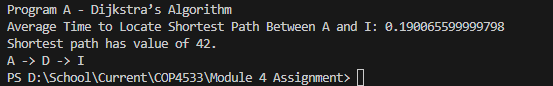
Assignment 4

Program\_a.py



This program utilizes Dijkstra’s Algorithm and locates the shortest distance between A and I. This algorithm has a time complexity of O(ElogV) where V is the number of vertices and E is the number of edges. It has a special time complexity, not like O(n) or O(logn) because this algorithm traverses through a bubble graph where nodes are connected to each other at varying distances.

Program\_b.py

I was unable to complete program\_b.py because I was unable to implement my graph class with the Bellman Ford algorithm. I completed the algorithm, but I didn’t fully understand the values it needed to be passed, and how I should change my graph class to match the algorithm. Bellman Ford’s algorithm has a time complexity of O(V\*E), where V and E are the same as in the previous algorithms time complexity. This means that this algorithm is less efficient than Dijkstra’s because O(ElogV) will rapidly scale when more vertices are introduced. However, O(V\*E)’s time complexity is faster to calculate, due to the lack of a logarithm, so it will be faster at smaller scale.

Graph.py:

class Graph(object):

def \_\_init\_\_(self, nodes, init\_graph):

self.nodes = nodes

self.graph = self.construct\_graph(nodes, init\_graph)

def construct\_graph(self, nodes, init\_graph):

# This makes sure each node can be traveled from either direction, if A to B is 10, B to A is also set to 10

graph = {}

for node in nodes:

graph[node] = {}

graph.update(init\_graph)

for node, edges in graph.items():

for adjacent\_node, value in edges.items():

if graph[adjacent\_node].get(node, False) == False:

graph[adjacent\_node][node] = value

return graph

def get\_nodes(self):

# Returns nodes in graph

return self.nodes

def get\_out\_edges(self, node):

# Returns neighbors of a specified node

connections = []

for out\_node in self.nodes:

if self.graph[node].get(out\_node, False) != False:

connections.append(out\_node)

return connections

def value(self, node1, node2):

# Returns value of bridge between nodes

return self.graph[node1][node2]

program\_a.py:

import sys

import timeit

from graph import Graph

#Program A - Algorithm

def dijkstra\_algorithm(graph, start\_node):

unv\_nodes = list(graph.get\_nodes())

min\_path = {}

prev\_n = {}

max\_value = sys.maxsize

for node in unv\_nodes:

min\_path[node] = max\_value

min\_path[start\_node] = 0

while unv\_nodes:

current\_min\_node = None

for node in unv\_nodes: # Iterate over the nodes

if current\_min\_node == None:

current\_min\_node = node

elif min\_path[node] < min\_path[current\_min\_node]:

current\_min\_node = node

# The code below retrieves the current node's neighbors and updates their distances

neighbors = graph.get\_out\_edges(current\_min\_node)

for neighbor in neighbors:

tentative\_value = min\_path[current\_min\_node] + graph.value(current\_min\_node, neighbor)

if tentative\_value < min\_path[neighbor]:

min\_path[neighbor] = tentative\_value

# Update the best path to the current node

prev\_n[neighbor] = current\_min\_node

# After visiting its neighbors, mark the node as "visited"

unv\_nodes.remove(current\_min\_node)

return prev\_n, min\_path

# Prints outputs

def print\_result(prev\_n, min\_path, start\_node, target\_node):

path = []

node = target\_node

while node != start\_node:

path.append(node)

node = prev\_n[node]

# Add the start node manually

path.append(start\_node)

print("Shortest path has value of {}.".format(min\_path[target\_node]))

print(" -> ".join(reversed(path)))

# Creating graph, adding nodes and distances

nodes = ["A", "B", "C", "D", "E", "F", "G", "H", "I"]

init\_graph = {}

for node in nodes:

init\_graph[node] = {}

init\_graph["A"]["B"] = 22

init\_graph["A"]["C"] = 9

init\_graph["A"]["D"] = 12

init\_graph["B"]["C"] = 35

init\_graph["B"]["F"] = 36

init\_graph["B"]["H"] = 34

init\_graph["C"]["F"] = 42

init\_graph["C"]["E"] = 65

init\_graph["C"]["D"] = 4

init\_graph["F"]["H"] = 24

init\_graph["F"]["G"] = 39

init\_graph["F"]["E"] = 18

init\_graph["E"]["G"] = 23

init\_graph["E"]["D"] = 33

init\_graph["D"]["I"] = 30

init\_graph["G"]["I"] = 21

init\_graph["G"]["H"] = 25

init\_graph["H"]["I"] = 19

programagraph = Graph(nodes, init\_graph)

# Testing Functionality using timeit

def program\_a\_test():

programagraph = Graph(nodes, init\_graph)

#this line runs the algorithm, these values are used for the print function but not needed in this test function

prev\_n, min\_path = dijkstra\_algorithm(graph=programagraph, start\_node="A")

# Outputs average time and the shortest path

print("Program A - Dijkstra’s Algorithm\nAverage Time to Locate Shortest Path Between A and I:", timeit.timeit(program\_a\_test, number=10000))

programagraph = Graph(nodes, init\_graph)

prev\_n, min\_path = dijkstra\_algorithm(graph=programagraph, start\_node="A")

print\_result(prev\_n, min\_path, start\_node="A", target\_node="I")

program\_b.py (incomplete):

from collections import defaultdict

from graph import Graph

#Bellman Ford Algorithm

def BellmanFord(self, src):

V=len(self.get\_nodes())

dist = [float("Inf")] \* V

dist[src] = 0

for i in range(V - 1):

for u in self.graph:

if dist[u] != float("Inf") and dist[u] + u < dist[u]:

dist[u] = dist[u] + u

for u in self.graph:

if dist[u] != float("Inf") and dist[u] + u < dist[u]:

print("Negative weight cycle")

return

nodes = ["A", "B", "C", "D", "E", "F", "G", "H", "I"]

init\_graph = {}

for node in nodes:

init\_graph[node] = {}

init\_graph["A"]["B"] = 22

init\_graph["A"]["C"] = 9

init\_graph["A"]["D"] = 12

init\_graph["B"]["C"] = 35

init\_graph["B"]["F"] = 36

init\_graph["B"]["H"] = 34

init\_graph["C"]["F"] = 42

init\_graph["C"]["E"] = 65

init\_graph["C"]["D"] = 4

init\_graph["F"]["H"] = 24

init\_graph["F"]["G"] = 39

init\_graph["F"]["E"] = 18

init\_graph["E"]["G"] = 23

init\_graph["E"]["D"] = 33

init\_graph["D"]["I"] = 30

init\_graph["G"]["I"] = 21

init\_graph["G"]["H"] = 25

init\_graph["H"]["I"] = 19

programbgraph = Graph(nodes, init\_graph)

BellmanFord(programbgraph, 0)