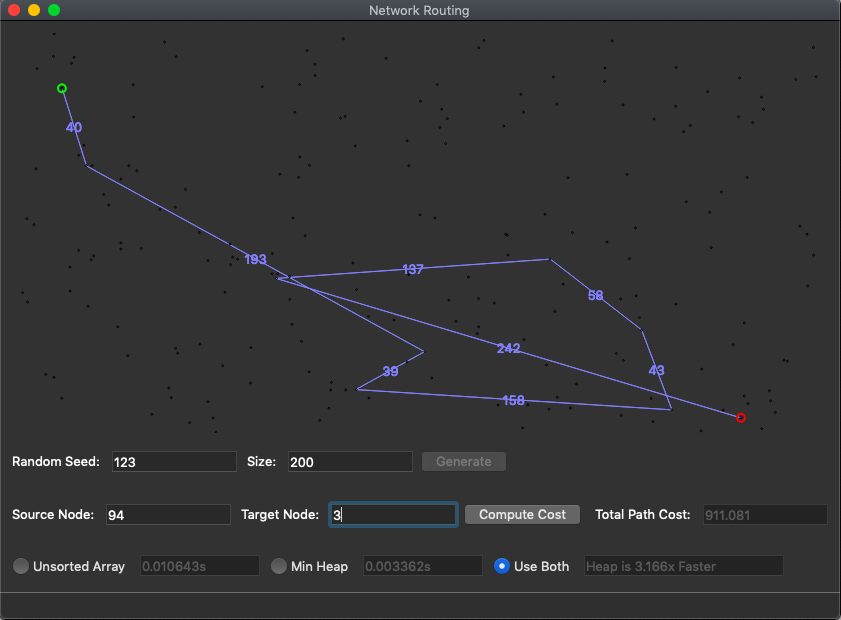
bubbleUp():

bubbleUp() runs in O(logV) time. The worst case time complexity is logV because at worst case we take a node from the bottom of one side of a tree and swap it all the way to the root of the tree, which takes logV swaps.

Total Complexity:

The total complexity for Dijkstra’s with a MinHeap is O((V+E)logV) as explained in the code comments above my algorithm. Because each call to DeleteMin() is O(logV), and Dijkstras at worst case needs to iterate over each node V, we get a complexity of O(VlogV), adding on top of that the number of edges of each node with is E, and each call to DecreaseKey() is O(logV) complexity, we get a final complexity of O(VlogV + ElogV) or O((V+E)logV)

Expected Output:



Raw Data:

n: 100

Times:

0.003717, .002728, 1.363x faster

0.004861, .003324, 1.462x

0.005171, .002391, 2.163x

0.003636, .003289, 1.105x

0.005403, .002391, 2.260x

Average: 0.0045576, 0.0028246, 1.6135x faster

---------------

n: 1000

Times:

0.141491, .024973, 5.666x

0.155772, .019490, 7.992x

0.138188, .019418, 7.116x

0.132291, .019429, 6.809x

0.132030, .019488, 6.775x

Average: 0.1399544, 0.0205596, 6.807x faster

---------------

n: 10000

Times:

13.256818, .286239, 46.314x

15.524566, .326156, 47.599x

13.799562, .297933, 46.318x

14.499669, .320354, 45.261x

13.458846, 0. 273931, 49.132x

Average: 14.1078902, .3008984, 49.132x faster

---------------

n: 100000

Times:

1497.429437, 3.887690, 385.172x

1515.358205, 4.211106, 359.848x

1599.383380, 4.249279, 376.389x

1593.81258, 4.662545, 341.833x

1605.49502, 4.305502, 372.894x

Average: 1562.2957244, 4.263224, 366.459x faster

---------------

n: 1000000

Times:

58.394533

58.955866

57.992041

59.302918

57.059681

Average: 58.3410338

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | N = 100 |  | N = 1000 |  | N = 10000 |  | N = 100000 |  | N = 1000 | Estimate |
|  | Array | Heap | Array | Heap | Array | Heap | Array | Heap | Heap |  |
|  | 0.003717 | 0.002728 | 0.141491 | 0.024973 | 13.256818 | 0.286239 | 1497.429437 | 3.887690 | 58.394533 |  |
|  | 0.004861 | 0.003324 | 0.155772 | 0.019490 | 15.524566 | 0. 326156 | 1515.358205 | 4.211106 | 58.955866 |  |
|  | 0.005171 | 0.002391 | 0.138188 | 0. .019418 | 13.799562 | 0. 297933 | 1599.383380 | 4.249279 | 57.992041 |  |
|  | 0.003636 | 0.003289 | 0.132291 | 0.019429 | 14.499669 | 0. 320354 | 1593.81258 | 4.662545 | 59.302918 |  |
|  | 0.005403 | 0.002391 | 0.132030 | 0.019488 | 13.458846 | 0. 273931 | 1605.49502 | 4.305502 | 57.059681 |  |
| Average: | 0.0045576 | 0.0028246 | 0.1399544 | 0.0205596 | 14.1078902 | 0. 3008984 | 1562.2957244 | 4.263224 | 58.3410338 | 15996.697 |

Graph:

After graphing the points and using linear regression formulas, I was able to estimate the array to be around 15996.697 seconds, or 4.4435 hours. This is a huge jump from the lower values of n. We barely see a difference at n=100 and even at n = 1000, but as soon as we get to 10000 points, we really see just how efficient the minimum heap is with its functions. As soon as we reach 100000, the heap is up to 360 times faster than the unsorted array.

Because for the unsorted array, the DeleteMin() function has a complexity of O(V), we can see that this makes a huge difference, especially for a graph that only has three edges for each node. This sparse graph makes it hard for the unsorted array to detect minimum values, and it can’t do it as quickly as it could with more edges in each node. The MinHeap is a huge optimization strategy, as we see just how quickly it can outshine the unsorted array with bigger values of points. Although it is harder to implement conceptually, the optimization in the long run as n approaches infinity is a huge increase to efficiency for our algorithm. The simple change in complexity of the whole program from O(V^2 + E) to O((V+E)logV) is a huge change in the efficiency of the program.

Because MinHeap has at worst logV complexity, we see on the graph that logarithmic behavior. At lower values of n, we see that both the UnsortedArray and the MinHeap are fairly similar, but that as n approaches infinity, that the logarithmic nature of MinHeap displays rather accurately, increasing in time very slowly while the UnsortedArray shoots up from 15 seconds to around 25 minutes.