CS-4710 & CS1217: Operating Systems, Spring 2025, Ashoka University

Assignment Code	CS_4710_A1 / CS_1217_A1
Topic:	Floyd-Warshall Algorithm using Multithreading
Due Date:	March 28, 2025
Total Marks:	32
Weightage:	8%

Submission Instructions:

- You have to submit both a PDF containing answers and explanations for the implementation, as well as a single file containing your code.
- Submissions must follow the format **Firstname_Lastname_A1** for both the PDF and code file(s).
- Please use C or C++ to solve the assignment. (Python or any other programming language is not allowed.)
- Ensure that your code compiles successfully; otherwise, no credit will be given.

What you'll submit:

1. One or more .c or .cpp file(s) containing your implementation, as well as a makefile. [17]
2. A PDF containing explanations for how you've deployed multi-threading as well as pseudocode for your multi-threaded approach. [15]
3. **BONUS:** The Floyd-Warshall Algorithm works better on adjacency matrices than adjacency lists. Why? Perform a runtime analysis of the algorithm and compare it with the multi-threaded approach. [8]

Multi-threading:

In this assignment, you are required to implement a multi-threaded version of the **Floyd-Warshall All-Pairs-Shortest-Path** algorithm. Since multi-threading is commonly used in graph algorithms, your task is to design your own approach and provide pseudocode to explain your implementation.

Here's the single threaded version of the algorithm. The single-threaded version of the algorithm follows a three-level nested loop structure. The outermost loop iterates over an intermediate node k, considering paths that pass through it. The two inner loops iterate over all node pairs (i, j), checking whether using k as an intermediate node offers a shorter path. The `dist[i][j]` matrix keeps track of the shortest known distances between node pairs so far, and each iteration updates these values. A key observation here is that in each iteration, we only need to read from the k^{th} row and column while updating the same `dist` matrix, without requiring additional copies.

```
for k=1 to n do
    for i = 1 to n do
        for j=1 to n do
                                        if (dist[i][k] + dist[k][j] < dist[i][j])
                                        dist[i][j] = dist[i][k] + dist[k][j]
            end for
    end for
end for
```

We modify this algorithm to take advantage of multi-threading. The outer k loop remains unchanged, but instead of sequentially iterating over i, we create n threads, where each thread is responsible for computing the updates for a specific row i. Each thread independently executes the inner j loop, computing updates in parallel.

Since multiple threads access the `dist` matrix simultaneously, you may need to handle race conditions correctly if they arise.

At the end of every iteration of the k loop, you need to wait for all the threads to finish. Only then can you start another iteration of the k loop. Therefore, you need to join the threads at the end of every iteration of the k loop.

Input Format:

The graph structure is given as follows:

The first line contains two integers, N and M, where N is the number of nodes and M is the number of undirected edges. The next M lines each contain three integers: u_i, v_i, and w_i, representing an undirected edge between nodes u_i and v_i with weight w_i.

Constraints:

- $N \in [1, 100]$
- All edge weights are positive.

Output Format:

Print the final `dist` matrix. If node j is not reachable from node i, print `INF` as the distance between i and j.

Example:

Input:

```
4 4
1 2 1
2 3 1
3 4 1
4 1 1
```

Output:

```
0 1 2 1
1 0 1 2
2 1 0 1
1 2 1 0
```