Name : Sharvan Kotharu

Roll No:43

Sec : A1_B3

DAA- TA

TOPIC : BELLMAN FORD ALGORITHM

Q1. Detecting Negative Weight Cycle

Problem Statement

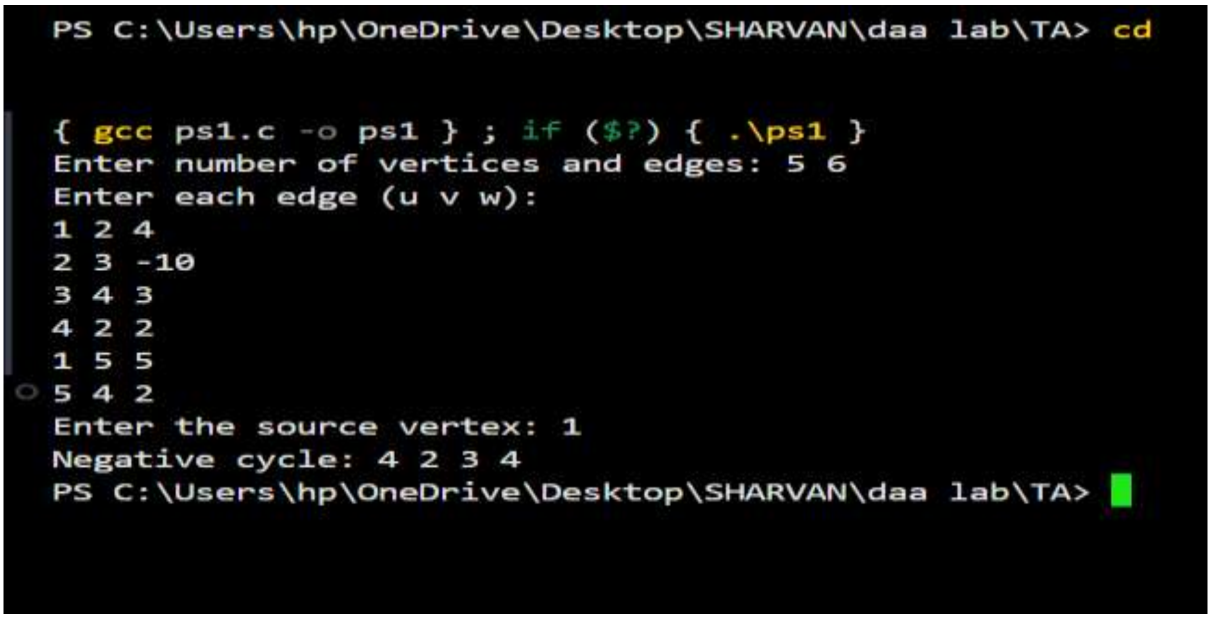
The Bellman-Ford algorithm is used to find the shortest paths from a single source to all other ertices in a weighted graph, even if edges have negative weights. However, if a graph contains a negative weight cycle (a cycle whose total edge weights sum to less than 0), then the concept of shortest paths breaks down, since the cost can be decreased indefinitely by looping through that cycle.In this problem, you are asked to:

1. Apply the Bellman-Ford algorithm to detect if a negative cycle exists.

2. If a cycle is found, reconstruct and output the cycle (sequence of vertices).

3. Otherwise, state that no negative cycle exists.

OUTPUT[CODE IS ATTATCHED] :



```
PS C:\Users\hp\OneDrive\Desktop\SHARVAN\daa lab\TA> cd


{ gcc ps1.c -o ps1 } ; if ($?) { .\ps1 }
Enter number of vertices and edges: 5 6
Enter each edge (u v w):
1 2 4
2 3 -10
3 4 3
4 2 2
1 5 5
5 4 2
Enter the source vertex: 1
Negative cycle: 4 2 3 4
PS C:\Users\hp\OneDrive\Desktop\SHARVAN\daa lab\TA>
```

APPROACH USED:

The program uses the **Bellman–Ford algorithm** to compute shortest paths from a source vertex in a weighted directed graph, even when some edges have negative weights. It works by **relaxing all edges repeatedly (V-1 times)**, updating the distance to each vertex if a shorter path is found. After relaxation, the algorithm checks for **negative weight cycles** by seeing if any edge can still reduce a distance; if so, a cycle exists. For cycle detection, the program traces back through the parent pointers to reconstruct the negative cycle. This approach is efficient for **sparse graphs** and guarantees detection of negative cycles while computing shortest paths.

Q2. Shortest Path in Currency Exchange Graph

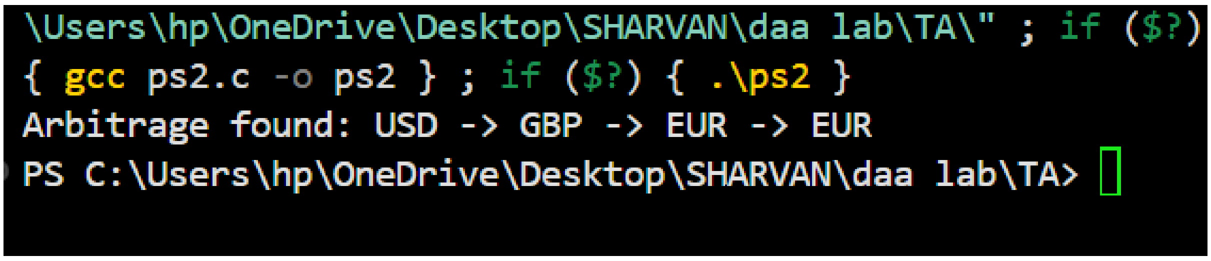
(Arbitrage Detection)

Problem in Easy Words

Imagine you are trading currencies like USD, EUR, GBP.

Normally, if you convert money around and come back to the same currency, you should end up with the same amount (or less, because of fees). But sometimes, due to differences in exchange rates, you can end up with more money than you started. This is called an arbitrage opportunity. The task:Represent the exchange market as a graph. Use the Bellman–Ford algorithm to detect whether such a profit-making cycle exists.

OUTPUT[CODE IS ATTATCHED] :



```
\Users\hp\OneDrive\Desktop\SHARVAN\daa lab\TA\" ; if ($?)
{ gcc ps2.c -o ps2 } ; if ($?) { .\ps2 }
Arbitrage found: USD -> GBP -> EUR -> EUR
PS C:\Users\hp\OneDrive\Desktop\SHARVAN\daa lab\TA>
```

APPROACH USED

The program models currencies as nodes and exchange rates as directed edges in a graph, converting each rate into a weight using **-log(rate)** so that multiplying rates becomes additive. It then applies the **Bellman–Ford algorithm**, which relaxes all edges repeatedly to find the shortest paths. After V-1 iterations, if any edge can still be relaxed, a **negative cycle** exists, indicating an arbitrage opportunity. The program traces back through parent pointers to reconstruct and print the cycle. This method efficiently detects profit-making cycles in currency exchange networks

Q3. Optimizing Data Packet Routing in a Network

Think of a computer network as a graph:

Routers are nodes (vertices).Links (cables/wireless connections) are directed edges with a weight = time delay (in milliseconds). We want to find the fastest way (minimum delay) to send a data packet from one router (source) to all others.The Bellman–Ford algorithm can do this, even if some links have negative weights (which can represent things like special accelerations or priority adjustments). But Bellman–Ford can be slow if the network is large.

o If the graph is sparse (few connections), it's better to use an adjacency list.

o If the graph is dense (almost every router connected to every other), an

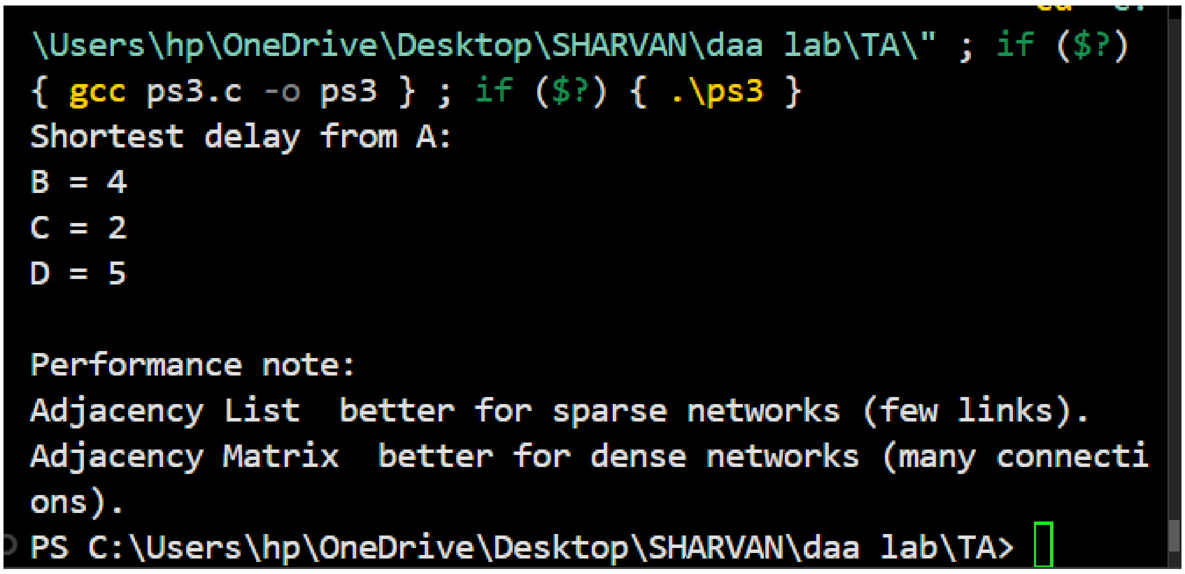
adjacency matrix may be better.

The problem asks you to:

1. Use Bellman–Ford to compute the shortest time from a source router to all others.

2. Compare the performance of adjacency list vs adjacency matrix implementations.

OUTPUT



```
\Users\hp\OneDrive\Desktop\SHARVAN\daa lab\TA\" ; if ($?)
{ gcc ps3.c -o ps3 } ; if ($?) { .\ps3 }
Shortest delay from A:
B = 4
C = 2
D = 5

Performance note:
Adjacency List  better for sparse networks (few links).
Adjacency Matrix  better for dense networks (many connecti
ons).
PS C:\Users\hp\OneDrive\Desktop\SHARVAN\daa lab\TA>
```

APPROACH USED:

The program models a computer network as a **directed weighted graph**, with routers as nodes and links as edges representing time delays. It uses the **Bellman–Ford algorithm** to compute the shortest delay from a source router to all others, even when some edges have negative weights (like priority paths). The algorithm initializes distances, then repeatedly **relaxes all edges V-1 times** to find the minimum delays. After relaxation, it checks for **negative weight cycles**, which indicate inconsistent or invalid paths. Finally, it prints the shortest delays and notes that **adjacency lists** are faster for sparse networks, while **adjacency matrices** perform better for dense networks.