CS 312 Project 3 – Network Routing

Braden Webb

1. I implemented Dijkstra's algorithm in the computeShortestPaths() function of the NetworkRoutingSolver class, and it runs well. My documented code is provided at the end of this document.
2. I implemented the priority queue both as an array and as a binary heap with the requisite time complexities.
3. I discuss the time and space complexities of each part of the algorithm in the comments and docstrings of my code. Refer there for complexity discussion.
4. I've attached the screenshots of the three different random seeds immediately after my empirical analysis and before my code.
5. Empirical Analysis:

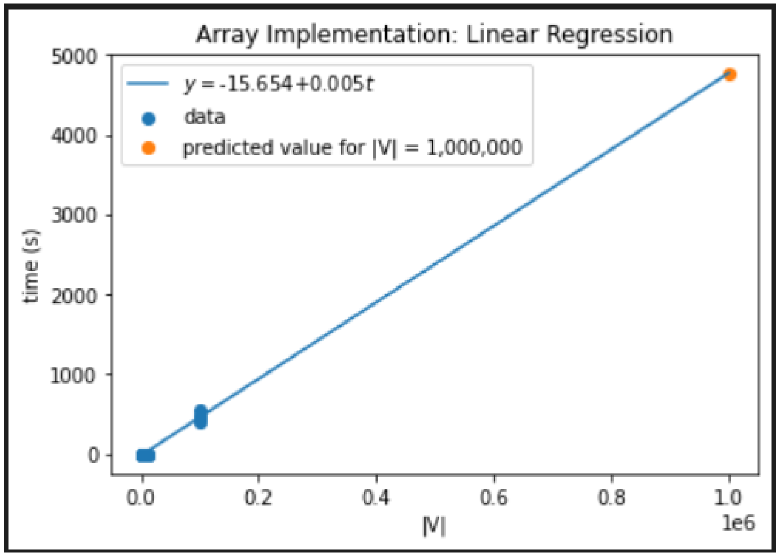
Raw data:

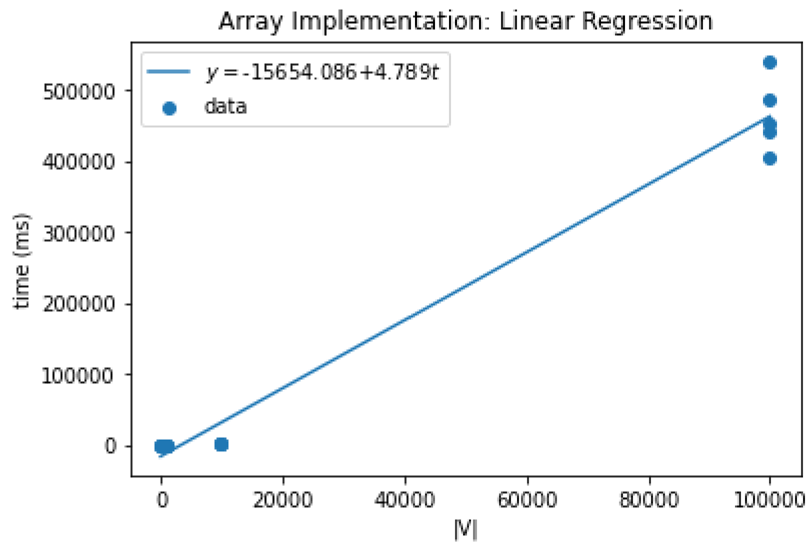
	Array Implementation			
n	**100**	**1000**	**10000**	**100000**
run 1	0.001	0.03	3.301	453.738
run 2	0	0.029	3.594	442.823
run 3	0.001	0.032	3.74	405.106
run 4	0	0.031	3.037	486.951
run 5	0.001	0.031	3.355	541.4
Average:	0.0006	0.0306	3.4054	466.0036

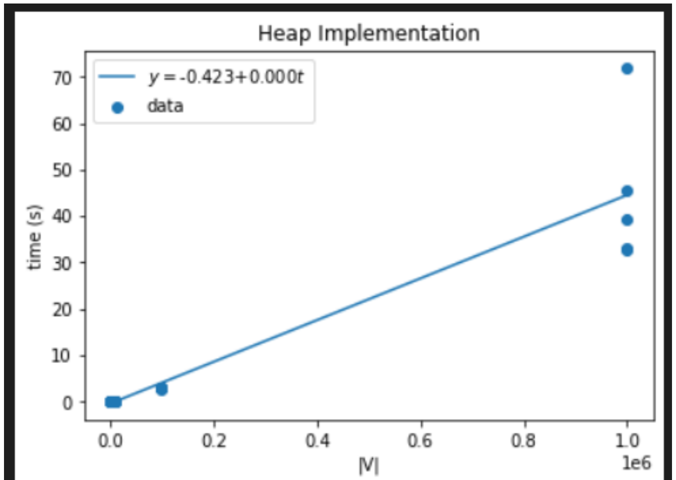
	Heap Implementation				
n	**100**	**1000**	**10000**	**100000**	**1000000**
run 1	0	0.004	0.17	2.81	45.574
run 2	0	0.009	0.182	3.108	72.019
run 3	0.016	0.018	0.144	3.08	39.436
run 4	0.001	0.002	0.2	2.98	33.247
run 5	0.001	0.009	0.161	3.098	32.865
Average:	0.0036	0.0084	0.1714	3.0152	44.6282

The linear regression model for the array implementation predicts that a 1,000,000-node graph would run in

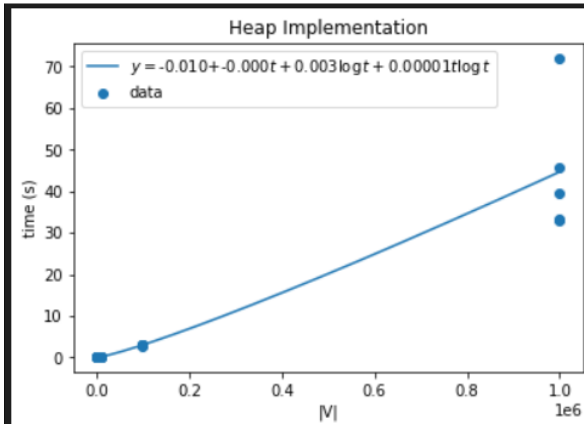
```
predicted time for 1,000,000 vertices using the array implmentation: 4773.334 s
```





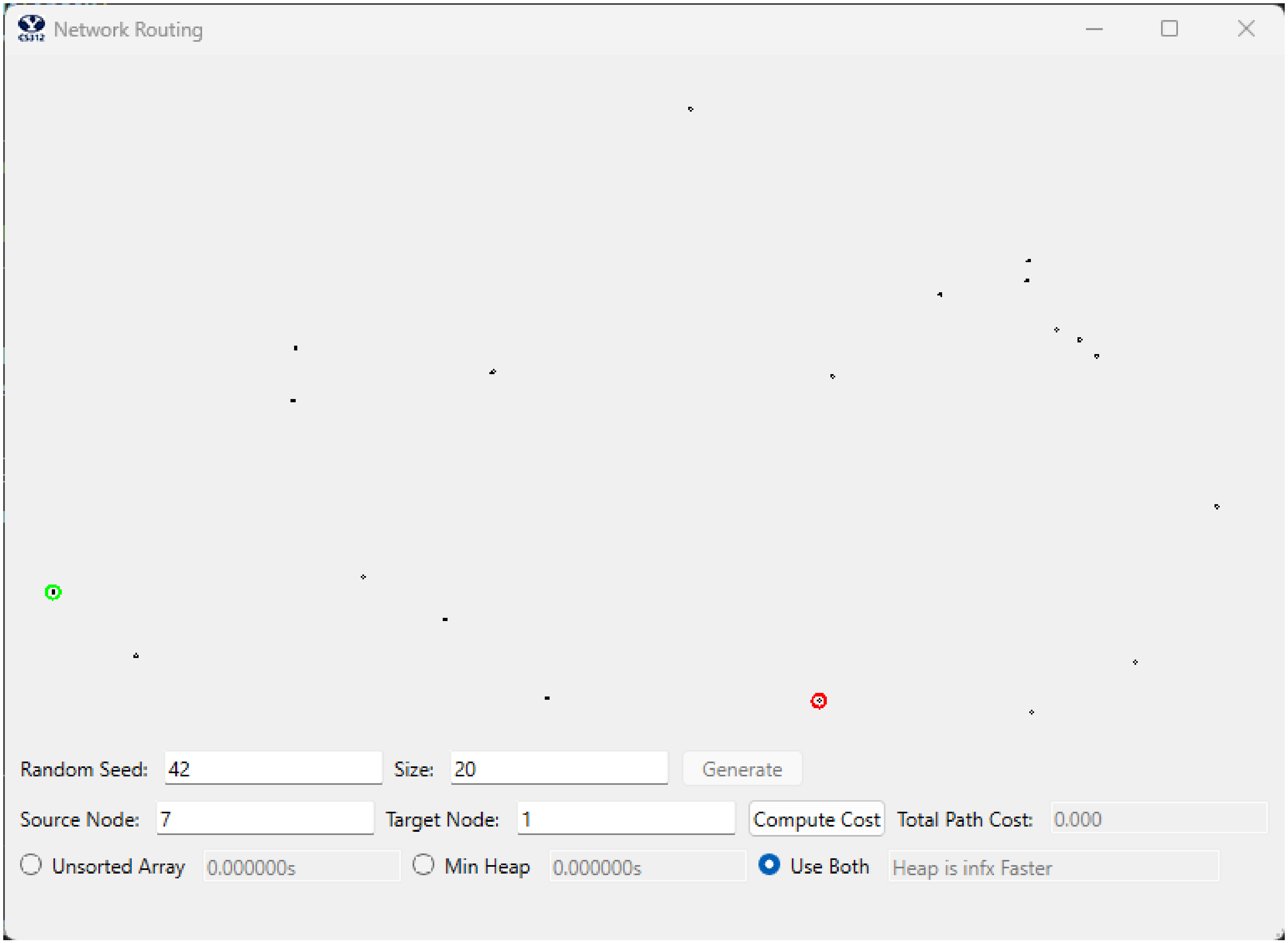


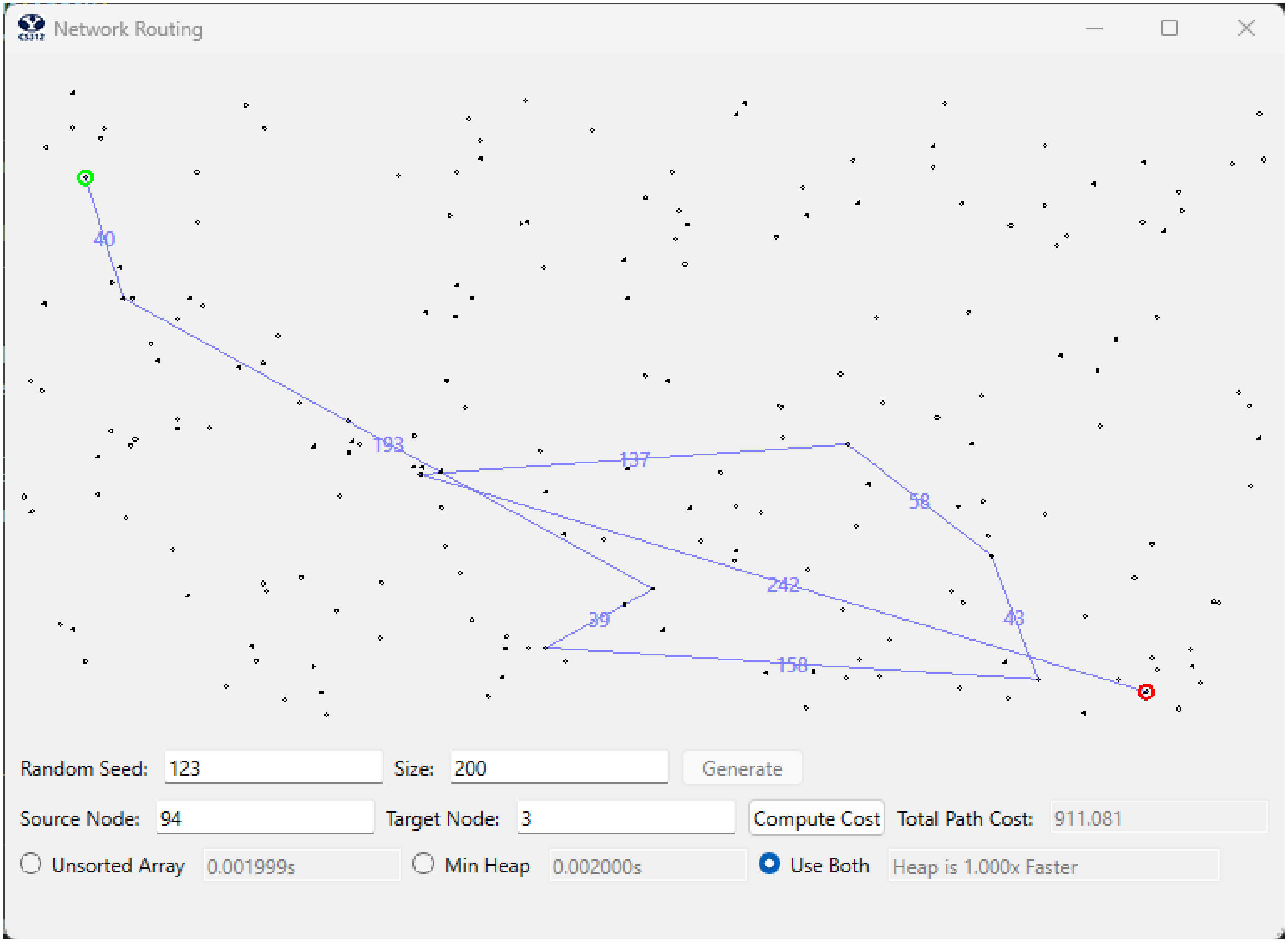

```
params: [-4.22810063e-01  4.49472148e-05]
sse: 1053.4570066139372
```

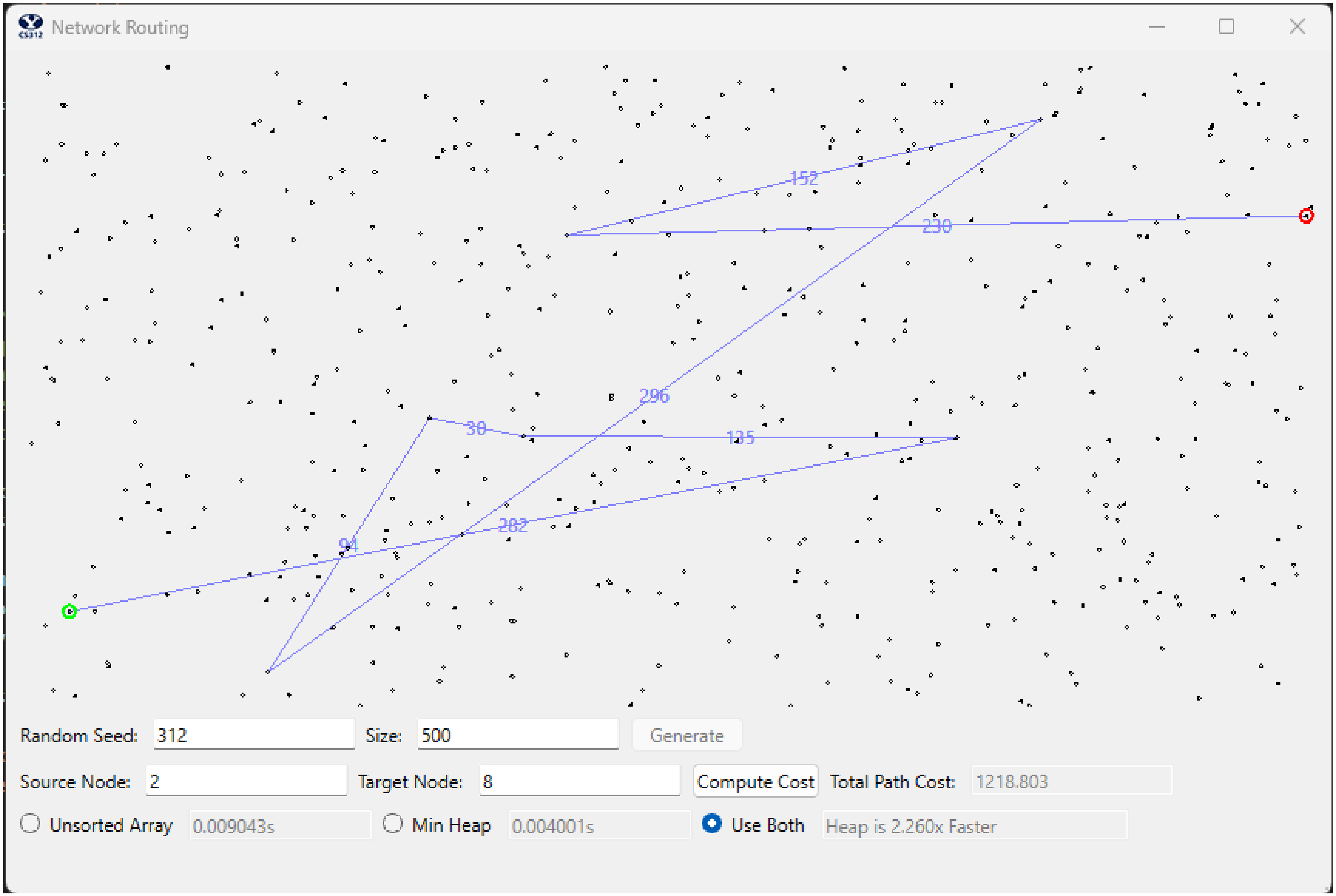



```
params: [-9.81174629e-03 -4.35076495e-05  2.95105781e-03  6.37724376e-06]
sse: 1046.0792013222763
```

The logarithmic basis functions seem to approximate the heap data better than the linear model.


Network Routing
Random Seed:
42
Size:
20
Generate
Source Node:
7
Target Node:
1
Compute Cost
Total Path Cost:
0.000
Unsorted Array
0.000000s
Min Heap
0.000000s
Use Both
Heap is infx Faster


Network Routing
40
193
137
58
242
39
43
158
Random Seed: 123
Size: 200
Generate
Source Node: 94
Target Node: 3
Compute Cost
Total Path Cost: 911.081
Unsorted Array 0.001999s
Min Heap 0.002000s
Use Both
Heap is 1.000x 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
0.009043s
Min Heap
0.004001s
Use Both
Heap is 2.260x Faster

```python
#!/usr/bin/python3


from CS312Graph import *
import time
import math

class PQInterface:
    def __init__(self):
        self.queue = None

    def insert(self, item):
        pass

    def decrease_key(self, node, new_key):
        pass

    def delete_min(self):
        pass

    def make_queue(self, iterable) -> "PQInterface":
        pass

    def __len__(self):
        return len(self.queue)

class PQ_Heap(PQInterface):
    def __init__(self):
        self.queue = []
        self.node_to_index = {}

    def insert(self, item):
        """
```

```
        Time Complexity: O(log(|V|)) since that is the complexity of bubbleUp().
        Space Complexity: O(1) since we are not allocating any new memory.
        """
        self.queue.append(item)
        child_index = len(self.queue) - 1
        self.node_to_index[item[2]] = child_index
        self._bubbleUp(child_index)

    def decrease_key(self, node, new_key):
        """
        Time Complexity: O(log(|V|)) since that is the worst-case complexity of bubbleUp().
        Space Complexity: O(1) since we are not allocating any new memory.
        """
        index = self.node_to_index[node]
        old_key, i, node = self.queue[index]
        self.queue[index] = (new_key, i, node)
        self._bubbleUp(index)

    def delete_min(self):
        """
        Time Complexity: O(log(|V|)) since that is the worst-case complexity of bubbleDown().
        Space Complexity: O(1) since we are not allocating any new memory.
        """
        min = self.queue[0]
        if len(self.queue) == 1:
            self.queue.pop()
            return min
        self.queue[0] = self.queue.pop()
        self.node_to_index[self.queue[0][2]] = 0
        self._bubbleDown(0)
        return min
```

```python
    def make_queue(self, iterable):
        """
        Time Complexity: In general, O(|V|log(|V|)) since we are inserting each element into the heap.
        However, in the case of Dijkstra's algorithm, we initialize almost every node with a key of infinity.
        This means that insert only needs to call bubble_up() once to check that the heap property is still
satisfied,
        reducing the time complexity to O(|V|).
        Space Complexity: O(|V|) since we are allocating a new array of size |V|.
        """
        for triple in iterable:
            self.insert(triple)

    def _bubbleUp(self, index):
        """
        Time Complexity: O(log(|V|)) since the maximum depth of the heap is log(|V|).
        Space Complexity: O(1) since we are not allocating any new memory.
        """
        if index == 0:
            return
        parent_index = (index - 1) // 2
        if self.queue[parent_index][0] > self.queue[index][0]:
            self._swap(parent_index, index)
            self._bubbleUp(parent_index)

    def _bubbleDown(self, index):
        """
        Time Complexity: O(log(|V|)) since the maximum depth of the heap is log(|V|).
        Space Complexity: O(1) since we are not allocating any new memory.
        """
        left_ind = index * 2 + 1
        right_ind = index * 2 + 2
        if left_ind >= len(self.queue):
```

```python
            return
        if right_ind >= len(self.queue):
            if self.queue[index] > self.queue[left_ind]:
                self._swap(index, left_ind)
            return
        if self.queue[left_ind] < self.queue[right_ind]:
            if self.queue[index] > self.queue[left_ind]:
                self._swap(index, left_ind)
                self._bubbleDown(left_ind)
        else:
            if self.queue[index] > self.queue[right_ind]:
                self._swap(index, right_ind)
                self._bubbleDown(right_ind)

    def _swap(self, index1, index2):
        """
        Time Complexity: O(1) since we are just swapping two elements in the array.
        Space Complexity: O(1) since we are not allocating any new memory.
        """
        self.queue[index1], self.queue[index2] = self.queue[index2], self.queue[index1]
        self.node_to_index[self.queue[index1][2]] = index1
        self.node_to_index[self.queue[index2][2]] = index2

class PQ_Array(PQInterface):
    def __init__(self):
        self.queue = []
        self.node_to_index = {}

    def insert(self, item):
        """
        Time Complexity: O(1) since we are just appending to the end of the array.
        Space Complexity: O(1) since we are only adding one element to the array.
```

```python
        This method is used to insert a new item into the priority queue.
        """
        self.queue.append(item)
        self.node_to_index[item[2]] = len(self.queue) - 1

    def decrease_key(self, node, new_key):
        """
        Time Complexity: O(1) since we are just updating the key of a node in the priority queue.
        Space Complexity: O(1) since we are not allocating any new memory.

        This method is used to decrease the key of a node in the priority queue.
        """
        index = self.node_to_index[node]
        old_key, i, node = self.queue[index]
        self.queue[index] = (new_key, i, node)

    def delete_min(self):
        """
        Time Complexity: O(|V|), since finding the minimum is O(|V|) and swapping the minimum with the last
element is O(1)
        Space Complexity: O(1), since we do not need to allocate any new memory

        This method is used to delete the minimum item from the priority queue.
        """
        min_index = self.node_to_index[min(self.queue, key=lambda x: x[0])[2]]
        if min_index == len(self.queue) - 1:
            return self.queue.pop()
        ret_value = self.queue[min_index]
        self.queue[min_index] = self.queue.pop()
        self.node_to_index[self.queue[min_index][2]] = min_index
        return ret_value
```

```python
    def make_queue(self, iterable):
        """
        Time Complexity: O(|V|)
        Space Complexity: O(|V|)

        This method is used to initialize the priority queue with a list of tuples.
        """
        for triple in iterable:
            self.insert(triple)


class NetworkRoutingSolver:
    def __init__( self):
        pass

    def initializeNetwork( self, network ):
        assert( type(network) == CS312Graph )
        self.network = network
        self.dist = {node : math.inf for node in network.nodes}
        self.prev = {node: None for node in network.nodes}

    def getShortestPath( self, destIndex ):
        path_edges = []
        total_length = 0
        cur_node = self.network.nodes[destIndex]
        while self.prev[cur_node] is not None:
            parent_node = self.prev[cur_node]
            # print(parent_node.neighbors)
            edge = next((edge for edge in parent_node.neighbors if edge.dest == cur_node), None)
            # print(edge)
            if edge is None:
                print('ERROR: could not find edge between {} and {}'.format(parent_node, cur_node))
```

```
            return

        path_edges.append( (edge.src.loc, edge.dest.loc, '{:.0f}'.format(edge.length)) )
        total_length += edge.length
        cur_node = parent_node
    return {'cost':total_length, 'path':path_edges}

def computeShortestPaths( self, srcIndex, use_heap=False ):
    """
    The space complexity of this method depends on that of the graph.  If the graph is represented
    as an adjacency list, then the space complexity is O(|V| + |E|).  If the graph is represented
    as an adjacency matrix, then the space complexity is O(|V|^2).

    The time complexity of this method depends on the data structure used to implement the priority
    queue. That complexity is |V| x delete_min + (|V| + |E|) x decrease_key, where |V| is the number of
nodes
    in the graph, |E| is the number of edges in the graph, and decrease_key and delete_min are the time
    complexities of the decrease_key and delete_min operations, respectively. Thus, if the priority queue
    is implemented as an array, the time complexity is O(|V|^2).  If the priority queue is
    implemented as a heap, the time complexity is O((|V| + |E|)log|V|).
    """

    self.source = self.network.nodes[srcIndex]
    t1 = time.time()

    # Run Dijkstra's algorithm
    self.dist[self.source] = 0
    if use_heap:
        H = PQ_Heap()
    else:
        H = PQ_Array()
```

```
        # We need to use a list of tuples because the priority queue needs to be able to
        # update the priority of a node.  The priority queue will use the first element
        # of the tuple as the priority, and the second element as a tie-breaker. Since
        # the second elements of the tuples are unique, the priority queue will not
        # never need to compare the actual nodes themselves.
        H.make_queue([(self.dist[node], i, node) for i, node in enumerate(self.network.nodes)])
        while len(H) > 0: # O(|V|) time
            u = H.delete_min()[2] # time complexity of delete_min depends on the data structure used
            for edge in u.neighbors: # O(|E|) time
                v = edge.dest
                if self.dist[v] > self.dist[u] + edge.length:
                    self.dist[v] = self.dist[u] + edge.length
                    self.prev[v] = u
                    H.decrease_key(v, self.dist[v]) # time complexity of decrease_key depends on the data
structure used

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