Gregory Knapp

CS 312

Dr. Grimsman

10/19/2021

Project 3 Network Routing Solver

# Time Complexity Analysis of Project 3 – Breakdown by Implementation

The time complexity for each function found in NetworkRoutingSolver.py of the Project 3 code is as follows and pulled from the comments for each function in the code.

Time Complexity for Array Queue Implementations:

Delete_min – lines 31 – 36

```
# delete_min with unsorted array. With the unsorted array implementation, delete_min iterates through every
# node in the array to find out which has the shortest distance in the dist array, saves the index, and
# removes and returns the node that was popped from the queue. Due to the linear nature of arrays, there is
# not a better way to find the min without checking every value, so the implementation of this function
# is time complexity O(n)
```

Insert – lines 59 – 61

```
# Insert with unsorted array - This function simply uses the Python list.append() function to add the next node
# to the array. Because append adds the node to the end of the array, and the array is unsorted meaning that no
# work post appending needs to be done, the time complexity of insert() is O(1) time.
```

Decrease_key – lines 66 - 68

```
# decrease_key - Decrease key does not do anything in the array implementation of the priority queue, so there
# is no time complexity to report for this function. It is called in the algorithm no matter which queue type is
# used, but is simply passed when the array implementation is in use.
```

Make_queue – lines 72 – 75

```
# Make queue - The initial make queue function takes all of the nodes in the network and adds them to the
# unsorted array using the insert() function. Insert runs at O(1) time, but because every node must be added to the
# array through a for loop, we call insert() n times where n = number of nodes in network. This makes the time
# complexity of make_queue = O(n)
```

Size – lines 81-82

```
# A helper function to get the size of the array currently. Len(object) in python returns the integer of the
# size of the list with O(1) time, so the time complexity of this is O(1)
```

Time Complexity for Heap Queue Implementations

Delete_min – lines 94 – 100

```
# delete_min function using binary heap implementation. This is is one of the two core functions of the binary
# heap implementation. Because the node with the shortest distance is bubbled up to the top with decrease_key,
# finding the minimum value is very fast, O(1) since we know it will be at index 0. However, delete_min's overall
# complexity is not O(1). Because the parent node is removed, we must re-build the tree using the last node in
# the array and bubble that node downwards until the heap is fixed. This bubbling takes approximately O(log n)
# since it does not iterate through every node in the tree, just two per iteration, resulting in a final
# time complexity of O(log n)
```

Insert – lines 193 – 196

```
# The insert function is similar to inserting on the array implementation, but does need to do some work after
# appending the node. If the node added is smaller than previous nodes added, it needs to bubble upwards until the
# heap order is restored. For this, insert is not O(1) time, but O(log n), as it relies on the other queue
# function decrease_key to perform the bubbling up, which has a complexity of O(log n)
```

Decrease_key – lines 207 – 212

```
# Decrease_key is the other core function of the heap priority queue. Whenever a distance value is updated, either
# when the node is initially inserted to the tree, or in the main Dijkstra's algorithm when distances are counted,
# decrease_key is used to update the value in the heap and fix the heap as necessary. Similar to delete_min, except
```

```
# the values are traveling up the tree, not down, since the key is being
decreased. The bubbling action is the same
# where in the worst case, a node can travel from the very lowest level of
the heap to the very first parent node.
# This worst case scenario dictates the run time of the function, making it
O(log n) time.
```

Make_queue – lines 259 – 262

```
# Make_queue works the same as in the array implementation of the queue,
however, the time complexity is slightly
# different. It iterates through every node in the array, a total of n times,
which is the same. However,
# the insert function has a time complexity of O(log n) because it relies on
decrease_key, which is different from
# the O(1) insert of the array implementation, making make_queue here O(n log
n) time.
```

Size – line 267

```
# Size() is the same as the array queue version, just returning the number of
nodes in the heap array at O(1) time.
```

Time Complexity for main Dijkstra algorithm functions

GetShortestPath – lines 280 – 285

```
# Get shortest path is the function used to iterate through the prev loop and
save the path and edges to be
# displayed on the GUI. As mentioned in the comments inside the function, it
is all O(1) functions except for the
# while loop that continues until the final node is found. Since the shortest
path will always be a small fraction
# of the entire set of nodes (unless the graph is a line of points) then the
time complexity is not quite O(n)
# since the while loop will never have to iterate through every node, so I
would more accurately call it
# O(s), where s is the number of nodes in the shortest path between source
and dest.
```

ComputeShortestPath – line 330 – 356

```
# computeShortestPaths is the code for the main Dijksta's algorithm. The
variable passed in from the GUI determines
# which queue implementation to use, then runs the same algorithm for both.
All the same functions are called on
# either queue type, with different results time complexity wise. (e.g.
decrease_key is called for the array queue,
# but does nothing within the implementation).
#
# As has been discussed in class, the complexity of Dijkstra's algorithm can
be simplified to the following:
# O(|V| (cost to insert + cost to delete min) + |E| (cost to decrease key))
```

```
#
# For the array queue implementation, there are two main factors to consider;
the while loop that
# essentially works as a for loop that iterates until the queue is empty, and
the delete_min function.
# The while loop is O(n) because every node is pushed onto the queue and the
while loop
# continues until all are removed, making it clearly O(n) time. Lastly,
delete_min for the array queue
# is O(n) (discussed more in depth inside of the class), because it has to
iterate through every node in the array
# to find the minimum value. Because our cost to delete min is O(n) and our
cost to insert is O(1), if we apply the
# equation above, our complexity would be O(|V| x O(n) + (|V|+|E|) x O(1)),
(|V| is the same as n, just kept |V| to
# match the terminology from class), our time would simplify to O(n^2) or
O(|V|^2), both are the same.
#
# For the heap queue implementation, there are similar factors at work, just
different time complexities for the
# queue functions. The while loop remains the same, an O(n) factor that is
unavoidable since every node needs to
# be processed in the algorithm. The difference comes from delete_min and
decrease_key, which are both O(log n)
# functions. Instead of having the compounding effect of delete_min looping
through each value of the array, the
# heap queue implementation makes big gains since both of these functions are
O(log n), making the total complexity
# for the Dijkstra's algorithm O(n log n), since the log n functions are
being called during each loop of the
# main while loop. Applying the same equation above to this implementation,
we get:
# O(|V| x O(log n) + (|V|+|E|) x O(log n), which can simplify down to O((|V|
+ |E|) log n) since both sides are
# log n. Again saying that n = |V| as the array of all the nodes/vertices, we
have O((n + |E|) log n), which we
# could further simply down to O(n log n) since |E| is a constant value and
would not change the overal complexity.
```

## Empirical Analysis of Project 3 Code

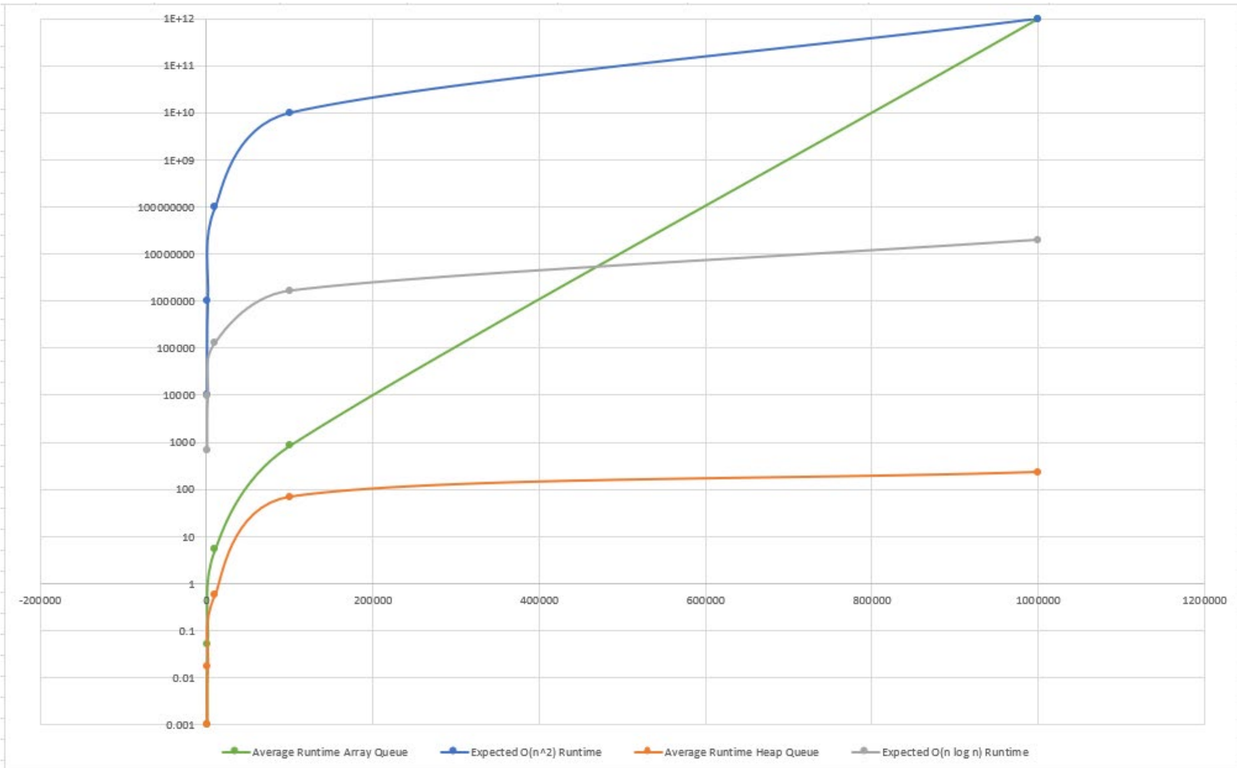
The first table is the run time of the Array Implementation (table is split into two screenshots to fit the word document).

| Array Implementation | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| N # of Points | Test 1 | Test 2 | Test 3 | Test 4 | Test 5 | Average Runtime | Expected O(n^2) | Seeds Used |
| 100 | 0.001 | 0.001002 | 0.001 | 0.001001 | 0.001 | 0.0010006 | 10000 | 55, 80, 99, 25,3 |
| 1000 | 0.052012 | 0.048504 | 0.052013 | 0.050013 | 0.051012 | 0.0507108 | 1000000 | 60, 91, 4, 19, 71 |
| 10000 | 5.485227 | 5.48367 | 5.397827 | 5.193173 | 5.412509 | 5.3944812 | 100000000 | 88, 7, 15, 29, 33 |
| 100000 | 833.707143 | 824.090715 | 823.461168 | 895.829108 | 891.377459 | 853.6931186 | 10000000000 | 44, 13, 28, 79, 105 |
| 1000000 | 1.00E+12 | 1.00E+12 | 1.00E+12 | 1.00E+12 | 1.00E+12 | 1.00E+12 | 1E+12 | N/A |

The second table is the Heap Implementation runtime and expected times table.

| Heap Implementation | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| N # of Points | Test 1 | Test 2 | Test 3 | Test 4 | Test 5 | Average Runtime | Expected O(n log n) | |
| 100 | 0.001 | 0.001001 | 0.001 | 0.001 | 0.001 | 0.0010002 | 664.385619 | 55, 80, 99, 25,3 |
| 1000 | 0.017004 | 0.017022 | 0.017004 | 0.017004 | 0.017004 | 0.0170076 | 9965.784285 | 60, 91, 4, 19, 71 |
| 10000 | 0.588142 | 0.591133 | 0.589143 | 0.585142 | 0.585693 | 0.5878506 | 132877.1238 | 88, 7, 15, 29, 33 |
| 100000 | 75.920164 | 64.835658 | 65.0277701 | 75.440055 | 66.045932 | 69.45391582 | 1660964.047 | 44, 13, 28, 79, 105 |
| 1000000 | 229.064501 | 235.012054 | 228.71036 | 237.89456 | 230.32601 | 232.201497 | 19931568.57 | 39, 8, 96, 67, 103 |

Lastly, the graph made from the two tables is as follows. The graph plots the average runtime for each test result for each implementation, along with the expected runtime for both implementations, resulting in 4 series being shown.




# Empirical Analysis - Conclusions

As is seen in the graph above, at all levels of input, the heap queue outperforms the array queue implementation. The margins are extremely small at the lower levels but become much more significant at 10,000 points and especially 100,000 points. The tests were not run for the array queue for 1,000,000 points, but it is safe to expect that there would be an even greater margin of difference between the two implementations.

There are some quirks with the graph that should be addressed.

Firstly, both the array and heap queues outperformed the expected runtimes for each of the expected times. This happened in my previous process, and it is safe to say that this “time gain” is due to the speed of the computer in handling many of the simple calculations that take place in the algorithm. I don’t believe this disproves the time complexity of either of the algorithms, because the way the code is set up, the array queue uses a O(n) function across every node in the queue, resulting in the O(n^2) expected time. That I am certainly confident in, which leads me to believe that the difference between expected and average is just a result of hardware.

The other quirk that came up during my testing was the sudden jump between the 10,000 point and 100,000 point tests for the heap queue implementation. The array queue results do not concern me, as they seem to fit the expected pattern of a dramatic increase in runtime with each factor of 10 added to the input. Dr. Ventura mentioned on slack having a run time of about 5 seconds for 10,000 array queue test, which matches my output, so I am confident in the array queue results being fair. The heap queue however jumps from a sub 1 second time for the tests up to 10,000 points, then suddenly goes up to 65 to 75 seconds per test. This is still fitting of the expectation that the heap queue will severely outperform the array queue, but the jump seems a little sudden.

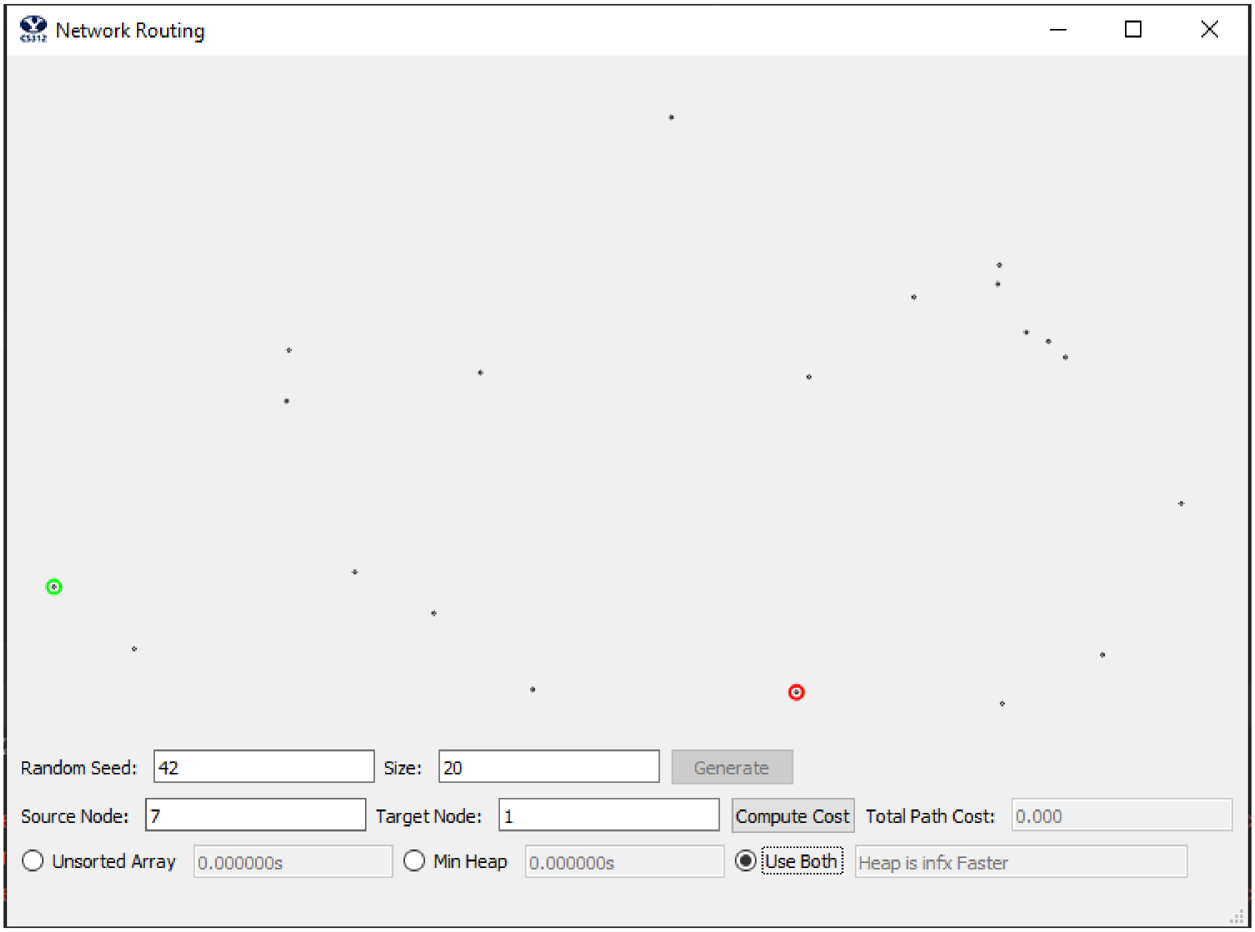
I looked around at slack and talked to other students and saw a very wide variety of results, ranging from like mine to significantly faster or slower, making it hard to be sure about what the deal was. I double checked my implementation and noticed I was using dictionaries to store the index of the heap array as well as for the distance array, so I switched those out for normal python arrays, but did not see any significant change in result.

When I would run the 100,000 or 1,000,000 point tests, even just for the heap queue, the GUI window would immediately stop responding and load an error. If I left it alone for a while, it would eventually run to completion and give me the result, but this leads me to believe that maybe it is the unresponsiveness of the GUI that is resulting in the strange run times for the heap. This is just speculation, and I should in the future maybe try the test without the GUI and make my own script to generate points and solve for the array.

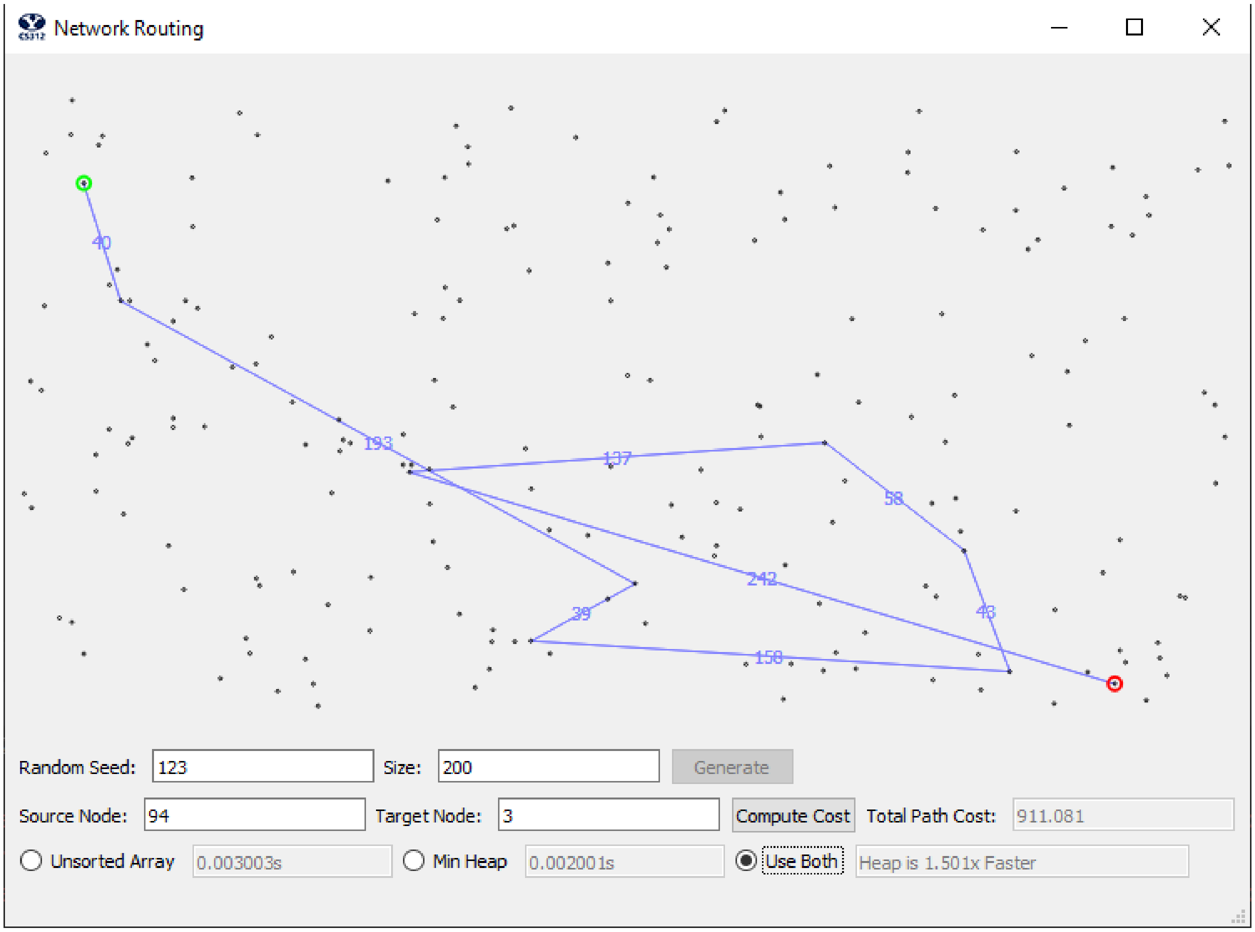
The strange result was worth mentioning, and I am also prone to leaving this up to a hardware issue as well that may have affected performance on the very large tests. I have been noticing my computer having difficulty running high memory usage processes, if the RAM shows more than 50% memory usage, things will start crashing and failing. I'm hoping to take my computer into a repair shop and see if there are any hardware issues, especially before the next project so that I can get a better result, if the hardware was an issue in the first place.

## Working Screenshots

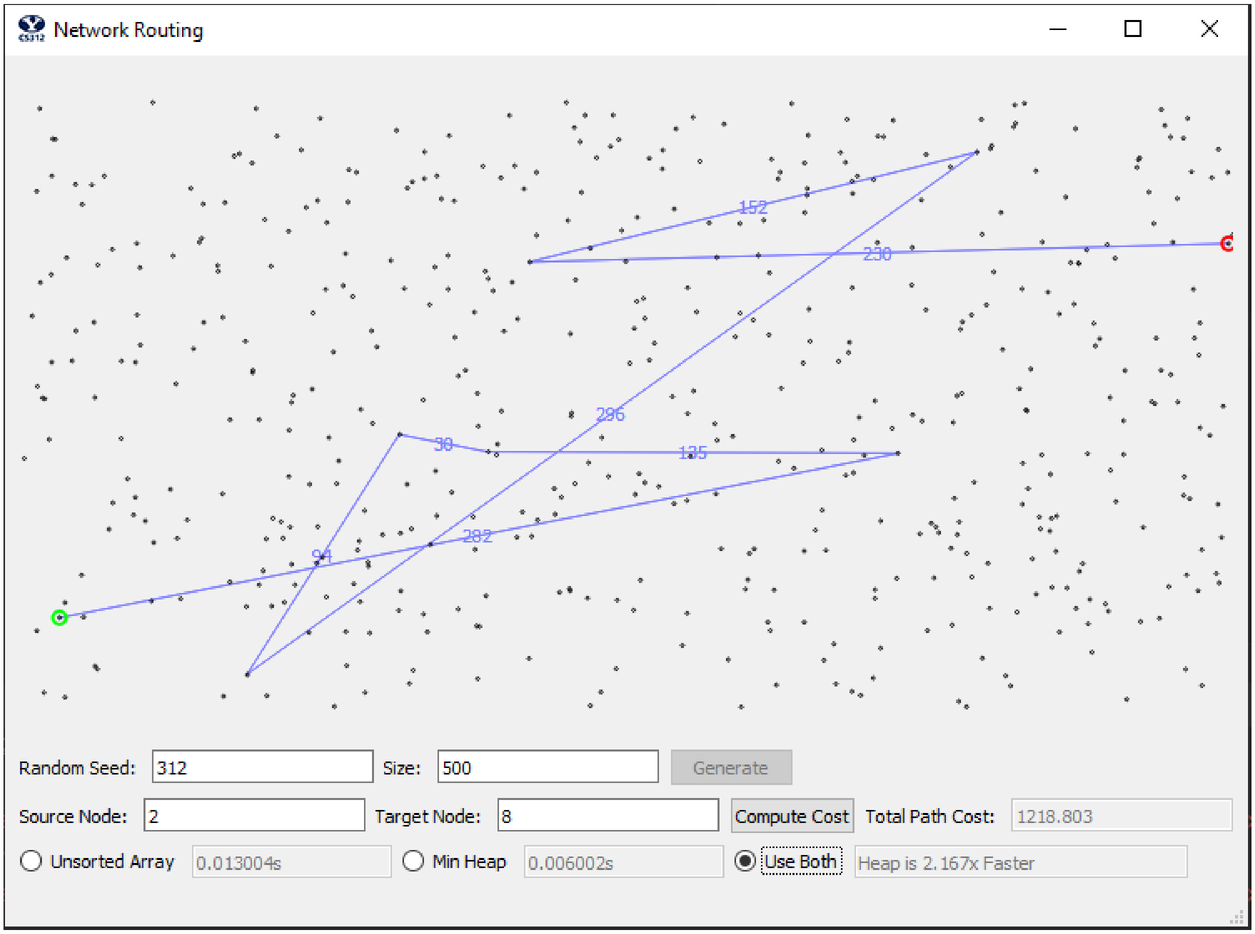
Screenshot #1 – Random seed 42, Size 20, Node 7 to Node 1


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

Screenshot #2 – Random seed 123, Size 200, Node 94 to Node 3




Screenshot #3 – Random seed 312, Size 500, Node 2 to Node 8


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
0.013004s
Min Heap
0.006002s
Use Both
Heap is 2.167x Faster

# Source Code Files

NetworkRoutingSolver.py

```
#!/usr/bin/python3


from CS312Graph import *
import time

# Abstract Queue class to build on for two implementations used in project 3
class Queue:
    def __init__(self):
        self.nodes = []
        self.array_indices = []

    def delete_min(self, dist):
        pass

    def decrease_key(self, node, dist):
        pass

    def insert(self,  node, index, dist):
        pass

    def make_queue(self, network_nodes, dist):
        pass

    def size(self):
        pass

# Priority Queue child class using an unsorted array to solve for shortest
path
class ArrayQueue(Queue):

    # delete_min with unsorted array. With the unsorted array implementation,
delete_min iterates through every
    # node in the array to find out which has the shortest distance in the
dist array, saves the index, and
    # removes and returns the node that was popped from the queue. Due to the
linear nature of arrays, there is
    # not a better way to find the min without checking every value, so the
implementation of this function
    # is time complexity O(n)
    def delete_min(self, dist):
        # Initialize min and min index used to find min value - O(1) for both
variable initializations
        min = 10000000
        minindex = 0

        # Loop through all nodes in node array of queue - O(n)
        for index, node in enumerate(self.nodes):
            # Get distance from array for node id - O(1)
            distance = dist[node.node_id]
```

```
            # If statement, if distance is new min then save it - O(1)
            if distance < min:
                min = distance  # Save min - O(1)
                minindex = index  # Save min index - O(1)

        # Save node at minindex to return and use in Dijkstra's algorithm
calculations - O(1)
        nodetoremove = self.nodes[minindex]

        # Remove node at minindex from node array - O(1) removing from an
array at index
        self.nodes.remove(nodetoremove)

        return nodetoremove  # Return node - O(1)

    # Insert with unsorted array - This function simply uses the Python
list.append() function to add the next node
    # to the array. Because append adds the node to the end of the array, and
the array is unsorted meaning that no
    # work post appending needs to be done, the time complexity of insert()
is O(1) time.
    def insert(self, node, index, dist):
        # Append new node to end of array list - O(1)
        self.nodes.append(node)

    # decrease_key - Decrease key does not do anything in the array
implementation of the priority queue, so there
    # is no time complexity to report for this function. It is called in the
algorithm no matter which queue type is
    # used, but is simply passed when the array implementation is in use.
    def decrease_key(self, node, dist):
        pass

    # Make queue - The initial make queue function takes all of the nodes in
the network and adds them to the
    # unsorted array using the insert() function. Insert runs at O(1) time,
but because every node must be added to the
    # array through a for loop, we call insert() n times where n = number of
nodes in network. This makes the time
    # complexity of make_queue = O(n)
    def make_queue(self, network_nodes, dist):
        # Loop through each node and add to node array - O(n)
        for index, node in enumerate(network_nodes):
            self.insert(node, index, dist)  # Add node using insert function
- O(1)

    # A helper function to get the size of the array currently. Len(object)
in python returns the integer of the
    # size of the list with O(1) time, so the time complexity of this is O(1)
    def size(self):
        return len(self.nodes)

# Binary Heap implementation of parent Queue class. The same functions are
found from above, but the implementation of
# has a very different effect on performance time wise. Functions such as
insert and make queue run with a slightly
# longer run time because the array must be sorted to retain binary heap
```

```
functionality, but performance is made up
# with a dramatically faster delete_min function, from O(n) in the array
queue, to O(log n) in the heap queue.
# This time save makes the performance of the heap queue many times faster
than the performance of the array queue in
# empirical testing.
class HeapQueue(Queue):

    # delete_min function using binary heap implementation. This is is one of
the two core functions of the binary
    # heap implementation. Because the node with the shortest distance is
bubbled up to the top with decrease_key,
    # finding the minimum value is very fast, O(1) since we know it will be
at index 0. However, delete_min's overall
    # complexity is not O(1). Because the parent node is removed, we must re-
build the tree using the last node in
    # the array and bubble that node downwards until the heap is fixed. This
bubbling takes approximately O(log n)
    # since it does not iterate through every node in the tree, just two per
iteration, resulting in a final
    # time complexity of O(log n)
    def delete_min(self, dist):
        parent_index = 0  # Set variable to reference index 0 of array - O(1)
        nodetoremove = self.nodes[parent_index]  # Save node at index 0 (min)
to return at the end - O(1)

        # Get node at end of try self.size()-1 - O(1)
        heap_end_node = self.nodes[self.size() - 1]

        # Remove node at end of array - O(1)
        self.nodes.remove(heap_end_node)

        # If node removed was final node in tree, return immediately - if
statement O(1)
        if self.size() == 0:
            return nodetoremove  # Return node at index 0 - O(1)

        # Replace node at index 0 with final node of heap - O(1)
        self.nodes[parent_index] = heap_end_node

        # Update indices array
        self.array_indices[nodetoremove.node_id] = -1  # Update value in
array - O(1)
        self.array_indices[heap_end_node.node_id] = parent_index  # Update
value in array - O(1)

        # While new dist[self.nodes[0]] > either child, percolate down
        # While in best cases this will only run one loop, on average or
worst case, the node will bubble down to the
        # bottom of the tree, making time complexity for the while loop O(log
n)
        while True:
            # Get two children nodes, index * 2 + 1 (left), index * 2 + 2
(right)
            left_child_index = (parent_index * 2) + 1  # Calculate index -
O(1)
            right_child_index = (parent_index * 2) + 2  # Calculate index -
```

```
O(1)

            # Get parent node and distance
            parent_node = self.nodes[parent_index]  # Get value from heap
array - O(1)
            parent_dist = dist[parent_node.node_id]  # Get value from dist
array - O(1)

            # If left child exists, get left child node and distances - If
statement comparison - O(1)
            if left_child_index < self.size():
                left_child_node = self.nodes[left_child_index]  # Get value
from heap array - O(1)
                left_child_dist = dist[left_child_node.node_id]  # Get value
from dist array - O(1)
            else:
                left_child_dist = float('inf')  # Set dist value to inf -
O(1)
                left_child_node = None  # Set node to none - O(1)

            # If right child exists, get right child node and distances - If
statement comparison - O(1)
            if right_child_index < self.size():
                right_child_node = self.nodes[right_child_index]  # Get value
from heap array - O(1)
                right_child_dist = dist[right_child_node.node_id]  # Get
value from dist array - O(1)
            else:
                right_child_dist = float('inf')  # Set dist value to inf -
O(1)
                right_child_node = None  # Set node to none - O(1)

            # Initialize boolean checker variable - O(1)
            parent_swapped = False

            # Compare values, choose which child is smallest (compare
children, then compare with parent)
            # Default behavior will swap with the left node if left and right
distance values are tied.

            # Left child is smaller or equal to right try swap with left - If
statement comparison - O(1)
            if left_child_dist <= right_child_dist:
                # If left child is smaller than parent then swap - If
statement comparison - O(1)
                if left_child_dist < parent_dist:
                    # Swap parent and child nodes
                    self.nodes[parent_index] = left_child_node  # Set value
in heap array - O(1)
                    self.nodes[left_child_index] = parent_node  # Set value
in heap array - O(1)

                    # Update indices of swapped nodes
                    self.array_indices[parent_node.node_id] =
left_child_index  # Set value in array - O(1)
                    self.array_indices[left_child_node.node_id] =
parent_index  # Set value in array - O(1)
```

```
                    # Update parent index for next iteration
                    parent_index = left_child_index  # Update variable value
- O(1)
                    parent_swapped = True  # Update boolean value - O(1)
            else:
                # If right child is smaller than parent then swap - If
statement comparison - O(1)
                if right_child_dist < parent_dist:
                    # Swap parent and child nodes
                    self.nodes[parent_index] = right_child_node  # Set value
in heap array - O(1)
                    self.nodes[right_child_index] = parent_node  # Set value
in heap array - O(1)

                    # Update indices of swapped nodes
                    self.array_indices[parent_node.node_id] =
right_child_index  # Set value in dictionary - O(1)
                    self.array_indices[right_child_node.node_id] =
parent_index  # Set value in dictionary - O(1)

                    # Update parent index for next iteration
                    parent_index = right_child_index  # Update variable value
- O(1)
                    parent_swapped = True  # Update boolean value - O(1)

            # If neither is bigger, break loop
            if not parent_swapped:  # If statement comparison - O(1)
                break  # Break - O(1)

        # Return deleted node - O(1)
        return nodetoremove

    # The insert function is similar to inserting on the array
implementation, but does need to do some work after
    # appending the node. If the node added is smaller than previous nodes
added, it needs to bubble upwards until the
    # heap order is restored. For this, insert is not O(1) time, but O(log
n), as it relies on the other queue
    # function decrease_key to perform the bubbling up, which has a
complexity of O(log n)
    def insert(self, node, index, dist):
        # Add node to heap array - List.append() - O(1)
        self.nodes.append(node)

        # Store index in heap array in dictionary - Append value to
dictionary - O(1)
        self.array_indices.append(index)

        # Call decrease_key to fix heap order if needed - O(log n)
        self.decrease_key(node, dist)

    # Decrease_key is the other core function of the heap priority queue.
Whenever a distance value is updated, either
    # when the node is initially inserted to the tree, or in the main
Dijkstra's algorithm when distances are counted,
    # decrease_key is used to update the value in the heap and fix the heap
```

```
as necessary. Similar to delete_min, except
    # the values are traveling up the tree, not down, since the key is being
decreased. The bubbling action is the same
    # where in the worst case, a node can travel from the very lowest level
of the heap to the very first parent node.
    # This worst case scenario dictates the run time of the function, making
it O(log n) time.
    def decrease_key(self, node, dist):
        # Check if node has already been popped off of the queue then return
early.
        # This happens when a node has a neighbor to a node that has already
been popped off the queue as the min value,
        # so the calculation does not need to happen.
        if self.array_indices[node.node_id] == -1:  # If statement comparison
- O(1)
            return  # Return - O(1)

        # Get location of node in heap array using indices dictionary
        child_index = self.array_indices[node.node_id]  # Get value from
array - O(1)

        # While node is smaller than parent node, percolate
        # As mentioned above, the while loop is what causes the O(log n), as
each loop represents a level up the heap
        # that node moves up, approximately O(log n).
        while True:
            # If node is already top of heap, break
            if child_index == 0:  # If statement comparison - O(1)
                break  # Break - O(1)

            # Get child node from array
            child_node = self.nodes[child_index]  # Get value from heap array
- O(1)

            # Get parent node using (child_index - 1) // 2
            parent_index = (child_index - 1) // 2  # Calculate index value -
O(1)
            parent_node = self.nodes[parent_index]  # Get value from heap
array - O(1)

            # compare distances using node id and dist array
            child_dist = dist[child_node.node_id]  # Get value from dist
array - O(1)
            parent_dist = dist[parent_node.node_id]  # Get value from dist
array - O(1)

            # If child is smaller than parent - If statement comparison -
O(1)
            if child_dist < parent_dist:
                # Swap child and parent
                self.nodes[child_index] = parent_node  # Set value in array -
O(1)
                self.nodes[parent_index] = child_node  # Set value in array -
O(1)

                # Update indices array if swap happens
                self.array_indices[child_node.node_id] = parent_index  # Set
```

```
value in array - O(1)
                self.array_indices[parent_node.node_id] = child_index  # Set
value in array - O(1)

                # Set child_index = parent_index for next iteration
                child_index = parent_index  # Update variable value - O(1)

            # Else, node is in the correct spot, break while loop
            else:
                break

    # Make_queue works the same as in the array implementation of the queue,
however, the time complexity is slightly
    # different. It iterates through every node in the array, a total of n
times, which is the same. However,
    # the insert function has a time complexity of O(log n) because it relies
on decrease_key, which is different from
    # the O(1) insert of the array implementation, making make_queue here O(n
log n) time.
    def make_queue(self, network_nodes, dist):
        for index, node in enumerate(network_nodes):
            self.insert(node, index, dist)

    # Size() is the same as the array queue version, just returning the
number of nodes in the heap array at O(1) time.
    def size(self):
        return len(self.nodes)


class NetworkRoutingSolver:
    def __init__( self):
        self.prev = {}

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

    # Get shortest path is the function used to iterate through the prev loop
and save the path and edges to be
    # displayed on the GUI. As mentioned in the comments inside the function,
it is all O(1) functions except for the
    # while loop that continues until the final node is found. Since the
shortest path will always be a small fraction
    # of the entire set of nodes (unless the graph is a line of points) then
the time complexity is not quite O(n)
    # since the while loop will never have to iterate through every node, so
I would more accurately call it
    # O(s), where s is the number of nodes in the shortest path between
source and dest.
    def getShortestPath( self, destIndex):
        self.dest = destIndex  # Save parameter to local variable - O(1)
        path_edges = []  # Initialize empty array - O(1)
        total_length = 0  # Initialize length variable - O(1)
        node = self.network.nodes[destIndex]  # Get node from network array -
O(1)

        # Iterate through prev array starting from destination node, until
```

```
reaching None which is the value for prev
        # of the starting node, saving path length and edges on the way.

        # While there could be situations where each node is an edge on the
way to the destination (i.e. all points
        # were in a straight line) that won't happen with our distributions.
The path from source to destination is
        # generally a small fraction of the entire set of nodes, so I won't
say the time complexity is O(n), but a
        # more arbitrary O(s) where s is the number of edges between the
source and destination node.
        while True:
            # Get node from previous array
            previous_node_id = self.prev[node.node_id]  # Get value from
array - O(1)

            # Previous node is None, reached source node so break while loop
            if previous_node_id is None:  # If statement comparison - O(1)
                break

            previous_node = self.network.nodes[previous_node_id]  # Get node
from array - O(1)

            # Get the edge between previous and the current node
            edge_to_add = None  # Initialize variable to use

            # Loop through neighbors of previous node to find edge from
previous to current node

            # Similar to the neighbor loop in the main Dijkstra's algorithm,
there are usually a very small number of
            # neighbors for each node due to the code generation, so this for
loop should not be considered O(n) time.
            # It is instead something much smaller, which I will arbitrarily
call O(m) where m is the number of
            # neighbor nodes for the current node.
            for edge in previous_node.neighbors:
                if edge.dest == node:  # If statement comparison - O(1)
                    edge_to_add = edge  # Save value to variable - O(1)
                    break  # Break - O(1)

            # Append saved edge to path edges - Append to array - O(1)
            path_edges.append((edge_to_add.src.loc, edge_to_add.dest.loc,
'{:.0f}'.format(edge_to_add.length)))
            total_length += edge_to_add.length  # Update variable value -
O(1)
            node = edge_to_add.src  # Update node variable valeu - O(1)

        return {'cost': total_length, 'path': path_edges}  # Return array -
O(1)

    # computeShortestPaths is the code for the main Dijksta's algorithm. The
variable passed in from the GUI determines
    # which queue implementation to use, then runs the same algorithm for
both. All the same functions are called on
    # either queue type, with different results time complexity wise. (e.g.
decrease_key is called for the array queue,
```

```
    # but does nothing within the implementation).
    #
    # As has been discussed in class, the complexity of Dijkstra's algorithm
can be simplified to the following:
    # O(|V| (cost to insert + cost to delete min) + |E| (cost to decrease
key))
    #
    # For the array queue implementation, there are two main factors to
consider; the while loop that
    # essentially works as a for loop that iterates until the queue is empty,
and the delete_min function.
    # The while loop is O(n) because every node is pushed onto the queue and
the while loop
    # continues until all are removed, making it clearly O(n) time. Lastly,
delete_min for the array queue
    # is O(n) (discussed more in depth inside of the class), because it has
to iterate through every node in the array
    # to find the minimum value. Because our cost to delete min is O(n) and
our cost to insert is O(1), if we apply the
    # equation above, our complexity would be O(|V| x O(n) + (|V|+|E|) x
O(1)), (|V| is the same as n, just kept |V| to
    # match the terminology from class), our time would simplify to O(n^2) or
O(|V|^2), both are the same.
    #
    # For the heap queue implementation, there are similar factors at work,
just different time complexities for the
    # queue functions. The while loop remains the same, an O(n) factor that
is unavoidable since every node needs to
    # be processed in the algorithm. The difference comes from delete_min and
decrease_key, which are both O(log n)
    # functions. Instead of having the compounding effect of delete_min
looping through each value of the array, the
    # heap queue implementation makes big gains since both of these functions
are O(log n), making the total complexity
    # for the Dijkstra's algorithm O(n log n), since the log n functions are
being called during each loop of the
    # main while loop. Applying the same equation above to this
implementation, we get:
    # O(|V| x O(log n) + (|V|+|E|) x O(log n), which can simplify down to
O((|V| + |E|) log n) since both sides are
    # log n. Again saying that n = |V| as the array of all the
nodes/vertices, we have O((n + |E|) log n), which we
    # could further simply down to O(n log n) since |E| is a constant value
and would not change the overal complexity.
    def computeShortestPaths(self, srcIndex, use_heap=False):
        self.source = srcIndex  # Save parameter to local variable - O(1)
        t1 = time.time()

        # Choose appropriate data structure based on settings - If statement
blocks - O(1)
        if use_heap:
            queue = HeapQueue()
        else:
            queue = ArrayQueue()
            pass

        # Get all node in network and initialize dist and prev arrays
```

```
        node_array = self.network.nodes  # Save class variable to local
variable to use - O(1)
        dist = []  # Initialize empty dictionary - O(1)

        # Iterate through each node in network and initialize dist and prev
values - O(n)
        for node in node_array:
            dist.append((float('inf')))  # Initialize value in array - O(1)
            self.prev[node.node_id] = None  # Initialize value in array -
O(1)

        # Initialize starting node distance to 0
        dist[self.source] = 0  # Set value in array - O(1)

        # Call make queue on array of nodes
        queue.make_queue(node_array, dist)  # Make queue - O(n) for array,
O(n log n) for binary heap

        # While queue size is not 0, repeat Dijkstra's
        # Because the queue starts with all nodes, the while loop will repeat
n times - O(n) time complexity
        while queue.size() != 0:

            # Call delete min to pop and get node with smallest distance
            current_node = queue.delete_min(dist)  # Delete_min - Array Queue
O(n), Heap Queue O(log n)

            # Get neighboring nodes to check for updating distances
            current_neighbors = current_node.neighbors  # Get value from node
object - O(1)

            # Iterate through neighbor node edges - For loop s times where s
is number of neighbors - O(s) time.
            # The number of neighbors is at most n - 1, but this code only
creates sets of 3 neighbors, making the for
            # loop not significant time wise, and thus indicated by O(s)
where s is number of neighbors.
            for index, neighbor in enumerate(current_neighbors):

                # Get source, dest, and length of each edge for easy
reference
                src = neighbor.src  # Save value from neighbor node object -
O(1)
                dest = neighbor.dest  # Save value from neighbor node object
- O(1)
                length = neighbor.length  # Save value from neighbor node
object - O(1)

                # Current path is shorter than path previously stored in dist
array, update dist and prev
                if dist[dest.node_id] > dist[src.node_id] + length:  # If
statement comparison - O(1)
                    dist[dest.node_id] = dist[src.node_id] + length  # Update
value in array - O(1)
                    self.prev[dest.node_id] = src.node_id  # Update value in
array - O(1)
                    queue.decrease_key(dest, dist)  # decrease_key - Array
```

```
Queue N/A, Heap Queue O(log n)

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

CS312Graph.py

```
#!/usr/bin/python3


class CS312GraphEdge:
    def __init__( self, src_node, dest_node, edge_length ):
        self.src    = src_node
        self.dest   = dest_node
        self.length= edge_length

    def __repr__( self ):
        return self.__str__()

    def __str__( self ):
        return '(src={} dest={}
length={})'.format(self.src,self.dest,self.length)

class CS312GraphNode:
    def __init__( self, node_id, node_loc ):
        self.node_id    = node_id
        self.loc        = node_loc
        self.neighbors = [] #node_neighbors

    def addEdge( self, neighborNode, weight ):
        self.neighbors.append( CS312GraphEdge(self,neighborNode,weight) )

    def __str__( self ):
        neighbors = [edge.dest.node_id for edge in self.neighbors]
        return 'Node(id:{},neighbors:{})'.format(self.node_id,neighbors)


class CS312Graph:
    def __init__( self, nodeList, edgeList ):
        self.nodes     = []
        for i in range(len(nodeList)):
            self.nodes.append( CS312GraphNode( i, nodeList[i] ) )

        for i in range(len(nodeList)):
            neighbors = edgeList[i]
            for n in neighbors:
                self.nodes[i].addEdge( self.nodes[n[0]], n[1] )

    def __str__( self ):
        s = []
        for n in self.nodes:
            s.append(n.neighbors)
        return str(s)

    def getNodes( self ):
        return self.nodes
```

Proj3GUI.py

```
#!/usr/bin/python3


from CS312Graph import *
import time

# Abstract Queue class to build on for two implementations used in project 3
class Queue:
    def __init__(self):
        self.nodes = []
        self.array_indices = {}

    def delete_min(self, dist):
        pass

    def decrease_key(self, node, dist):
        pass

    def insert(self,  node, index, dist):
        pass

    def make_queue(self, network_nodes, dist):
        pass

    def size(self):
        pass

# Priority Queue child class using an unsorted array to solve for shortest
path
class ArrayQueue(Queue):

    # delete_min with unsorted array. With the unsorted array implementation,
delete_min iterates through every
    # node in the array to find out which has the shortest distance in the
dist array, saves the index, and
    # removes and returns the node that was popped from the queue. Due to the
linear nature of arrays, there is
    # not a better way to find the min without checking every value, so the
implementation of this function
    # is time complexity O(n)
    def delete_min(self, dist):
        # Initialize min and min index used to find min value - O(1) for both
variable initializations
        min = 10000000
        minindex = 0

        # Loop through all nodes in node array of queue - O(n)
        for index, node in enumerate(self.nodes):
            # Get distance from array for node id - O(1)
            distance = dist[node.node_id]

            # If statement, if distance is new min then save it - O(1)
            if distance < min:
                min = distance  # Save min - O(1)
                minindex = index  # Save min index - O(1)
```

```
        # Save node at minindex to return and use in Dijkstra's algorithm
calculations - O(1)
        nodetoremove = self.nodes[minindex]

        # Remove node at minindex from node array - O(1) removing from an
array at index
        self.nodes.remove(nodetoremove)

        return nodetoremove  # Return node - O(1)

    # Insert with unsorted array - This function simply uses the Python
list.append() function to add the next node
    # to the array. Because append adds the node to the end of the array, and
the array is unsorted meaning that no
    # work post appending needs to be done, the time complexity of insert()
is O(1) time.
    def insert(self, node, index, dist):
        # Append new node to end of array list - O(1)
        self.nodes.append(node)

    # decrease_key - Decrease key does not do anything in the array
implementation of the priority queue, so there
    # is no time complexity to report for this function. It is called in the
algorithm no matter which queue type is
    # used, but is simply passed when the array implementation is in use.
    def decrease_key(self, node, dist):
        pass

    # Make queue - The initial make queue function takes all of the nodes in
the network and adds them to the
    # unsorted array using the insert() function. Insert runs at O(1) time,
but because every node must be added to the
    # array through a for loop, we call insert() n times where n = number of
nodes in network. This makes the time
    # complexity of make_queue = O(n)
    def make_queue(self, network_nodes, dist):
        # Loop through each node and add to node array - O(n)
        for index, node in enumerate(network_nodes):
            self.insert(node, index, dist)  # Add node using insert function
- O(1)

    # A helper function to get the size of the array currently. Len(object)
in python returns the integer of the
    # size of the list with O(1) time, so the time complexity of this is O(1)
    def size(self):
        return len(self.nodes)

# Binary Heap implementation of parent Queue class. The same functions are
found from above, but the implementation of
# has a very different effect on performance time wise. Functions such as
insert and make queue run with a slightly
# longer run time because the array must be sorted to retain binary heap
functionality, but performance is made up
# with a dramatically faster delete_min function, from O(n) in the array
queue, to O(log n) in the heap queue.
# This time save makes the performance of the heap queue many times faster
```

```
than the performance of the array queue in
# empirical testing.
class HeapQueue(Queue):

    # delete_min function using binary heap implementation. This is is one of
the two core functions of the binary
    # heap implementation. Because the node with the shortest distance is
bubbled up to the top with decrease_key,
    # finding the minimum value is very fast, O(1) since we know it will be
at index 0. However, delete_min's overall
    # complexity is not O(1). Because the parent node is removed, we must re-
build the tree using the last node in
    # the array and bubble that node downwards until the heap is fixed. This
bubbling takes approximately O(log n)
    # since it does not iterate through every node in the tree, just two per
iteration, resulting in a final
    # time complexity of O(log n)
    def delete_min(self, dist):
        parent_index = 0  # Set variable to reference index 0 of array - O(1)
        nodetoremove = self.nodes[parent_index]  # Save node at index 0 (min)
to return at the end - O(1)

        # Get node at end of try self.size()-1 - O(1)
        heap_end_node = self.nodes[self.size() - 1]

        # Remove node at end of array - O(1)
        self.nodes.remove(heap_end_node)

        # If node removed was final node in tree, return immediately - if
statement O(1)
        if self.size() == 0:
            return nodetoremove  # Return node at index 0 - O(1)

        # Replace node at index 0 with final node of heap - O(1)
        self.nodes[parent_index] = heap_end_node

        # Update indices array
        self.array_indices.pop(nodetoremove)  # Dictionary.pop O(1)
        self.array_indices[heap_end_node] = parent_index  # Set new index
value in dictionary - O(1)

        # While new dist[self.nodes[0]] > either child, percolate down
        # While in best cases this will only run one loop, on average or
worst case, the node will bubble down to the
        # bottom of the tree, making time complexity for the while loop O(log
n)
        while True:
            # Get two children nodes, index * 2 + 1 (left), index * 2 + 2
(right)
            left_child_index = (parent_index * 2) + 1  # Calculate index -
O(1)
            right_child_index = (parent_index * 2) + 2  # Calculate index -
O(1)

            # Get parent node and distance
            parent_node = self.nodes[parent_index]  # Get value from heap
array - O(1)
```

```
            parent_dist = dist[parent_node.node_id]  # Get value from dist
array - O(1)

            # If left child exists, get left child node and distances - If
statement comparison - O(1)
            if left_child_index < self.size():
                left_child_node = self.nodes[left_child_index]  # Get value
from heap array - O(1)
                left_child_dist = dist[left_child_node.node_id]  # Get value
from dist array - O(1)
            else:
                left_child_dist = float('inf')  # Set dist value to inf -
O(1)
                left_child_node = None  # Set node to none - O(1)

            # If right child exists, get right child node and distances - If
statement comparison - O(1)
            if right_child_index < self.size():
                right_child_node = self.nodes[right_child_index]  # Get value
from heap array - O(1)
                right_child_dist = dist[right_child_node.node_id]  # Get
value from dist array - O(1)
            else:
                right_child_dist = float('inf')  # Set dist value to inf -
O(1)
                right_child_node = None  # Set node to none - O(1)

            # Initialize boolean checker variable - O(1)
            parent_swapped = False

            # Compare values, choose which child is smallest (compare
children, then compare with parent)
            # Default behavior will swap with the left node if left and right
distance values are tied.

            # Left child is smaller or equal to right try swap with left - If
statement comparison - O(1)
            if left_child_dist <= right_child_dist:
                # If left child is smaller than parent then swap - If
statement comparison - O(1)
                if left_child_dist < parent_dist:
                    # Swap parent and child nodes
                    self.nodes[parent_index] = left_child_node  # Set value
in heap array - O(1)
                    self.nodes[left_child_index] = parent_node  # Set value
in heap array - O(1)

                    # Update indices of swapped nodes
                    self.array_indices[parent_node] = left_child_index  # Set
value in dictionary - O(1)
                    self.array_indices[left_child_node] = parent_index  # Set
value in dictionary - O(1)

                    # Update parent index for next iteration
                    parent_index = left_child_index  # Update variable value
- O(1)
                    parent_swapped = True  # Update boolean value - O(1)
```

```
            else:
                # If right child is smaller than parent then swap - If
statement comparison - O(1)
                if right_child_dist < parent_dist:
                    # Swap parent and child nodes
                    self.nodes[parent_index] = right_child_node  # Set value
in heap array - O(1)
                    self.nodes[right_child_index] = parent_node  # Set value
in heap array - O(1)

                    # Update indices of swapped nodes
                    self.array_indices[parent_node] = right_child_index  #
Set value in dictionary - O(1)
                    self.array_indices[right_child_node] = parent_index  #
Set value in dictionary - O(1)

                    # Update parent index for next iteration
                    parent_index = right_child_index  # Update variable value
- O(1)
                    parent_swapped = True  # Update boolean value - O(1)

            # If neither is bigger, break loop
            if not parent_swapped:  # If statement comparison - O(1)
                break  # Break - O(1)

        # Return deleted node - O(1)
        return nodetoremove

    # The insert function is similar to inserting on the array
implementation, but does need to do some work after
    # appending the node. If the node added is smaller than previous nodes
added, it needs to bubble upwards until the
    # heap order is restored. For this, insert is not O(1) time, but O(log
n), as it relies on the other queue
    # function decrease_key to perform the bubbling up, which has a
complexity of O(log n)
    def insert(self, node, index, dist):
        # Add node to heap array - List.append() - O(1)
        self.nodes.append(node)

        # Store index in heap array in dictionary - Append value to
dictionary - O(1)
        self.array_indices[node] = index

        # Call decrease_key to fix heap order if needed - O(log n)
        self.decrease_key(node, dist)

    # Decrease_key is the other core function of the heap priority queue.
Whenever a distance value is updated, either
    # when the node is initially inserted to the tree, or in the main
Dijkstra's algorithm when distances are counted,
    # decrease_key is used to update the value in the heap and fix the heap
as necessary. Similar to delete_min, except
    # the values are traveling up the tree, not down, since the key is being
decreased. The bubbling action is the same
    # where in the worst case, a node can travel from the very lowest level
of the heap to the very first parent node.
```

```
    # This worst case scenario dictates the run time of the function, making
it O(log n) time.
    def decrease_key(self, node, dist):
        # Check if node has already been popped off of the queue then return
early.
        # This happens when a node has a neighbor to a node that has already
been popped off the queue as the min value,
        # so the calculation does not need to happen.
        if self.array_indices.get(node) is None:  # If statement comparison -
O(1)
            return  # Return - O(1)

        # Get location of node in heap array using indices dictionary
        child_index = self.array_indices[node]  # Get value from dictionary -
O(1)

        # While node is smaller than parent node, percolate
        # As mentioned above, the while loop is what causes the O(log n), as
each loop represents a level up the heap
        # that node moves up, approximately O(log n).
        while True:
            # If node is already top of heap, break
            if child_index == 0:  # If statement comparison - O(1)
                break  # Break - O(1)

            # Get child node from array
            child_node = self.nodes[child_index]  # Get value from heap array
- O(1)

            # Get parent node using (child_index - 1) // 2
            parent_index = (child_index - 1) // 2  # Calculate index value -
O(1)
            parent_node = self.nodes[parent_index]  # Get value from heap
array - O(1)

            # compare distances using node id and dist array
            child_dist = dist[child_node.node_id]  # Get value from dist
array - O(1)
            parent_dist = dist[parent_node.node_id]  # Get value from dist
array - O(1)

            # If child is smaller than parent - If statement comparison -
O(1)
            if child_dist < parent_dist:
                # Swap child and parent
                self.nodes[child_index] = parent_node  # Set value in array -
O(1)
                self.nodes[parent_index] = child_node  # Set value in array -
O(1)

                # Update indices array if swap happens
                self.array_indices[child_node] = parent_index  # Set value in
dictionary - O(1)
                self.array_indices[parent_node] = child_index  # Set value in
dictionary - O(1)

                # Set child_index = parent_index for next iteration
```

```
                child_index = parent_index  # Update variable value - O(1)

            # Else, node is in the correct spot, break while loop
            else:
                break

    # Make_queue works the same as in the array implementation of the queue, 
however, the time complexity is slightly
    # different. It iterates through every node in the array, a total of n 
times, which is the same. However,
    # the insert function has a time complexity of O(log n) because it relies 
on decrease_key, which is different from
    # the O(1) insert of the array implementation, making make_queue here O(n 
log n) time.
    def make_queue(self, network_nodes, dist):
        for index, node in enumerate(network_nodes):
            self.insert(node, index, dist)

    # Size() is the same as the array queue version, just returning the 
number of nodes in the heap array at O(1) time.
    def size(self):
        return len(self.nodes)


class NetworkRoutingSolver:
    def __init__( self):
        self.prev = {}

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

    # Get shortest path is the function used to iterate through the prev loop 
and save the path and edges to be
    # displayed on the GUI. As mentioned in the comments inside the function, 
it is all O(1) functions except for the
    # while loop that continues until the final node is found. Since the 
shortest path will always be a small fraction
    # of the entire set of nodes (unless the graph is a line of points) then 
the time complexity is not quite O(n)
    # since the while loop will never have to iterate through every node, so 
I would more accurately call it
    # O(s), where s is the number of nodes in the shortest path between 
source and dest.
    def getShortestPath( self, destIndex):
        self.dest = destIndex  # Save parameter to local variable - O(1)
        path_edges = []  # Initialize empty array - O(1)
        total_length = 0  # Initialize length variable - O(1)
        node = self.network.nodes[destIndex]  # Get node from network array - 
O(1)

        # Iterate through prev array starting from destination node, until 
reaching None which is the value for prev
        # of the starting node, saving path length and edges on the way.

        # While there could be situations where each node is an edge on the 
way to the destination (i.e. all points
```

```
        # were in a straight line) that won't happen with our distributions. The path from source to destination is
        # generally a small fraction of the entire set of nodes, so I won't say the time complexity is O(n), but a
        # more arbitrary O(s) where s is the number of edges between the source and destination node.
        while True:
            # Get node from previous array
            previous_node_id = self.prev[node.node_id]  # Get value from array - O(1)

            # Previous node is None, reached source node so break while loop
            if previous_node_id is None:  # If statement comparison - O(1)
                break

            previous_node = self.network.nodes[previous_node_id]  # Get node from array - O(1)

            # Get the edge between previous and the current node
            edge_to_add = None  # Initialize variable to use

            # Loop through neighbors of previous node to find edge from previous to current node

            # Similar to the neighbor loop in the main Dijkstra's algorithm, there are usually a very small number of
            # neighbors for each node due to the code generation, so this for loop should not be considered O(n) time.
            # It is instead something much smaller, which I will arbitrarily call O(m) where m is the number of
            # neighbor nodes for the current node.
            for edge in previous_node.neighbors:
                if edge.dest == node:  # If statement comparison - O(1)
                    edge_to_add = edge  # Save value to variable - O(1)
                    break  # Break - O(1)

            # Append saved edge to path edges - Append to array - O(1)
            path_edges.append((edge_to_add.src.loc, edge_to_add.dest.loc, '{:.0f}'.format(edge_to_add.length)))
            total_length += edge_to_add.length  # Update variable value - O(1)
            node = edge_to_add.src  # Update node variable valeu - O(1)

        return {'cost': total_length, 'path': path_edges}  # Return array - O(1)

    # computeShortestPaths is the code for the main Dijksta's algorithm. The variable passed in from the GUI determines
    # which queue implementation to use, then runs the same algorithm for both. All the same functions are called on
    # either queue type, with different results time complexity wise. (e.g. decrease_key is called for the array queue,
    # but does nothing within the implementation).
    #
    # As has been discussed in class, the complexity of Dijkstra's algorithm can be simplified to the following:
    # O(|V| (cost to insert + cost to delete min) + |E| (cost to decrease
```

```
key))
    #
    # For the array queue implementation, there are two main factors to
consider; the while loop that
    # essentially works as a for loop that iterates until the queue is empty,
and the delete_min function.
    # The while loop is O(n) because every node is pushed onto the queue and
the while loop
    # continues until all are removed, making it clearly O(n) time. Lastly,
delete_min for the array queue
    # is O(n) (discussed more in depth inside of the class), because it has
to iterate through every node in the array
    # to find the minimum value. Because our cost to delete min is O(n) and
our cost to insert is O(1), if we apply the
    # equation above, our complexity would be O(|V| x O(n) + (|V|+|E|) x
O(1)), (|V| is the same as n, just kept |V| to
    # match the terminology from class), our time would simplify to O(n^2) or
O(|V|^2), both are the same.
    #
    # For the heap queue implementation, there are similar factors at work,
just different time complexities for the
    # queue functions. The while loop remains the same, an O(n) factor that
is unavoidable since every node needs to
    # be processed in the algorithm. The difference comes from delete_min and
decrease_key, which are both O(log n)
    # functions. Instead of having the compounding effect of delete_min
looping through each value of the array, the
    # heap queue implementation makes big gains since both of these functions
are O(log n), making the total complexity
    # for the Dijkstra's algorithm O(n log n), since the log n functions are
being called during each loop of the
    # main while loop. Applying the same equation above to this
implementation, we get:
    # O(|V| x O(log n) + (|V|+|E|) x O(log n), which can simplify down to
O((|V| + |E|) log n) since both sides are
    # log n. Again saying that n = |V| as the array of all the
nodes/vertices, we have O((n + |E|) log n), which we
    # could further simply down to O(n log n) since |E| is a constant value
and would not change the overal complexity.
    def computeShortestPaths(self, srcIndex, use_heap=False):
        self.source = srcIndex  # Save parameter to local variable - O(1)
        t1 = time.time()

        # Choose appropriate data structure based on settings - If statement
blocks - O(1)
        if use_heap:
            queue = HeapQueue()
        else:
            queue = ArrayQueue()
            pass

        # Get all node in network and initialize dist and prev arrays
        node_array = self.network.nodes  # Save class variable to local
variable to use - O(1)
        dist = {}  # Initialize empty dictionary - O(1)

        # Iterate through each node in network and initialize dist and prev
```

```
values - O(n)
        for node in node_array:
            dist[node.node_id] = (float('inf'))  # Initialize value in array
- O(1)
            self.prev[node.node_id] = None  # Initialize value in array -
O(1)

        # Initialize starting node distance to 0
        dist[self.source] = 0  # Set value in array - O(1)

        # Call make queue on array of nodes
        queue.make_queue(node_array, dist)  # Make queue - O(n) for array,
O(n log n) for binary heap

        # While queue size is not 0, repeat Dijkstra's
        # Because the queue starts with all nodes, the while loop will repeat
n times - O(n) time complexity
        while queue.size() != 0:

            # Call delete min to pop and get node with smallest distance
            current_node = queue.delete_min(dist)  # Delete_min - Array Queue
O(n), Heap Queue O(log n)

            # Get neighboring nodes to check for updating distances
            current_neighbors = current_node.neighbors  # Get value from node
object - O(1)

            # Iterate through neighbor node edges - For loop s times where s
is number of neighbors - O(s) time.
            # The number of neighbors is at most n - 1, but this code only
creates sets of 3 neighbors, making the for
            # loop not significant time wise, and thus indicated by O(s)
where s is number of neighbors.
            for index, neighbor in enumerate(current_neighbors):

                # Get source, dest, and length of each edge for easy
reference
                src = neighbor.src  # Save value from neighbor node object -
O(1)
                dest = neighbor.dest  # Save value from neighbor node object
- O(1)
                length = neighbor.length  # Save value from neighbor node
object - O(1)

                # Current path is shorter than path previously stored in dist
array, update dist and prev
                if dist[dest.node_id] > dist[src.node_id] + length:  # If
statement comparison - O(1)
                    dist[dest.node_id] = dist[src.node_id] + length  # Update
value in array - O(1)
                    self.prev[dest.node_id] = src.node_id  # Update value in
array - O(1)
                    queue.decrease_key(dest, dist)  # decrease_key - Array
Queue N/A, Heap Queue O(log n)

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

which_pyqt.py

```
PYQT_VER = 'PYQT5'
```