Project 3 Report
Joseph Koetting
Section 2

1. Copy of Code (DIJKSTRAS)

****

```
#!/usr/bin/python3


from CS312Graph import *
from PriorityQueueArray import *
from PriorityQueueHeap import *
import time


class Distance:
    def __init__(self):
        self.fromNode = None
        self.distance = float('inf')


class NetworkRoutingSolver:
    def __init__( self, display ):
        self.distances = None

    def initializeNetwork( self, network ):
        assert( type(network) == CS312Graph )
        self.network = network


    ##########################################
    # def getShortestPath
    # Returns the shortest path from source node to destination
    #
    # Time Complexity: O(n) Path could be across all nodes in the graph
    # Space Complexity: O(n) If the Path could be across all nodes in the graph,
    # I would have to store edges from each node
    ##########################################
    def getShortestPath(self, destIndex ):

        # init variables
        path_edges = []

        # finite variable
        total_distance = self.distances[destIndex].distance

        # Temporary variables for loop
        tmp_distance = self.distances[destIndex]
        edge_distance = total_distance
        tmp_location = self.network.nodes[destIndex].loc

        while tmp_distance.fromNode is not None:

            # get the previous node
            edge = self.network.nodes[tmp_distance.fromNode]

            # get distance info for the node
```

```
            tmp_distance = self.distances[tmp_distance.fromNode]

            # append new edge to array with info
            path_edges.append((edge.loc, tmp_location, '{:.0f}'.format(edge_distance -
    tmp_distance.distance)))

            # prepare distance for next distance calculation
            edge_distance = edge_distance - (edge_distance - tmp_distance.distance)

            # retain the location for next calculation
            tmp_location = edge.loc

        return {'cost': total_distance, 'path': path_edges}

    def computeShortestPaths( self, srcIndex, use_heap=False):
        self.source = srcIndex

        t1 = time.time()

        self.dijkstras(srcIndex, use_heap)

        t2 = time.time()

        return t2-t1


    ##########################################
    # def dijkstras
    # Performs BFS search on the graph
    #
    # DIJKSRAS TOTAL WITH ARRAY
    # Time Complexity: O(n^2) loops through each vertices then, for each
    # vertices loop through all priorities
    # Space Complexity: O(n) stores all the priorities and distance
    #
    # DIJKSRAS TOTAL WITH HEAP
    # Time Complexity: O(n * log(n)) Loops through each vertices then,
    # for each vertices traverse through a binary tree
    # Space Complexity: O(n) stores all the priorities and distances
    #
    # DIJKSRAS BY ITSELF
    # Time Complexity: O(n) loops through entire array of vertices
    # Space Complexity: 0(n) stores all the distances
    #
    # ARRAY BY ITSELF
    # Time Complexity Array: O(n) most complex method loops through an array
    # Space Complexity Array: O(n) stores all the priorities
    #
    # HEAP BY ITSELF
    # Time Complexity Heap: O(log(n)) most complex method traverses through a binary tree
    # Space Complexity Heap: O(n) stores all the priorities
    ##########################################
    def dijkstras(self, srcIndex, use_heap):

        # init variables
        nodes = self.network.nodes
        distances = []

        # Set Distance for all nodes to Infinity
        for i in range(0, len(nodes)):
```

```
            distances.append(Distance())

        # Set Initial Node Distance to Zero
        distances[srcIndex].distance = 0

        # make queue
        if use_heap:
            priorityQueue = PriorityQueueHeap(srcIndex)
        else:
            priorityQueue = PriorityQueueArray(len(nodes), srcIndex)

        min_index = priorityQueue.delete_min()

        while min_index is not None:

            # get min node and delete
            node = nodes[min_index]

            for i in range(0, len(node.neighbors)):
                if distances[node.neighbors[i].dest.node_id].distance > \
                        distances[node.node_id].distance + node.neighbors[i].length:

                    distances[node.neighbors[i].dest.node_id].distance = \
                        distances[node.node_id].distance + node.neighbors[i].length
                    distances[node.neighbors[i].dest.node_id].fromNode = node.node_id
                    priorityQueue.decrease_key(node.neighbors[i].dest.node_id,
                                               distances[node.node_id].distance + node.neighbors[i].length)

            min_index = priorityQueue.delete_min()

        self.distances = distances
```

****

2. Copy of Code (PRIORITY QUEUE USING ARRAY AND HEAP)

****

```
#########################################
# class Priority Queue Array
# Inserts Item into Priority Queue
#
# Time Complexity Array: O(n) most complex method loops through an array
# Space Complexity Array: O(n) stores all the priorities
#########################################



class PriorityQueueArray:

    #########################################
    # def make queue
    # Init a list of priorities
    #
    # Time Complexity: O(n)  Creates a list with n elements
    # Space Complexity: O(n) The list has to have n elements in it
    #########################################
    def __init__(self, n, srcIndex):
        self.priorities = []

        for i in range(0, n):
```

```
            self.priorities.append(float('inf'))

        self.decrease_key(srcIndex, 0)

    ##########################################
    # def insert
    # Does Nothing
    #
    # Time Complexity: Not Really Applicable
    # Space Complexity: Not Really Applicable
    ##########################################
    @staticmethod
    def insert():
        pass

    ##########################################
    # def decrease_key
    # Decreases the Key
    #
    # Time Complexity: 0(1)
    # Space Complexity: Not Really Applicable
    ##########################################
    def decrease_key(self, index, priority):
        self.priorities[index] = priority

    ##########################################
    # def delete_min
    # Finds Deletes and returns Node with smallest edge
    #
    # Time Complexity: O(n)  Loops through the array
    # Space Complexity: O(1) stores some temporary variables
    ##########################################
    def delete_min(self):
        smallest_index = None
        smallest = None

        # loop through priorities and get the index of the smallest priority
        for i in range(0, len(self.priorities)):
            if self.priorities[i] is not None and \
                    (smallest is None or smallest > self.priorities[i]):
                smallest = self.priorities[i]
                smallest_index = i

        # if no smallest priority exists return None
        if smallest_index is None:
            return None

        # return smallest priority and mark it as visited
        self.priorities[smallest_index] = None
        return smallest_index
```

****

```
##########################################
# class Priority Queue Heap
# Inserts Item into Priority Queue
#
# Time Complexity Heap: O(log(n)) most complex method traverses through a binary tree
# Space Complexity Heap: O(n) stores all the priorities
##########################################
```

```python
class Element:
    def __init__(self, index, priority):
        self.index = index
        self.priority = priority


class PriorityQueueHeap:

    ##########################################
    # def make queue
    # init variables
    #
    # Time Complexity: Not Really Applicable
    # Space Complexity: O(1) stores some variables
    ##########################################
    def __init__(self, srcIndex):
        self.priorities = []
        self.size = 0
        self.dictionary = {}
        self.insert(srcIndex, 0)

    ##########################################
    # def insert
    # Inserts Item into Priority Queue
    #
    # Time Complexity: O(log(n)) calls bubble_up which is O(log(n))
    # However the insert method does not add on significant complexity
    # Space Complexity: O(1) Creates temp variables and stores things.
    # also appends an item to the heap
    ##########################################
    def insert(self, index, priority):

        # if index already exists in tree
        if index in self.dictionary:
            # if visited return
            if self.dictionary[index] is None:
                return
            # not visited bubble_up
            else:
                self.bubble_up(self.dictionary[index])
                return

        # create new node in tree
        self.priorities.append(Element(index, priority))
        self.size = self.size + 1
        self.dictionary[index] = self.size

        # reorder tree
        self.bubble_up(len(self.priorities))

    ##########################################
    # def bubble_up
    # Inserts Item into Priority Queue
    #
    # Time Complexity: O(log(n))  recursively traverses a binary tree
    # for a worst case of to a leaf node to a root node
    # Space Complexity: O(1) Creates some temporary variables
    ##########################################
```

```
def bubble_up(self, child_loc):

    # if the child_loc is not the root
    if child_loc > 1:

        # get the parent_loc
        if child_loc % 2 == 0:
            parent_loc = child_loc // 2
        else:
            parent_loc = (child_loc - 1) // 2

        # get priorities
        child_priority = self.priorities[child_loc - 1].priority
        parent_priority = self.priorities[parent_loc - 1].priority

        # if child priority is less than parents
        if child_priority < parent_priority:

            # swap everything
            child_index = self.priorities[child_loc - 1].index
            parent_index = self.priorities[parent_loc - 1].index

            self.dictionary[child_index] = parent_loc
            self.dictionary[parent_index] = child_loc

            self.priorities[child_loc - 1], self.priorities[parent_loc - 1] = \
                self.priorities[parent_loc - 1],  self.priorities[child_loc - 1]

            # keep recursing until we get to the root
            self.bubble_up(parent_loc)
        else:
            # tree is good
            return

#########################################
# def decrease_key
# Decreases the Key
#
# Time Complexity:  O(log(n)) calls insert, which calls bubble_up
# which is O(log(n))
# Space Complexity: 0(1) All functions this calls has 0(1) space complexity
#########################################
def decrease_key(self, index, priority):
    self.insert(index, priority)

#########################################
# def delete_min
# Finds Deletes and returns Node with smallest edge
#
# Time Complexity: O(log(n)) calls bubble_down which is O(log(n))
# Space Complexity: O(1) stores some variables
#########################################
def delete_min(self):

    if len(self.priorities) == 0:
        return None

    # get min value
    min = self.priorities[0].index
    self.dictionary[min] = None
```

```
        # pop last element of array
        tmp = self.priorities.pop()
        self.size = self.size - 1

        # reset location of last node in tree to first
        if self.size != 0:
            self.dictionary[tmp.index] = 1
            self.priorities[0] = tmp
            self.bubble_down(1)

        return min

    #########################################
    # def bubble_down
    # Inserts Item into Priority Queue
    #
    # Time Complexity: O(log(n)) recursively traverses a binary tree
    # for a worst case of to a root node to a leaf node
    # Space Complexity: O(1) O(n) creates some variables
    #########################################
    def bubble_down(self, parent_loc):

        # if child is left node
        if parent_loc * 2 == self.size:
            child_loc = parent_loc * 2

        # if there are two children
        elif parent_loc * 2 + 1 <= self.size:

            left_child_loc = parent_loc * 2
            right_child_loc = parent_loc * 2 + 1

            left_child_priority = self.priorities[left_child_loc - 1].priority
            right_child_priority = self.priorities[right_child_loc - 1].priority

            # Keep child with smallest priority
            if left_child_priority < right_child_priority:
                child_loc = left_child_loc
            else:
                child_loc = right_child_loc

        # no children
        else:
            return

        # get priorities
        child_priority = self.priorities[child_loc - 1].priority
        parent_priority = self.priorities[parent_loc - 1].priority

        # if child priority is less than parent
        if child_priority < parent_priority:

            # get the indexes
            child_index = self.priorities[child_loc - 1].index
            parent_index = self.priorities[parent_loc - 1].index

            # swap everything
            self.dictionary[child_index] = parent_loc
            self.dictionary[parent_index] = child_loc
```

```
        self.priorities[child_loc - 1], self.priorities[parent_loc - 1] = \
            self.priorities[parent_loc - 1], self.priorities[child_loc - 1]

        # recurse down the tree until tree is sorted
        self.bubble_down(child_loc)

    return
```

****

3. Complexity : Every Method Explained

****

```
#########################################
# def getShortestPath
# Returns the shortest path from source node to destination
#
# Time Complexity: O(n) Path could be across all nodes in the graph
# Space Complexity: O(n) If the Path could be across all nodes in the graph,
# I would have to store edges from each node
#########################################

#########################################
# def dijkstras
# Performs BFS search on the graph
#
# DIJKSRAS TOTAL WITH ARRAY
# Time Complexity: O(n^2) loops through each vertices then, for each
# vertices loop through all priorities
# Space Complexity: O(n) stores all the priorities and distance
#
# DIJKSRAS TOTAL WITH HEAP
# Time Complexity: O(n * log(n)) Loops through each vertices then,
# for each vertices traverse through a binary tree
# Space Complexity: O(n) stores all the priorities and distances
#
# DIJKSRAS BY ITSELF
# Time Complexity: O(n) loops through entire array of vertices
# Space Complexity: 0(n) stores all the distances
#
# ARRAY BY ITSELF
# Time Complexity Array: O(n) most complex method loops through an array
# Space Complexity Array: O(n) stores all the priorities
#
# HEAP BY ITSELF
# Time Complexity Heap: O(log(n)) most complex method traverses through a binary tree
# Space Complexity Heap: O(n) stores all the priorities
#########################################
```

****

```
#########################################
# class Priority Queue Array
# Inserts Item into Priority Queue
#
# Time Complexity Array: O(n) most complex method loops through an array
# Space Complexity Array: O(n) stores all the priorities
```

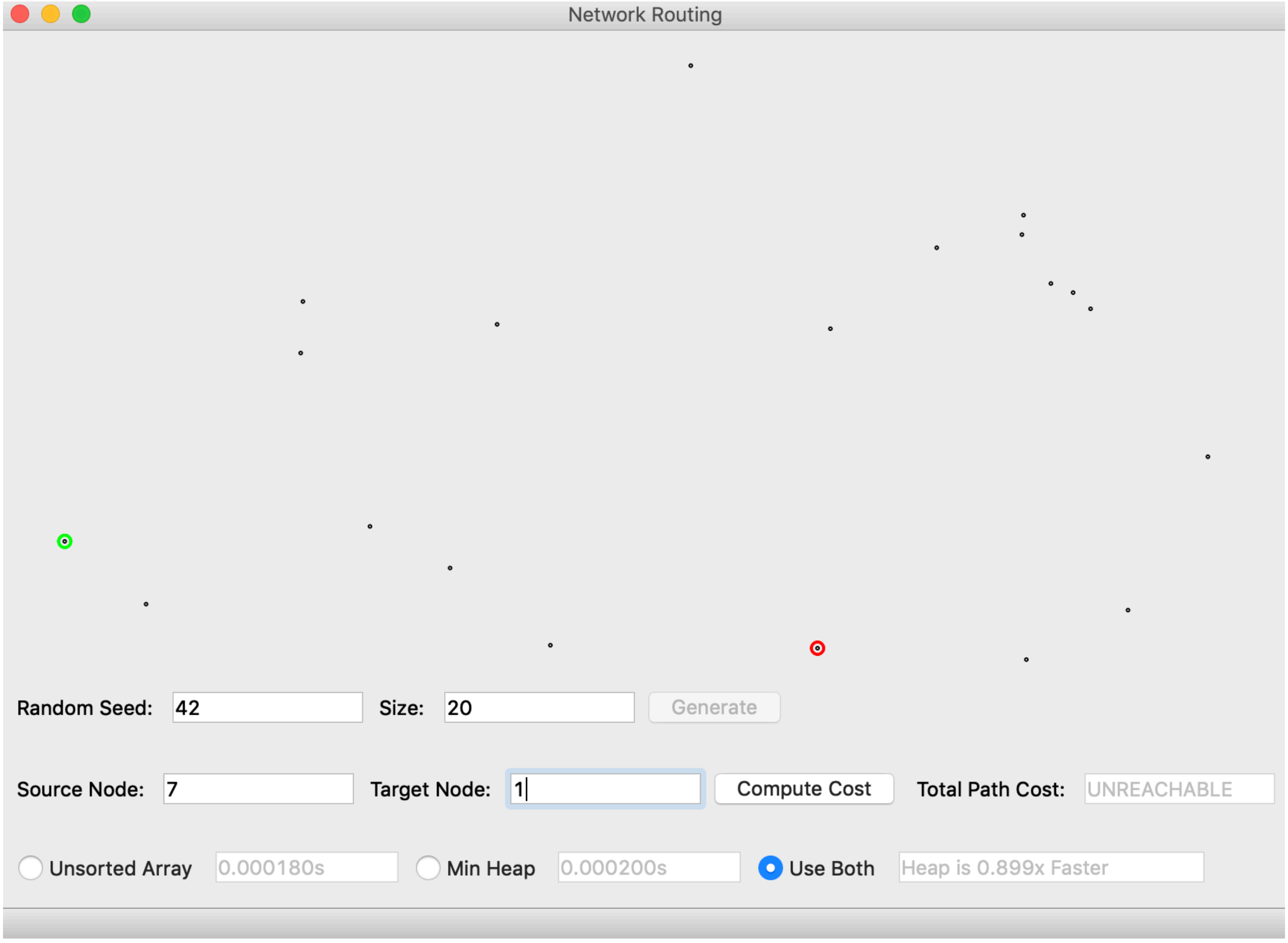
```
##########################################

##########################################
# def make queue
# Init a list of priorities
#
# Time Complexity: O(n)  Creates a list with n elements
# Space Complexity: O(n) The list has to have n elements in it
##########################################

##########################################
# def insert
# Does Nothing
#
# Time Complexity: Not Really Applicable
# Space Complexity: Not Really Applicable
##########################################

##########################################
# def decrease_key
# Decreases the Key
#
# Time Complexity: 0(1)
# Space Complexity: Not Really Applicable
##########################################

##########################################
# def delete_min
# Finds Deletes and returns Node with smallest edge
#
# Time Complexity: O(n)  Loops through the array
# Space Complexity: O(1) stores some temporary variables
##########################################

**********************************************************************************************************************************

##########################################
# class Priority Heap
# Inserts Item into Priority Queue
#
# Time Complexity Heap: O(log(n)) most complex method traverses through a binary tree
# Space Complexity Heap: O(n) stores all the priorities
##########################################

##########################################
# def make queue
# init variables
#
# Time Complexity: Not Really Applicable
# Space Complexity: O(1) stores some variables
##########################################

##########################################
# def insert
# Inserts Item into Priority Queue
#
# Time Complexity: O(log(n)) calls bubble_up which is O(log(n))
# However the insert method does not add on significant complexity
# Space Complexity: O(1) Creates temp variables and stores things.
# also appends an item to the heap
```

```
##########################################

##########################################
# def bubble_up
# Inserts Item into Priority Queue
#
# Time Complexity: O(log(n))  recursively traverses a binary tree
# for a worst case of to a leaf node to a root node
# Space Complexity: O(1) Creates some temporary variables
##########################################

##########################################
# def decrease_key
# Decreases the Key
#
# Time Complexity:  O(log(n)) calls insert, which calls bubble_up
# which is O(log(n))
# Space Complexity: 0(1) All functions this calls has 0(1) space complexity
##########################################

##########################################
# def delete_min
# Finds Deletes and returns Node with smallest edge
#
# Time Complexity: O(log(n)) calls bubble_down which is O(log(n))
# Space Complexity: O(1) stores some variables
##########################################

##########################################
# def bubble_down
# Inserts Item into Priority Queue
#
# Time Complexity: O(log(n)) recursively traverses a binary tree
# for a worst case of to a root node to a leaf node
# Space Complexity: O(1) O(n) creates some variables
##########################################
```

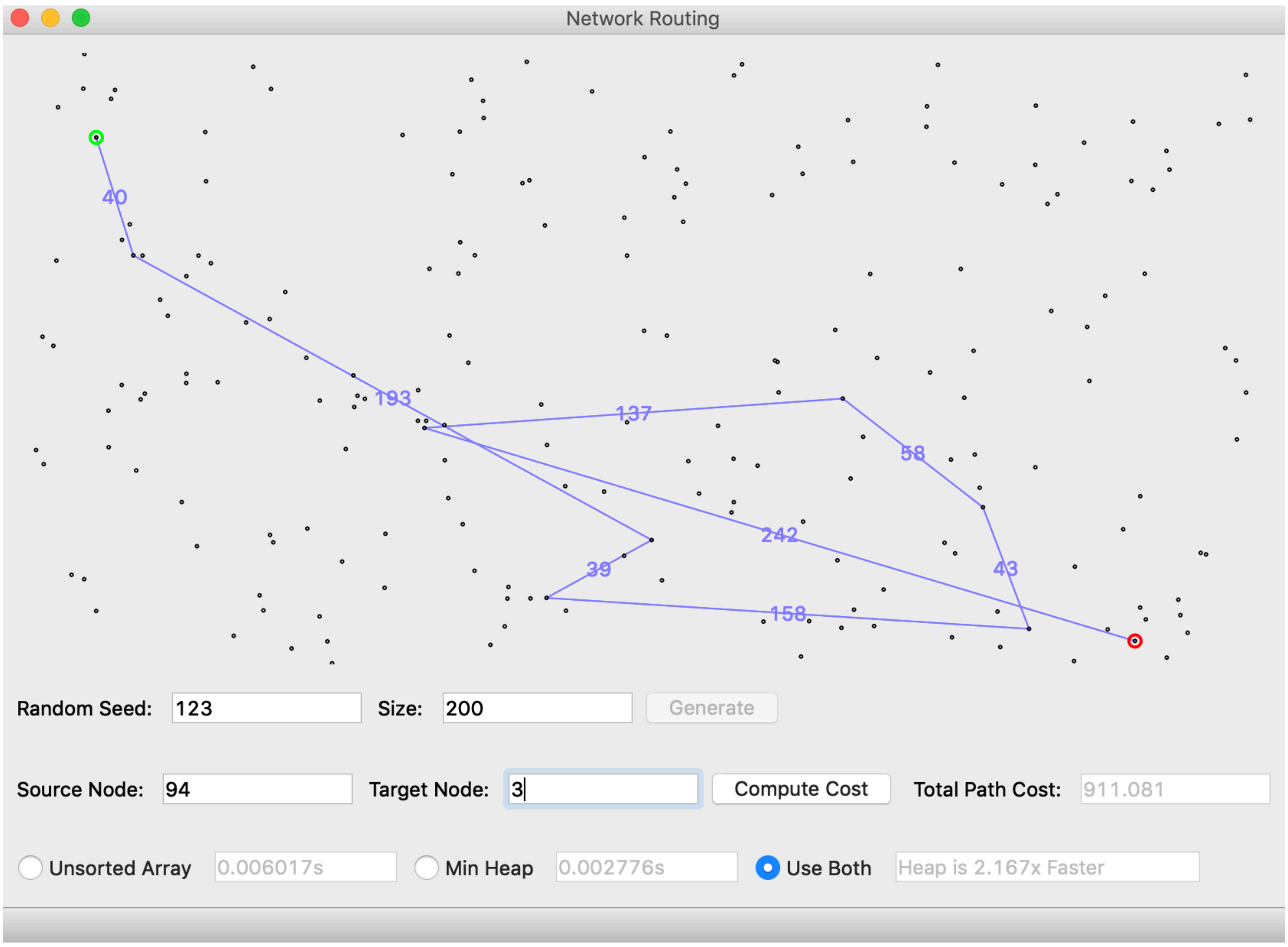
****

4. ScreenShots

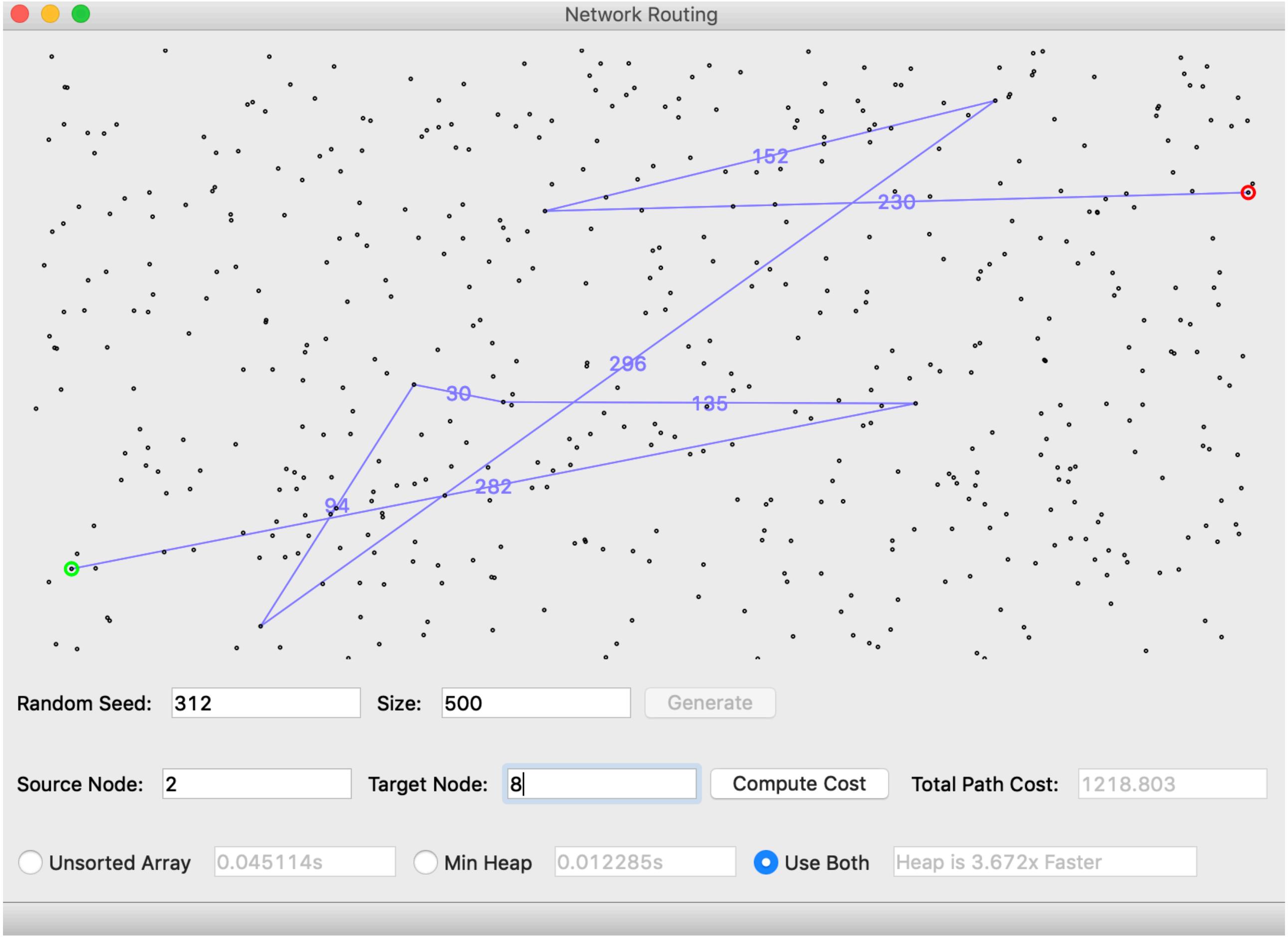
A.)


Network Routing
Random Seed: 42
Size: 20
Generate
Source Node: 7
Target Node: 1
Compute Cost
Total Path Cost: UNREACHABLE
Unsorted Array 0.000180s
Min Heap 0.000200s
Use Both
Heap is 0.899x Faster

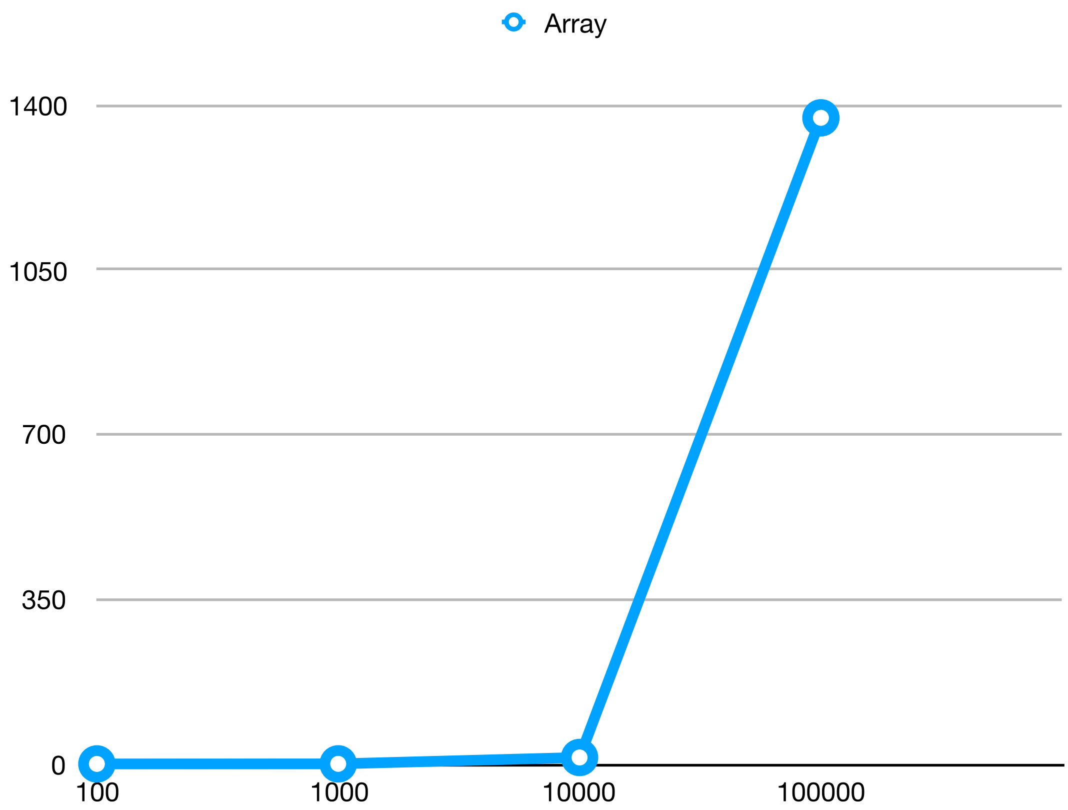
B.)


Network Routing
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 0.006017s
Min Heap 0.002776s
Use Both
Heap is 2.167x Faster

C.)

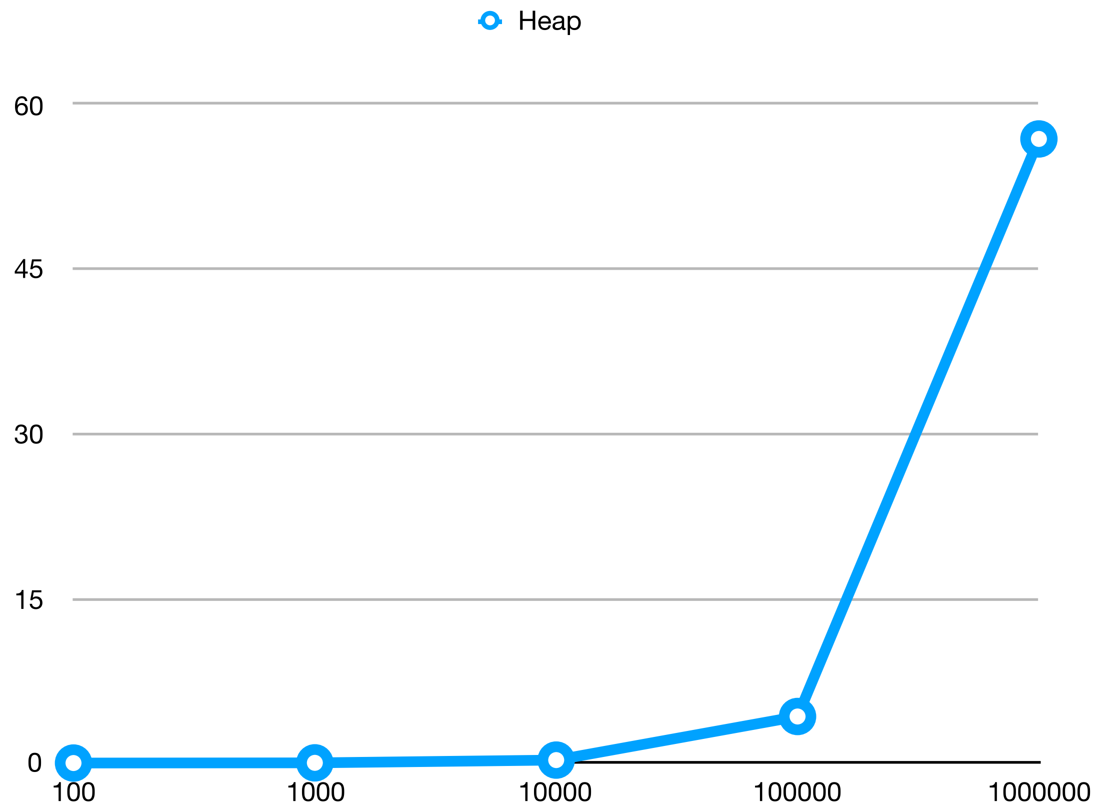

Network Routing
152
230
296
30
135
282
94
Random Seed: 312
Size: 500
Generate
Source Node: 2
Target Node: 8
Compute Cost
Total Path Cost: 1218.803
Unsorted Array 0.045114s
Min Heap 0.012285s
Use Both
Heap is 3.672x Faster

5.) Analysis



Array	Trial 1	Trial 2	Trial 3	Trial 4	Trial 5	Average
100	0.0019	0.0017	0.0017	0.0018	0.0017	0.0018
1000	0.16	0.14	0.13	0.13	0.14	0.14
10000	13.6	13.2	14.1	14.0	13.8	13.7
100000	1370	1345	1390	1388	1376	1373
1000000					Estimate:	1369000

* I didn't plot 1,000,000 because it is such a big outlier



Heap	Trial 1	Trial 2	Trial 3	Trial 4	Trial 5	Average
100	0.0013	0.0012	0.0012	0.0012	0.0013	0.0012
1000	0.019	0.022	0.018	0.019	0.020	0.019
10000	0.29	0.27	0.28	0.27	0.28	0.27
100000	4.36	4.24	4.13	4.20	4.28	4.24
1000000	59.42	56.11	55.34	56.34	55.84	56.84

Further Analysis:

Array Estimate 1000000 : 1369000

The Heap structure performed significantly better than the Array. While the Heap's runtime was scalable across 100 to 1000000, the Array was not scalable at all. The Array's final iteration it jumped from 1373 seconds to 1369000 seconds which is a lot more time consuming. This is because a n * Log(n) algorithm increases dramatically less than of a n^2 algorithm.

Also the heap structure would perform closer to an array if the graph had more edges (closer to n^2) but because each node had only three edges, it performed very well (closer to n * log(n))

When n = 100, each implementation performed around the same because both complexities operate similar with fewer vertices or n values.