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:

Table

Description automatically generated Table

Description automatically generated

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



Chart, line chart

Description automatically generated Chart

Description automatically generated with medium confidence

Chart, line chart

Description automatically generated Chart, line chart

Description automatically generated

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

A picture containing calendar

Description automatically generated

Chart, line chart

Description automatically generated

Chart

Description automatically generated

#!/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

    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):

            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.

        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

    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)