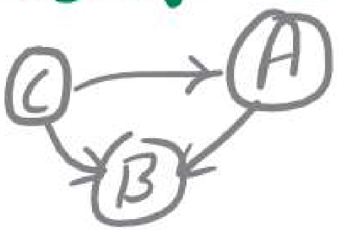
FLOYD WARSHALL ALGORITHM (GRAPH)

MSSP - Multi Source SHORTEST PATH



Linkedin@ manojofficialmj

3. Floyd War-shall Algorithm

MSSP: Multiple Source Shortest Path

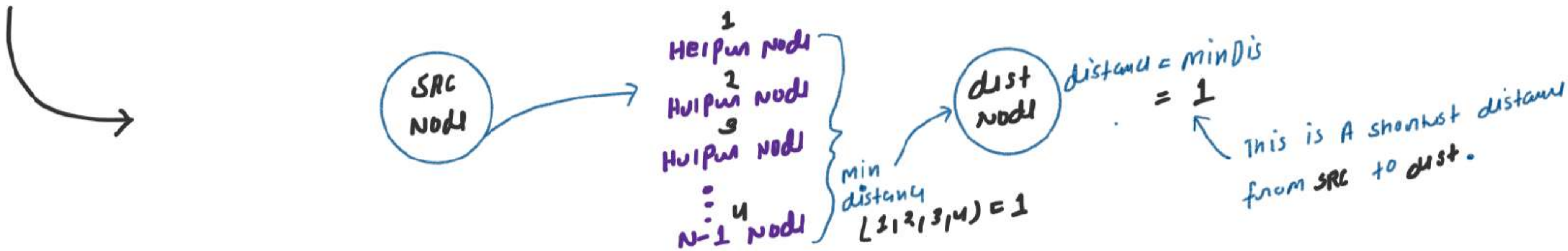
What is Floyd War-shall algorithm:
*Basically, the Floyd War-shall algorithm is a **multi-source shortest path algorithm** and it helps to **detect negative cycles** as well.*

Note: Dijkstra's** Shortest Path algorithm and **Bellman-Ford** algorithm are **single-source shortest path algorithms.

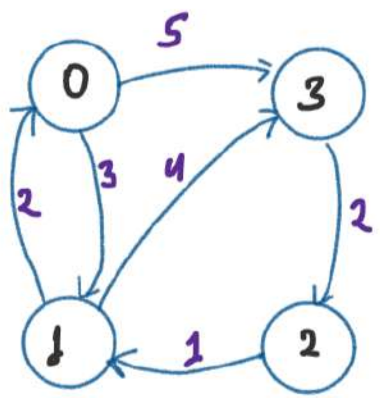
Where to use Floyd War-shall algorithm:
*Floyd War-shall algorithm can be used to **find the shortest paths between all pairs of vertices** in a directed weighted graph.*
It can also be used to find the shortest cycle in both directed and undirected graphs.

***Note:** It also doesn't work for graphs with **negative cycles,** where the sum of the edges in a cycle is negative.*

Working flow of Floyd War-shall algorithm:
The algorithm works by checking every possible path between every possible node, and then choosing the shortest one.



Example



Output

Dest
Destination

Source
SRC

Dist	0	1	2	3
0	0	3	7	5
1	2	0	6	4
2	3	1	0	5
3	5	3	2	0

mini Distance

NxN

No. of Nodes

Explanation

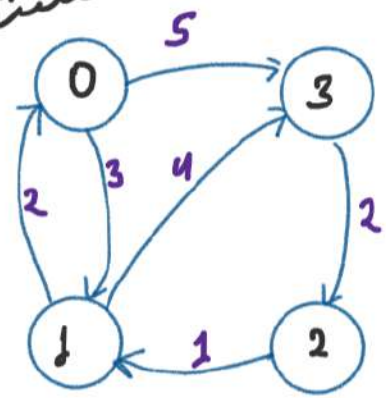
From SRC	To Dest	minimum Distance
0	0	0
	1	3
	2	7
	3	5

SRC	Dest	minimum Distance
1	0	2
	1	0
	2	6
	3	4

SRC	Dest	minimum Distance
2	0	3
	1	1
	2	0
	3	5

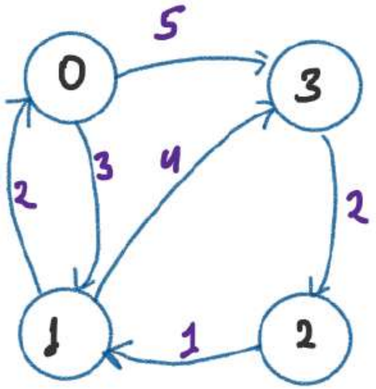
SRC	Dest	minimum Distance
3	0	5
	1	3
	2	2
	3	0

GRAPH



MSSP - mutiple source shortest path

Logic



STEP1

Initial state

DST Destination

Source SRC

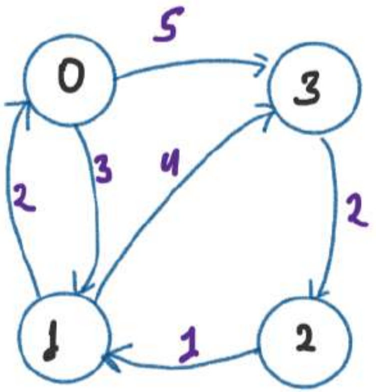
Dist	0	1	2	3
0	∞	∞	∞	∞
1	∞	∞	∞	∞
2	∞	∞	∞	∞
3	∞	∞	∞	∞

mini Distance

NXN

No. of Nodes

DRY RUN



STEP2

Fill Diagonal cell with 0 Distance

Dist Destination

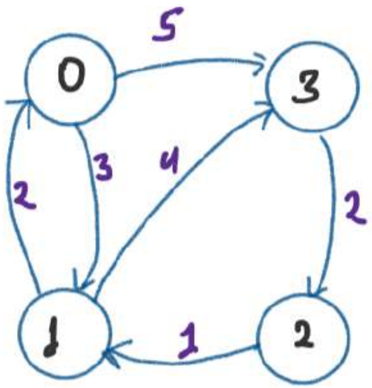
Source SRC

Dist	0	1	2	3
0	0	∞	∞	∞
1	∞	0	∞	∞
2	∞	∞	0	∞
3	∞	∞	∞	0

mini Distance

NxN

No. of nodes



AdjList

$0 \xrightarrow{5} 3$

$0 \xrightarrow{3} 1$

$3 \xrightarrow{2} 2$

$2 \xrightarrow{1} 1$

$1 \xrightarrow{4} 3$

$1 \xrightarrow{2} 0$

STEP 3

Fill given weight of graph from Each one node to other node

so go to AdjList to do this job

Dist (Destination)

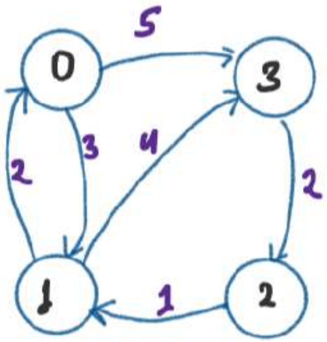
Source SRC

Dist	0	1	2	3
0	0	3	∞	5
1	2	0	∞	4
2	∞	1	0	∞
3	∞	∞	2	0

mini Distance

NXN

No. of nodes



STEP 4 MAIN Logic

Dist
Dest
Destination

Dist	0	1	2	3
0	0	3	∞	5
1	2	0	∞	4
2	∞	1	0	∞
3	∞	∞	2	0

mini Distance

Source
SRC

NxN

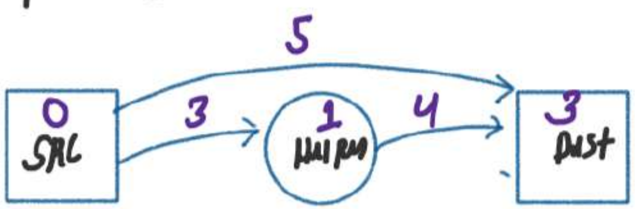
no. of nodes

meaning

FORMULA Dist[SRC][Dest] =

min[Dist[SRC][Dest],
Dist[SRC][Helper] + Dist[Helper][dest]]

I want to find shortest path from $\boxed{0}$ to Dest $\boxed{3}$ $\Rightarrow$ 5



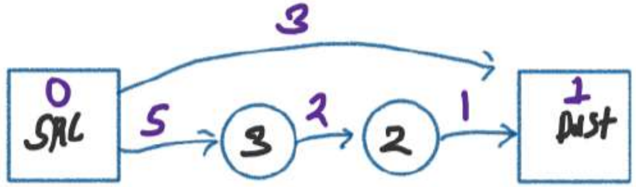
When Helper = 1

Dist[0][3] = min[Dist[0][3] , Dist[0][1] + Dist[1][3]]

$\Rightarrow$ min [5 , 3+4]

$\Rightarrow$ 5

I want to find shortest path from $\boxed{0}$ to Dest $\boxed{1}$ $\Rightarrow$



min (8, 3) = 3

When Helper = 2

D[0][1] = min [D[0][1] , D[0][2] + D[2][1]]

3   ∞ + 1

= 3

When Helper = 3

D[0][1] = min [D[0][1] , D[0][3] + D[3][1]]

3   5 + ∞

= 3

∞ Jab use कर रहे हैं। जब एक Node Dusre Node से directly connect Nahi Hai.

```
Helper: 0        Helper: 1        Helper: 2        Helper: 3        Printing distance array
0, 3, ∞, 5,      0, 3, ∞, 5,      0, 3, ∞, 5,      0, 3, 7, 5,      0 3 7 5
2, 0, ∞, 4,      2, 0, ∞, 4,      2, 0, ∞, 4,      2, 0, 6, 4,      2 0 6 4
∞, 1, 0, ∞,      3, 1, 0, 5,      3, 1, 0, 5,      3, 1, 0, 5,      3 1 0 5
∞, ∞, 2, 0,      ∞, ∞, 2, 0,      5, 3, 2, 0,      5, 3, 2, 0,      5 3 2 0
```

Why use "1e9" instead of "INT_MAX":

Using 1e9 ensures that we are within the safe range of integer values and avoids potential overflow problems like

2147483647

- `INT_MAX + 5; // Risk of overflow`
- `1e9 + 5; // No risk of overflow`

1000000000

```cpp
// 3. Floyd Warshall Algorithm

#include<iostream>
#include<vector>
#include<unordered_map>
#include<limits.h>
#include<list>

using namespace std;

class Graph
{
    public:
        unordered_map<int, list<pair<int, int>>> adjList;

        void addEdges(int u, int v, int wt, int direction){
            if(direction == 1){
                // Directed Graph
                adjList[u].push_back({v,wt});
            }
            else{
                // Undirected Graph
                adjList[u].push_back({v,wt});
                adjList[v].push_back({u,wt});
            }
        }

        void floydWarshal(int n){
            ...
        }
};

int main(){
    Graph g;
    g.addEdges(0,1,3,1);
    g.addEdges(0,3,5,1);
    g.addEdges(3,2,2,1);
    g.addEdges(2,1,1,1);
    g.addEdges(1,3,4,1);
    g.addEdges(1,0,2,1);

    int n = 4;
    g.floydWarshal(n);
    return 0;
}
```

```cpp
void floydWarshal(int n){
    // Step 1: initial state
    vector<vector<int>> dist(n, vector<int>(n, 1e9));

    // Step 2: fill diagonal cell with 0 distance from src to src
    for(int i=0; i<n; i++){
        dist[i][i] = 0;
    }

    // Step 3: goto adjList to fill distance cell with the given weight of graph from u to v
    for(auto a: adjList){
        for(auto b: a.second){
            int u = a.first;
            int v = b.first;
            int wt = b.second;
            dist[u][v] = wt;
        }
    }

    // Step 4: main logic -> helper node (Helper node is intermediate node between source and destination)
    for(int helper = 0; helper < n; helper++){
        for(int src = 0; src < n; src++){
            for(int dest = 0; dest < n; dest++){
                dist[src][dest] = min(dist[src][dest], dist[src][helper]+dist[helper][dest]);
            }
        }
    }

    // Printing the distance array
    for(int i=0; i<n; i++){
        for(int j=0; j<n; j++){
            cout<<dist[i][j]<<" ";
        }
        cout << endl;
    }
}
```

T.C. = $O(V)^3$, V is no. of nodes/vertices

S.C. = $O(V^2)$