COMPSCI 2XC3 Final Report Part-6(Indivitual Component)

Prakhar Saxena
400451379

12 April 2024

Question

In the code posted with this document, you will find a unknown() function. It takes a graph as input. Do some reverse engineering. Try to figure out what exactly this function is accomplishing. You should explore the possibility of testing it on graphs with negative edge weights (create some small graphs manually for this). Determine the complexity of this function by running some experiments as well as inspecting the code. Given what this code does, is the complexity surprising? Why or why not?

Code Snippet

```python
import random

class DirectedWeightedGraph:

    def __init__(self):
        self.adj = {}
        self.weights = {}

    def are_connected(self, node1, node2):
        for neighbour in self.adj[node1]:
            if neighbour == node2:
                return True
        return False

    def adjacent_nodes(self, node):
        return self.adj[node]

    def add_node(self, node):
        self.adj[node] = []

    def add_edge(self, node1, node2, weight):
        if node2 not in self.adj[node1]:
            self.adj[node1].append(node2)
        self.weights[(node1, node2)] = weight

    def w(self, node1, node2):
        if self.are_connected(node1, node2):
            return self.weights[(node1, node2)]

    def number_of_nodes(self):
        return len(self.adj)

def init_d(G):
    n = G.number_of_nodes()
    d = [[float("inf") for j in range(n)] for i in range(n)]
    for i in range(n):
        for j in range(n):
            if G.are_connected(i, j):
                d[i][j] = G.w(i, j)
        d[i][i] = 0
    return d

#Assumes G represents its nodes as integers 0,1,...,(n-1)
def unknown(G):
    n = G.number_of_nodes()
    d = init_d(G)
    for k in range(n):
        for i in range(n):
            for j in range(n):
                if d[i][j] > d[i][k] + d[k][j]:
                    d[i][j] = d[i][k] + d[k][j]
    return d
```

Listing 1: Code Snippet

Solution

After analyzing the code provided and reverse-engineering it, we found out that the unknown algorithm is an algorithm that finds the shortest distance from Node A to another Node B on the same graph. It then returns the shortest distance from each node to another in a matrix form. Below we test it out on a few small graphs, including a normal graph with positive weights, a negative-weighted graph without a negative cycle, and a negative-weighted graph with a negative cycle. Then we analyze each test case.

Test Case1: Normal graph with positive weights

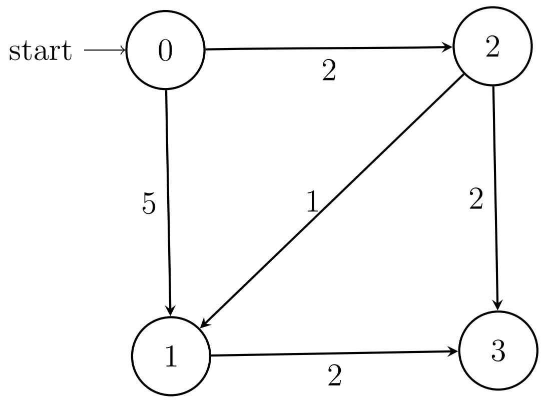

Figure 1: Positive weighted graph.

	0	1	2	3
0	0	3	2	4
1	∞	0	∞	2
2	∞	1	0	2
3	∞	∞	∞	0

Table 1: Output matrix.

The above is the output matrix that unknown algorithm gives us. The function accurately iterates through all possible paths from one node to another and determines the shortest distance. In this (column 3, row 1) reads the shortest distance from Node 0 to Node 1 is '3'.

Test Case2: Negative-weighted graph with a negative cycle

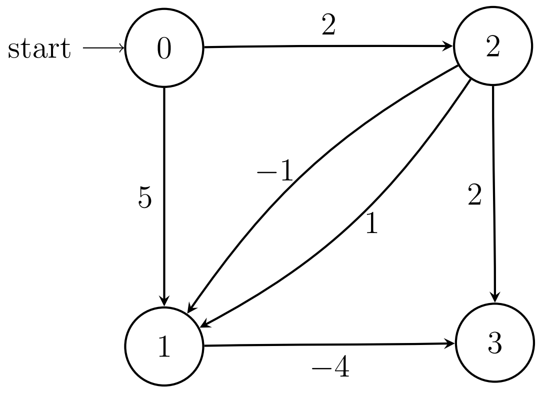

Figure 2: Negative weighted graph with a negative cycle.

	0	1	2	3
0	0	3	2	4
1	∞	0	∞	-4
2	∞	1	0	-3
3	∞	∞	∞	0

Table 2: Output matrix.

As we can clearly see in the output matrix, the unknown algorithm cannot accurately find the shortest distance for negative graphs with a negative cycle. For example, the shortest path from Node 2 to Node 3 should have been -5 instead of -3. This behavior from the algorithm was anticipated as graphs with negative cycles usually lead to infinite loops unless the algorithms are coded to take them into consideration like the Bellman-Ford algorithm. This produces incorrect answers since our code runs on O(V^3) complexity.

Test Case3: Negative-weighted graph without a negative cycle

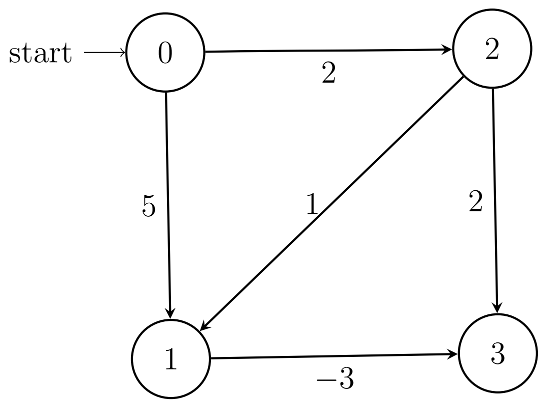

Figure 3: Negative weighted graph without a negative cycle.

We see that the output accurately gives all the shortest path from all the nodes. This shows that the algorithm takes care of all the negative nodes.

Time Complexity

The time complexity of this algorithm is O(V^3), this is because the algorithm has 3 nested for loops which each run V times where V is the number of vertices in the graph. The relatively high time complexity is understandable as the algorithm has to explore each and every path from one node to the other. The algorithm is also capable of accommodating negative weighted graphs which also impacts the high time complexity as it has to go through comprehensive checks with the negative weights. The algorithm has a brute-force way or working which essentially means that there are no break sentences which could eliminate any unnecessary work, this also leads to a higher complexity. The algorithm also uses simple data structures (lists) hence, it does not benefit from optimization which more advanced data structures might provide. All these factors add up to the high time-complexity that this algorithm has.

To understand and figure out the time complexity, I also design an experiment that first generates random graphs and then plots the run time. Firstly, we generate 150 random graphs with a range of vertices and edges to minimize the generation of similar graphs.

Generate random graphs code

```python
import random

# Random graph generator
def generate_graph(Vertex, Edge, W):
    G = DirectedWeightedGraph()
    for i in range(Vertex):
        G.add_node(i)
    max_edges = Vertex * (Vertex - 1)
    if Edge <= max_edges:
        possible_edges = [(i, j) for i in range(Vertex) for j
    in range(Vertex) if i != j]
        random_edges = random.sample(possible_edges,Edge)
        for start, end in random_edges:
            weight = random.uniform(*W)
            G.add_edge(start, end, weight)
        return G
    else:
        raise ValueError("Maximum Edges")
```

Listing 2: Generate random graphs

Plotting run time code

```python
\begin{figure}
    \centering
    \includegraphics[width=0.5\linewidth]{image.png}
    \caption{Enter Caption}
    \label{fig:enter-label}
\end{figure}
import timeit
import random
import matplotlib.pyplot as plt

def experiment():
    n_graphs = 150
    tests = {}
    weight_range = (0, 20)

    for _ in range(n_graphs):
        vertex_count = random.randint(2, 250)
        min_edges = vertex_count - 1
        max_edges = vertex_count * (vertex_count - 1)
        edge_count = random.randint(min_edges, max_edges)
        graph = generate_graph(vertex_count, edge_count, weight_range)

        if graph.number_of_nodes() not in tests:
            tests[graph.number_of_nodes()] = graph

    x_values = sorted(tests.keys())
    y_values = []

    for size in x_values:
        graph = tests[size]

        start_time = timeit.default_timer()
        unknown(graph)
        end_time = timeit.default_timer()
        y_values.append(end_time - start_time)

    # Plot
    plt.plot(x_values, y_values, label='Algorithm Run Time')
    plt.xlabel('Number of Vertices')
    plt.ylabel('Run Time (seconds)')
    plt.title('Algorithm Run Time with Increasing Graph Size')
    plt.legend()
    plt.show()

experiment()
```

Listing 3: Plotting run time.

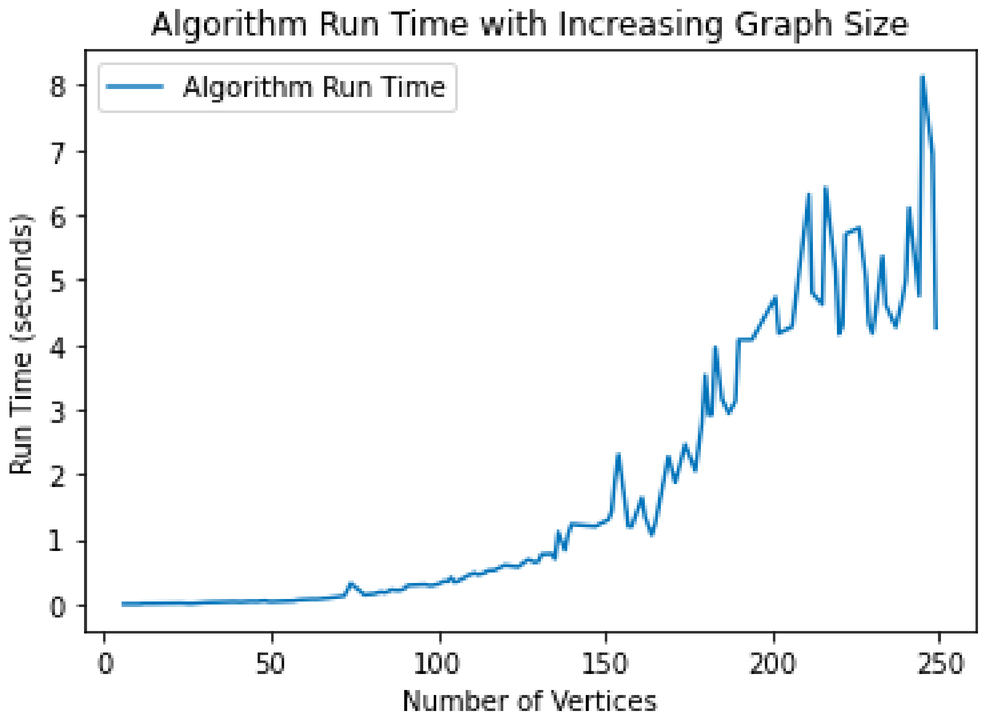

Figure 4: Run time.

From the above line graph we can clearly conclude that as the number of vertices in a graph increases, the time taken by the algorithm increases. This supports the fact that the algorithm's time complexity is O(V^3).