CS211FZ: Data Structures and Algorithms II

11- Dynamic Programming

Floyd Warshall Algorithm

LECTURER: MAHWISH KUNDI & IRFAN AHMAD

MAHWISH.KUNDI@MU.IE , IRFAN.AHMAD@MU.IE

Background

Dynamic Programming Vs Greedy Algorithm

- Both are employed to find best solution among various possibilities.

Dynamic programing

- Divides the complex problem into small subproblems.
- Make a choice at each step.
- Choice depends on knowing optimal solutions to subproblems. Solve subproblems first.
- Solve bottom-up. More optimal but slower.

Greedy algorithms

- Divides the complex problem into small subproblems.
- Make a choice at each step.
- Make the choice before solving the subproblems.
- Solve top-down. Less optimal but quicker.

Examples of Greedy Algorithms

- Prim's Algorithm: Finds the minimum spanning tree of a weighted graph. (already covered.)

- Huffman Coding: Used in data compression that assigns variable-length codes to input characters, with shorter codes for more frequent characters. (already covered.)

- Dijkstra's Algorithm: Used to find the shortest path from a starting node to all other nodes in a weighted graph. It cannot handle negative edge weights. (Topic for next class)

Examples of Dynamic Programming Algorithms

- Knapsack Problem: Aimed at selecting items with given weights and values to maximize the total value without exceeding a weight limit. (already covered.)

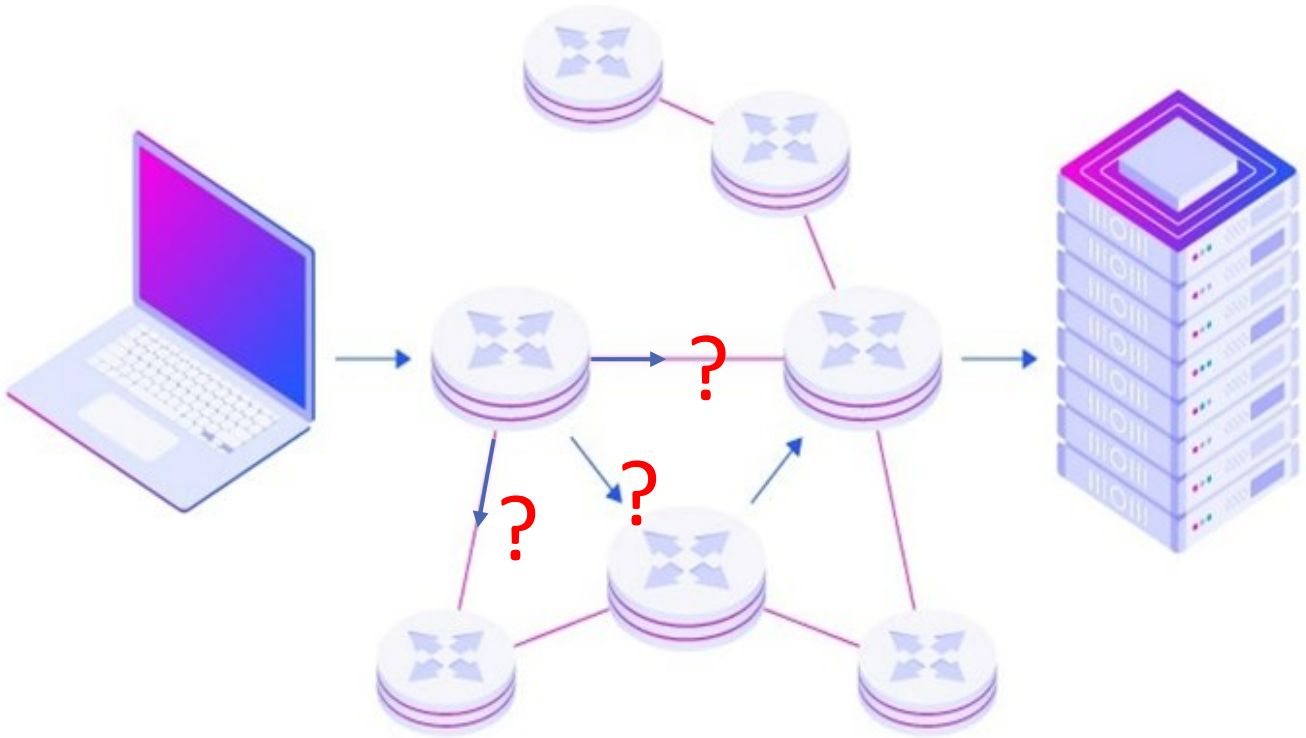
- Floyd-Warshall Algorithm: Computes the shortest paths between all pairs of nodes in a graph. (Topic for today.)

- Bellman-Ford Algorithm: Computes the shortest paths from a single source node to all other nodes in a graph. It can handle negative edge weights

Floyd-Warshall Algorithm

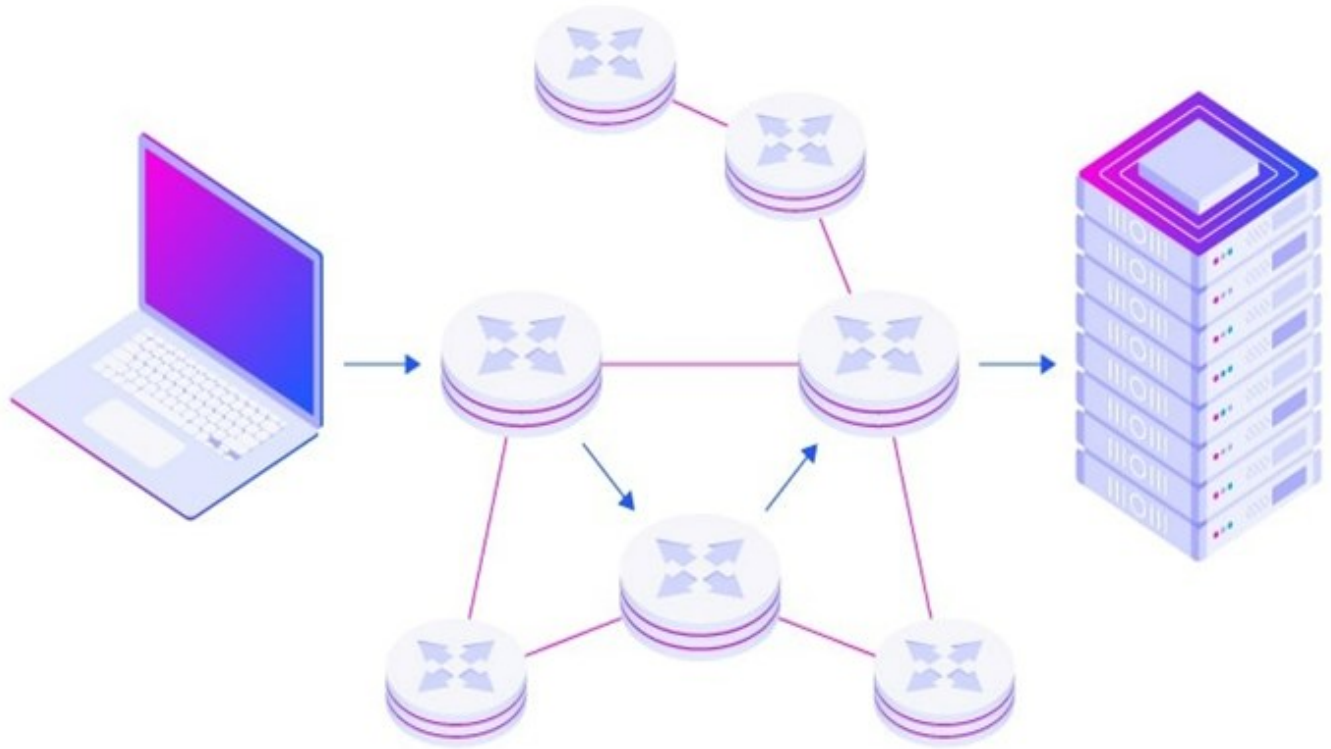
Problem 1 – Network Routing

- In large networks, finding the shortest path for data packets to travel between routers can be challenging due to multiple potential paths and varying network traffic.



Solution – Floyd-Warshall Algorithm

- Floyd-Warshall algorithm is used to precompute the shortest paths between all pairs of routers, ensuring efficient data routing and reducing latency.

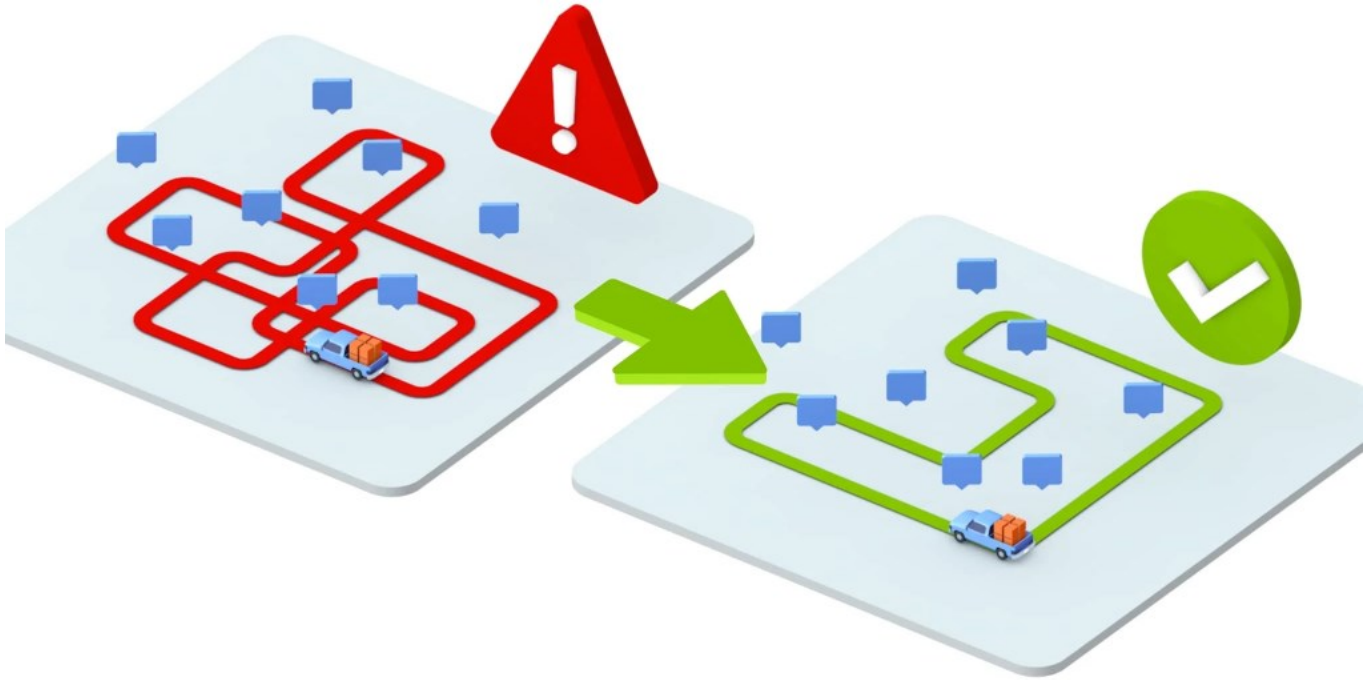


Problem 2 – Transportation and Logistics

- Determining the shortest routes for delivery trucks to minimize travel time and fuel consumption is one of the main concern for transportation and logistics companies.

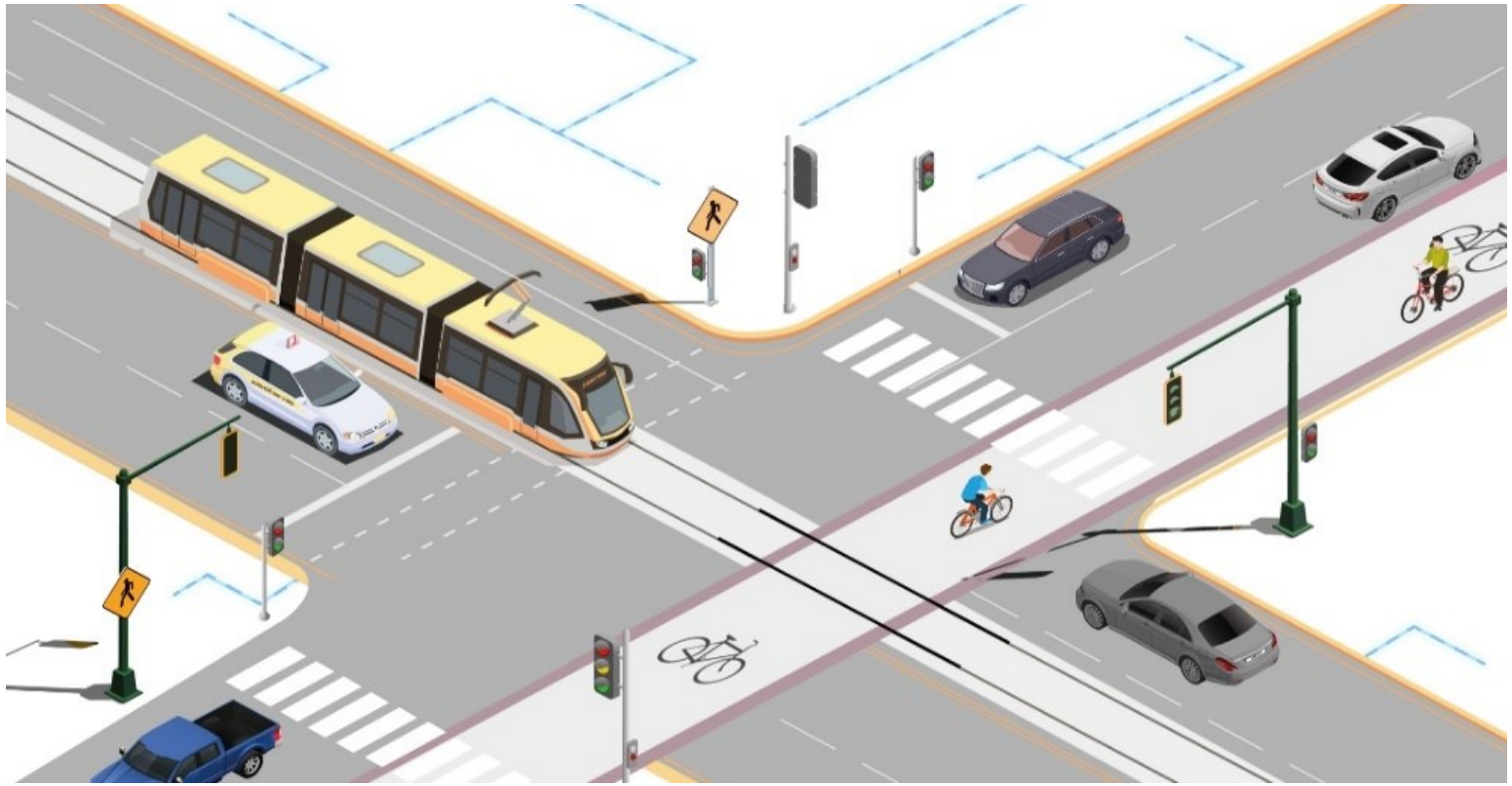
Solution – Floyd-Warshall Algorithm

- Floyd-Warshall algorithm helps in determining the most efficient routes for goods, reducing transit times and improving overall supply chain efficiency.



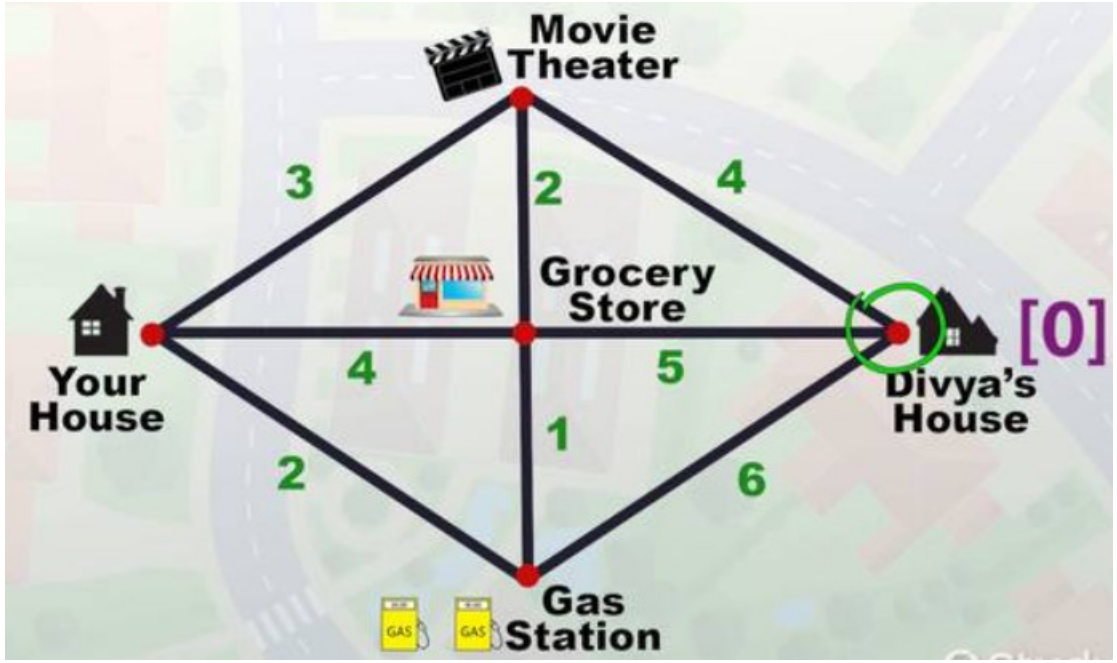
Problem 3 – Traffic management

- Managing and optimizing traffic flow in a city's transportation network to reduce congestion and travel time.



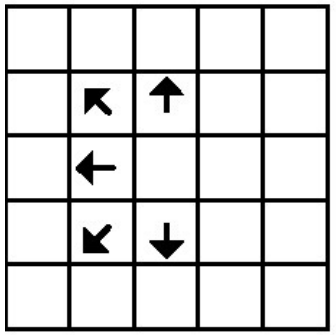
Solution - Floyd-Warshall Algorithm

- Urban planners and traffic engineers use Floyd-Warshall to analyze traffic flow and optimize signal timings at intersections.
- By finding the shortest paths between different points in a city, they can develop strategies to alleviate congestion and improve overall traffic efficiency.

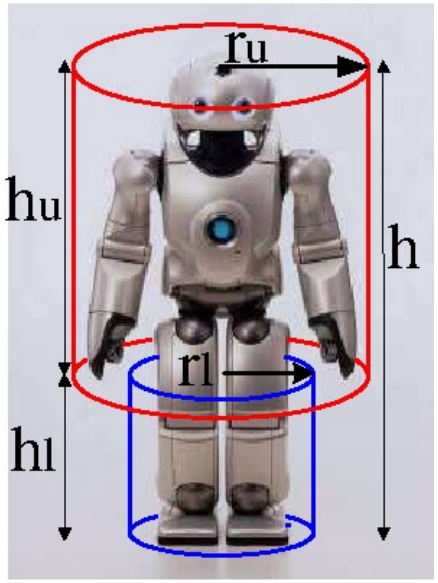


Floyd-Warshall Algorithm – Application to Robotics

- Path planning algorithms in robotics often utilize the Floyd-Warshall algorithm to find the shortest paths between various points in a robot's environment.
- This is crucial for autonomous robots navigating complex terrains or environments with obstacles.



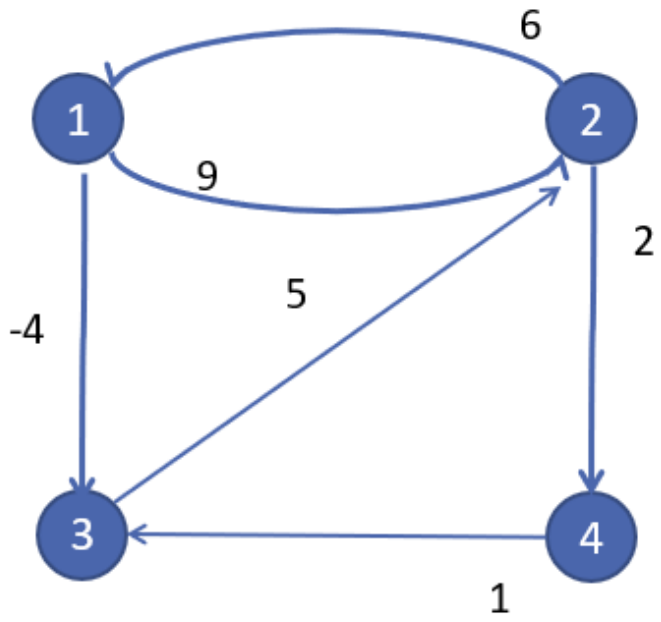
(a)



(b)

Floyd-Warshall Algorithm – Working

- Let the given graph be:



- How to find the shortest path between all the pairs of vertices?

Floyd-Warshall Algorithm – Working

Step 1: Create a distance matrix A^0

- Initialize Matrix A^0
  - Create a distance matrix A^0 of dimension $n \times n$ where n is the number of vertices.
  - Index the rows and columns as i and j respectively.

- Matrix values.
  - Distance to itself: If i=j, A^0[i,j] = 0 (distance to itself is zero).
  - Direct Edge: If there is an edge from i to j, A^0[i,j] = w_{ij} (weight of the edge).
  - No direct Edge: If there is no edge from i to j, A^0[i,j] = ∞.

Floyd-Warshall Algorithm – Working

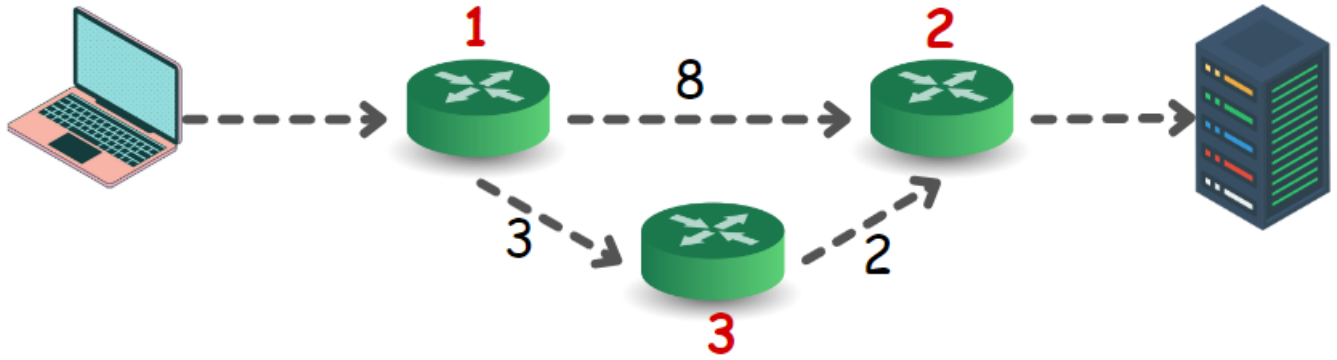
Step 2: Derive a next distance matrix A^k

- Derive another distance matrix $A^k = A^1$ from $A^{k-1} = A^0$.
  - Vertex k = 1 to n is intermediate vertex used to check if a shorter path exists through this intermediate vertex (if there is no direct short path between i and j).
- Here we use the first vertex acts as the intermediate vertex so k = 1.
  - Copy the kth row and the kth column (k=1 means first row and first column) from A^0 to A^1 without any changes.
  - Copy the diagonal elements without any changes.
  - For the remaining values, say A^k[i,j] = A^1[i,j] do the following:
    - If A^0[i,j]> A^0[i,k]+ A^0[k,j] then replace A^1[i,j] with A^0[i,k]+ A^0[k,j].
    - Otherwise, do not change the values.

Floyd-Warshall Algorithm – Working

Step 2: Derive a next distance matrix A^k

- To understand the logic, see the diagram below



A^0[i,j]> A^0[i,k]+ A^0[k,j]
A^0[1,2]> A^0[1,3]+ A^0[3,2]
8 > 3 + 2
8 > 5

Direct distance is more as compared to distance via intermediate vertex k=3, so we take the path through intermediate vertex k.

Therefore, in new matrix A^1 we should replace A^1[i,j] with A^0[i,k]+ A^0[k,j]
Because distance 5 is short in comparison to 8.

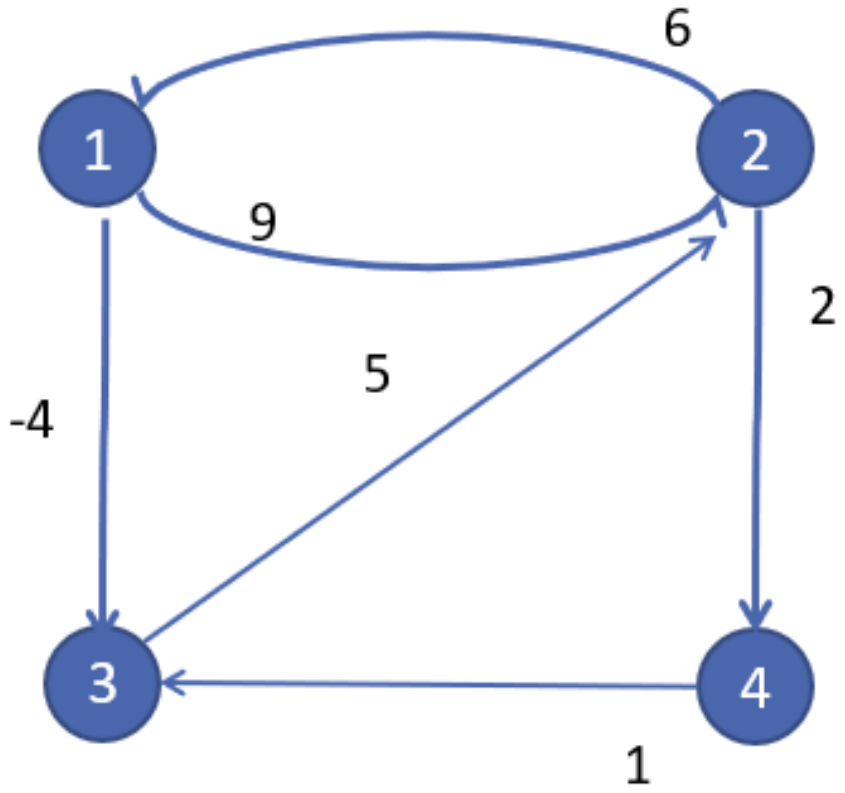
Floyd-Warshall Algorithm – Working

- **Step 3** – Repeat Step 2 for all the vertices in the graph by changing the k value for every intermediate vertex until the final matrix is achieved.

- **Step 4** – The final distance matrix obtained is the final solution with all the shortest paths.

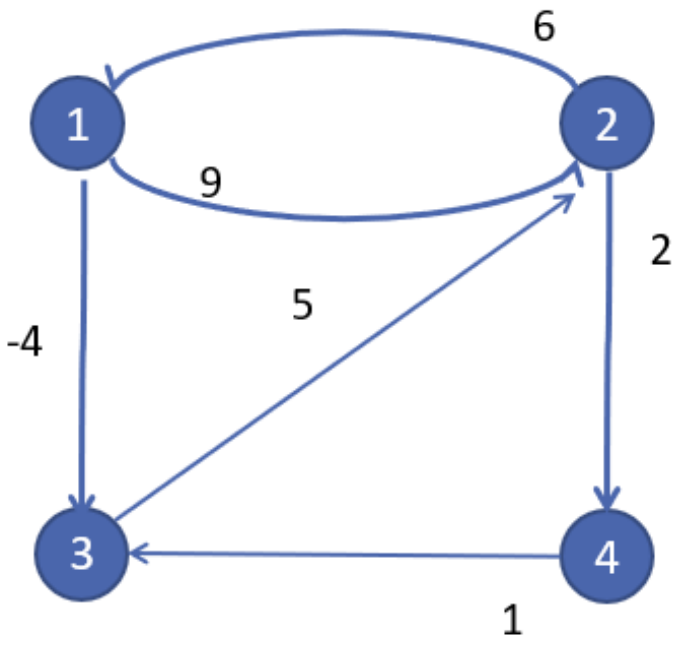
Floyd-Warshall Algorithm – Example

- Let's find the shortest path between all the pairs of vertices using Floyd-Warshall algorithm.



Step 1: Create A^0

- Create a distance matrix A^0 of dimension $n \times n$ where n is the number of vertices.

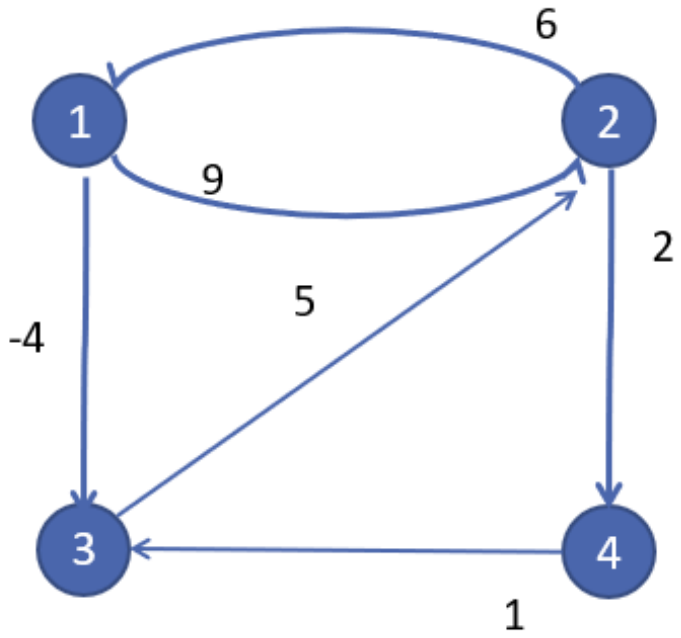


$A^0 =$

	1	2	3	4
1	-	-	-	-
2	-	-	-	-
3	-	-	-	-
4	-	-	-	-

Step 1: Create A^0

- Fill Matrix values as :
  - Distance to itself: If i=j, A^0[i,j] = 0 (distance to itself is zero).
  - Direct Edge: If there is an edge from i to j, A^0[i,j] = w_{ij} (weight of the edge).
  - No direct Edge: If there is no edge from i to j, A^0[i,j] = ∞.



A^0=

	1	2	3	4
1	0	9	-4	∞
2	6	0	∞	2
3	∞	5	0	∞
4	∞	∞	1	0

Step 2: Derive A^1 from A^0

- Here we use the first vertex k =1 acts as the intermediate vertex i.e., to check is there any better shortest path via vertex 1.
  - Copy the first row and first column from A^0 to A^1 without any changes.
  - Copy the diagonal elements without any changes.

A^0=

	1	2	3	4
1	0	9	-4	∞
2	6	0	∞	2
3	∞	5	0	∞
4	∞	∞	1	0

A^1=

	1	2	3	4
1	0	9	-4	∞
2	6	0	-	-
3	∞	-	0	-
4	∞	-	-	0

Step 2: Derive A^1 from A^0

- Now, we need to find the values of $A^1[2,3]$, $A^1[2,4]$, $A^1[3,2]$, $A^1[3,4]$, $A^1[4,2]$, and $A^1[4,3]$.

$A^0=$

	1	2	3	4
1	0	9	-4	∞
2	6	0	∞	2
3	∞	5	0	∞
4	∞	∞	1	0

$A^1=$

	1	2	3	4
1	0	9	-4	∞
2	6	0	-	-
3	∞	-	0	-
4	∞	-	-	0

Step 2: Derive A^1 from A^0

- To find $A^1[2,3]$ Check the following condition:
  - If $A^0[2,3] > A^0[2,1] + A^0[1,3]$ then replace $A^1[2,3]$ with $A^0[2,1] + A^0[1,3]$.
  - Otherwise, do not change the values.

$A^0 =$

	1	2	3	4
1	0	9	-4	∞
2	6	0	∞	2
3	∞	5	0	∞
4	∞	∞	1	0

$A^1 =$

	1	2	3	4
1	0	9	-4	∞
2	6	0	-	-
3	∞	-	0	-
4	∞	-	-	0

Step 2: Derive A^1 from A^0

- To find $A^1[2,3]$ Check:

$A^0[2,3] > A^0[2,1] + A^0[1,3]$

$\infty > 6 - 4$

$\infty > 2$

Since infinity is greater than 2, so we replace $A^1[2,3]$ with 2 (shorter distance).

$A^0 =$

	1	2	3	4
1	0	9	-4	∞
2	6	0	∞	2
3	∞	5	0	∞
4	∞	∞	1	0

$A^1 =$

	1	2	3	4
1	0	9	-4	∞
2	6	0	2	-
3	∞	-	0	-
4	∞	-	-	0

Step 2: Derive A^1 from A^0

- To find $A^1[2,4]$ Check:

  $A^0[2,4] > A^0[2,1] + A^0[1,4]$

  $2 > 6 + \infty$

  $2 > \infty$

Since 2 is not greater than infinity, so we keep the original value of $A^0[2,4]$ for $A^1[2,4]$ which is 2.

In other words, the direct distance between vertex 2 and 4 is short in comparison to via intermediate vertex 1

$A^0 =$

	1	2	3	4
1	0	9	-4	∞
2	6	0	∞	2
3	∞	5	0	∞
4	∞	∞	1	0

$A^1 =$

	1	2	3	4
1	0	9	-4	∞
2	6	0	2	2
3	∞	-	0	-
4	∞	-	-	0

Step 2: Derive A^1 from A^0

- To find $A^1[3,2]$ Check:

$A^0[3,2] > A^0[3,1] + A^0[1,2]$

$5 > \infty + 9$

$5 > \infty$

Since 5 is not greater than infinity, so we keep the original value of $A^0[3,2]$ for $A^1[3,2]$ which is 5.

$A^0 =$

	1	2	3	4
1	0	9	-4	∞
2	6	0	∞	2
3	∞	5	0	∞
4	∞	∞	1	0

$A^1 =$

	1	2	3	4
1	0	9	-4	∞
2	6	0	2	2
3	∞	5	0	-
4	∞	-	-	0

Step 2: Derive A^1 from A^0

- To find $A^1[3,4]$ Check:

$A^0[3,4] > A^0[3,1] + A^0[1,4]$

$\infty \quad > \quad \infty \quad + \quad \infty$

$\infty \quad > \infty$

Since infinity is not greater than infinity, so we keep the original value of $A^0[3,4]$ for $A^1[3,4]$ which is ∞.

$A^0 =$

	1	2	3	4
1	0	9	-4	∞
2	6	0	∞	2
3	∞	5	0	∞
4	∞	∞	1	0

$A^1 =$

	1	2	3	4
1	0	9	-4	∞
2	6	0	2	2
3	∞	5	0	∞
4	∞	-	-	0

Step 2: Derive A^1 from A^0

- To find $A^1[4,2]$ Check:

$A^0[4,2] > A^0[4,1] + A^0[1,2]$

$\infty \quad > \quad \infty \quad + \quad 9$

$\infty \quad > \quad \infty$

$A^0=$

	1	2	3	4
1	0	9	-4	∞
2	6	0	∞	2
3	∞	5	0	∞
4	∞	∞	1	0

Since infinity is not greater than infinity, so we keep the original value of $A^0[4,2]$ for $A^1[4,2]$ which is ∞.

$A^1=$

	1	2	3	4
1	0	9	-4	∞
2	6	0	2	2
3	∞	5	0	∞
4	∞	∞	-	0

Step 2: Derive A^1 from A^0

- To find $A^1[4,3]$ Check:

$A^0[4,3] > A^0[4,1] + A^0[1,3]$

$1 > \infty + -4$

$1 > \infty$

Since 1 is not greater than infinity, so we keep the original value of $A^0[4,3]$ for $A^1[4,3]$ which is 1.

$A^0=$

	1	2	3	4
1	0	9	-4	∞
2	6	0	∞	2
3	∞	5	0	∞
4	∞	∞	1	0

$A^1=$

	1	2	3	4
1	0	9	-4	∞
2	6	0	2	2
3	∞	5	0	∞
4	∞	∞	1	0

Step 2: Derive A^1 from A^0

- Final A^1 from A^0

$A^0=$

	1	2	3	4
1	0	9	-4	∞
2	6	0	∞	2
3	∞	5	0	∞
4	∞	∞	1	0

$A^1=$

	1	2	3	4
1	0	9	-4	∞
2	6	0	2	2
3	∞	5	0	∞
4	∞	∞	1	0

Step 3: Repeat Step 2 for all vertices

- We need to find.
  - A2 (for k =2) from A1
  - A3 (for k = 3) from A2
  - A4 for (k = 4) from A3

Step 3: Derive A^2 from A^1

- Here k = 2 acts as the intermediate vertex.
  - Copy the second row and second column from A^1 to A^2 without any changes.
  - Copy the diagonal elements without any changes.
  - Find the values of $A^2[1,3]$, $A^2[1,4]$, $A^2[3,1]$, $A^2[3,4]$, $A^2[4,1]$, and $A^2[4,3]$.

$A^1 =$

	1	2	3	4
1	0	9	-4	∞
2	6	0	2	2
3	∞	5	0	∞
4	∞	∞	1	0

$A^2 =$

	1	2	3	4
1	0	9	-	-
2	6	0	2	2
3	-	5	0	–
4	-	∞	-	0

Step 3: Derive A^2 from A^1

- To find $A^2[1,3]$ Check:

$A^1[1,3] > A^1[1,2] + A^1[2,3]$

$-4 > 9 + 2$

$-4 > 11$

Since -4 is not greater than 11, so we keep the original value of $A^1[1,3]$ for $A^2[1,3]$ which is -4.

$A^1 =$

	1	2	3	4
1	0	9	-4	∞
2	6	0	2	2
3	∞	5	0	∞
4	∞	∞	1	0

$A^2 =$

	1	2	3	4
1	0	9	-4	-
2	6	0	2	2
3	-	5	0	–
4	-	∞	-	0

Step 3: Derive A^2 from A^1

- Similarly find $A^2[1,4]$, $A^2[3,1]$, $A^2[3,4]$, $A^2[4,1]$, and $A^2[4,3]$.
- Final A^2 from A^1 is given below:

$A^1 =$

	1	2	3	4
1	0	9	-4	∞
2	6	0	2	2
3	∞	5	0	∞
4	∞	∞	1	0

$A^2 =$

	1	2	3	4
1	0	9	-4	-
2	6	0	2	2
3	11	5	0	7
4	∞	∞	1	0

Step 3: Derive A^3 from A^2

- Here k = 3 acts as the intermediate vertex.
  - Copy the third row and third column from A^2 to A^3 without any changes.
  - Copy the diagonal elements without any changes.
  - Find the values of $A^3[1,2]$, $A^3[1,4]$, $A^3[2,1]$, $A^3[2,4]$, $A^3[4,1]$, and $A^3[4,2]$.

$A^2 =$

	1	2	3	4
1	0	9	-4	-
2	6	0	2	2
3	11	5	0	7
4	∞	∞	1	0

$A^3 =$

	1	2	3	4
1	0	–	-4	–
2	–	0	2	–
3	11	5	0	7
4	–	–	1	0

Step 3: Derive A^3 from A^2

- Final A3 from A2 is given below:

$A^2 =$

	1	2	3	4
1	0	9	-4	-
2	6	0	2	2
3	11	5	0	7
4	∞	∞	1	0

$A^3 =$

	1	2	3	4
1	0	1	-4	3
2	6	0	2	2
3	11	5	0	7
4	12	6	1	0

Step 3: Derive A^4 from A^3

- Here k = 4 acts as the intermediate vertex.
  - Copy the fourth row and fourth column from A^3 to A^4 without any changes.
  - Copy the diagonal elements without any changes.
  - Find the values of $A^4[1,2]$, $A^4[1,3]$, $A^4[2,1]$, $A^4[2,3]$, $A^4[3,1]$, and $A^4[3,2]$.

$A^3 =$

	1	2	3	4
1	0	1	-4	3
2	6	0	2	2
3	11	5	0	7
4	12	6	1	0

$A^4 =$

	1	2	3	4
1	0	–	–	3
2	–	0	–	2
3	–	–	0	7
4	12	6	1	0

Step 3: Derive A^4 from A^3

- Final A4 from A3 is given below:

$A^3 =$

	1	2	3	4
1	0	1	-4	3
2	6	0	2	2
3	11	5	0	7
4	12	6	1	0

$A^4 =$

	1	2	3	4
1	0	1	−4	3
2	6	0	2	2
3	11	5	0	7
4	12	6	1	0

Step 4: Final distance matrix

- A4 gives us the shortest path between each pair of the graph.

$$A^4 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{pmatrix} 0 & 1 & -4 & 3 \\ 6 & 0 & 2 & 2 \\ 11 & 5 & 0 & 7 \\ 12 & 6 & 1 & 0 \end{pmatrix} \end{array}$$

Floyd-Warshall Algorithm – Pseudocode

```
function FloydWarshall (graph)
    dist = array of size n x n

    // Initialize distance matrix
    for i from 1 to n
        for j from 1 to n
            if i == j
                dist[i][j] = 0
            else if edge (i, j) exists
                dist[i][j] = weight of edge (i, j)
            else
                dist[i][j] = infinity

    // Main loop
    for k from 1 to n
        for i from 1 to n
            for j from 1 to n
                if dist[i][j] > dist[i][k] + dist[k][j]
                    dist[i][j] = dist[i][k] + dist[k][j]

    return dist
```

Floyd-Warshall Algorithm – Time complexity

- The time complexity of the Floyd-Warshall algorithm is O(n3), where ‘n’ is the number of vertices in the graph. There are three loops in the algorithm that iteratively update the shortest path distances between every pair of vertices.
- Since each of these loops runs n times, the total number of iterations is: $n \times n \times n = n^3$
  - Outer Loop (k): This loop runs n times, iterating over each possible intermediate vertex.
  - Middle Loop (i): For each iteration of the outer loop, this loop runs n times, iterating over each source vertex.
  - Inner Loop (j): For each iteration of the middle loop, this loop runs n times, iterating over each destination vertex.

Floyd-Warshall Algorithm – Space complexity

- The time complexity of the Floyd-Warshall algorithm is O(n^3), where 'n' is the number of vertices in the graph.
- There are three loops in the algorithm that iteratively update the shortest path distances between every pair of vertices. Since each of these loops runs n times, the total number of iterations is :$n \times n \times n = n^3$
  - Outer Loop (k): This loop runs n times, iterating over each possible intermediate vertex.
  - Middle Loop (i): For each iteration of the outer loop, this loop runs n times, iterating over each source vertex.
  - Inner Loop (j): For each iteration of the middle loop, this loop runs n times, iterating over each destination vertex.

- The space complexity of the algorithm is **O(n^2).**

Floyd-Warshall Algorithm – Practical Implementation

```java
public class FloydWarshall {
    final static int INF = 99999;

    public static void main(String[] args) {
        // Given graph information
        int graph[][] = { { 0, 9, -4, INF },
                { 6, 0, INF, 2 },
                { INF, 5, 0, INF },
                { INF, INF, 1, 0 } };

        int n = graph.length;
        // Create a distance matrix array
        int dist[][] = new int[n][n];

        // Initialize the distance matrix
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                if (i == j) {
                    dist[i][j] = 0;
                } else if (graph[i][j] != 0) {
                    dist[i][j] = graph[i][j];
                } else {
                    dist[i][j] = INF;
                }
            }
        }

        // Main loop
        for (int k = 0; k < n; k++) {
            for (int i = 0; i < n; i++) {
                for (int j = 0; j < n; j++) {
                    if (dist[i][j] > dist[i][k] + dist[k][j]) {
                        dist[i][j] = dist[i][k] + dist[k][j];
                    }
                }
            }
        }

        // Print the shortest distance matrix
        printSolution(dist);
    }
```

Floyd-Warshall Algorithm – Practical Implementation

```
    // A utility function to print the solution
    static void printSolution(int dist[][]) {
        int n = dist.length;
        System.out.println("The following matrix shows the shortest distances" +
        "between every pair of vertices:");
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                if (dist[i][j] == INF) {
                    System.out.print("INF ");
                } else {
                    System.out.print(dist[i][j] + " ");
                }
            }
            System.out.println();
        }
    }
}
```