Analysis of Algorithm

Dr. Naseer Ahmed Sajid email id: naseer@biit.edu.pk WhatsApp# 0346-5100010

(Week 15) Lectures 29 & 30

Objectives: Learning objectives of these lectures are

- **Floyd-Warshall Algorithm**
- **Dijkstra's Algorithm**

Text Book & Resources:

1. Introduction to Algorithms by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein, The MIT Press; 3rdEdition (2009). ISBN-10: 0262033844
2. Introduction to the Design and Analysis of Algorithms by Anany Levitin, Addison Wesley; 2ndEdition (2006). ISBN-10: 0321358287
3. Algorithms in C++ by Robert Sedgewick (1999). ASIN: B006UR4BJS
4. Algorithms in Java by Robert Sedgewick, Addison-Wesley Professional; 3rd Edition(2002). ISBN-10: 0201361205

Analysis of Algorithm

Dr. Naseer Ahmed Sajid email id: naseer@biit.edu.pk WhatsApp# 0346-5100010

In the last lecture (**Week#14**), we discussed "Graph Colouing" and problems in hashing and how to resolve these problems. In this week, we will discuss "Floyd Warshall Algorithm & Dijkstra's Algorithm"

❖ Floyd-Warshall Algorithm

Definition:

The **Floyd-Warshall Algorithm** is a dynamic programming algorithm used to compute the shortest paths between all pairs of nodes in a weighted graph. It works for directed and undirected graphs, as long as there are no negative weight cycles.

Key Features:

1. **Input**: A weighted graph represented as an adjacency matrix or tabular form.
2. **Output**: A matrix containing the shortest path distances between every pair of nodes.
3. **Process**: Iteratively updates paths by checking if an intermediate node provides a shorter route.

Explanation:

- The algorithm assumes that the direct edge between two nodes is the shortest path.
- It iteratively refines the paths by considering intermediate nodes one at a time.
- **Update rule:**

$$d[i][j] = \min(d[i][j] , d[i][k] + d[k][j])$$

d[i][j]: Shortest distance from node i to node j.
k: Current intermediate node or Key Node.

For example, in a graph with nodes A,B,C:

- Direct path from A→C (A to C = 10).
- Path via BB: A→B→C (A to B to C) with weights 3 and 4.
- The algorithm identifies A→C (A to C) via BB as shorter (7 instead of 10).

Applications in Real Life

1. **Network Routing**
   - Used in computer networks to find the shortest path for data packets, ensuring faster and efficient communication.

- Example: Optimizing routes in the Internet's backbone.

2. **Transportation and Logistics**
   - Determines the shortest paths between cities or locations in road and rail networks.
   - Helps companies like delivery services optimize routes for cost and time efficiency.
   - Helps cab services companies like Careem, Uber, Indrive and Yango.

3. **Social Network Analysis**
   - Identifies the shortest connections between individuals in social graphs.
   - Useful in platforms like Facebook or LinkedIn for suggesting connections.
4. **Game Development**
   - Helps in AI pathfinding where all-pairs shortest paths need to be calculated, such as in strategy or simulation games.
5. **Telecommunication Networks**
   - Determines the shortest path in communication systems to reduce signal delay.
6. **Urban Planning**
   - Used for optimizing traffic flow and designing public transport systems by finding the most efficient routes.

Advantages

- Finds all-pairs shortest paths efficiently.
- Straightforward and simple to implement.

Limitations

- Computationally expensive for very large graphs (**O(N^3) complexity**).
- Does not work for graphs with negative weight cycles.

By understanding the Floyd-Warshall Algorithm and its real-world applications, we see how a simple mathematical approach can solve complex, everyday problems efficiently. Let's now explore a step-by-step example to see the algorithm in action!

Step-by-Step Example of Floyd-Warshall Algorithm

Given Directed Graph

Let's work with a directed graph of 4 nodes: A, B, C, D. The weights of the edges are given below:

From → To	A	B	C	D
A	0	3	∞	7
B	8	0	2	∞

Dr. Naseer Ahmed Sajid email id: naseer@biit.edu.pk WhatsApp# 0346-5100010

From → To	A	B	C	D
C	5	∞	0	1
D	2	∞	∞	0

Here, ∞∞ represents the absence of a direct edge between two nodes.

Step 1: Initialization

- Create the **distance matrix** d[i][j], where d[i][j] is the weight of the edge from i to j.
- If no direct edge exists, set d[i][j]=∞.
- Set d[i][i]=0 for all i (distance from a node to itself is 0).

Initial distance matrix:

From → To	A_	B_	C_	D_
A	0	3	∞	7
B	8	0	2	∞
C	5	∞	0	1
D	2	∞	∞	0

Step 2: Iteration for Intermediate Nodes

We now update the matrix by considering each node kk as an intermediate node, one at a time, and checking if the paths i→k→j (i to k to j) are shorter than the direct path i→j (i to j).

Iteration 1: Using A as an Intermediate or Key Node (k=A)

Update d[i][j]=min(d[i][j] , d[i][A]+d[A][j]):
No new shorter paths are found using A. The distance matrix remains unchanged:

From → To	A_	B_	C_	D_
A	0	3	∞	7
B	8	0	2	15
C	5	8	0	1
D	2	5	∞	0

Iteration 2: Using B as an Intermediate Node (k=B)

Update d[i][j]=min(d[i][j] , d[i][B]+d[B][j]):

- A→C (A to C): min(d[A][C], d[A][B]+d[B][C]) = min(∞,3+2) = 5.
- B→D (B to D): No update, as d[B][D] = ∞.

Updated matrix:

From → To	A_	B_	C_	D_
A	0	3	5	7
B	8	0	2	15
C	5	8	0	1
D	2	5	7	0

Iteration 3: Using C as an Intermediate Node (k = C)

Update d[i][j] = min(d[i][j] , d[i][C] + d[C][j])
B→D (B to D): min (∞, d[B][C] + d[C][D]) = min(∞,2+1) = 3
D→A (D to A): No update, as d[D][A] = 2.
Updated matrix:

From → To	A_	B_	C_	D_
A	0	3	5	6
B	7	0	2	3
C	5	8	0	1
D	2	5	7	0

Iteration 4: Using D as an Intermediate Node (k = D)

Update d[i][j] = min(d[i][j],d[i][D]+d[D][j]):

- C→B (C to B): min(∞, d[C][D] + d[D][B]) = min(∞,1+∞) = ∞
- D→C (D to C): min(∞,d[D][C]) = remains unchanged.

Final matrix:

From → To	A_	B_	C_	D_
A	0	3	5	6
B	5	0	2	3

From → To	A_	B_	C_	D_
C	3	6	0	1
D	2	5	7	0

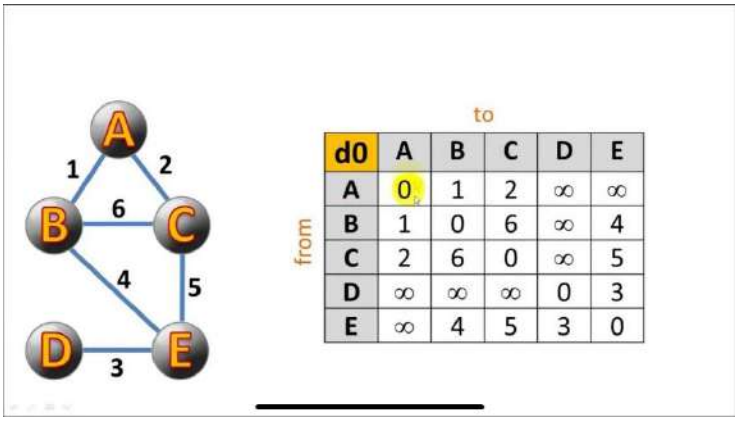
Final Result

The matrix now contains the shortest distances between all pairs of nodes. This completes the Floyd-Warshall Algorithm!

Step-by-Step Example of Floyd-Warshall Algorithm

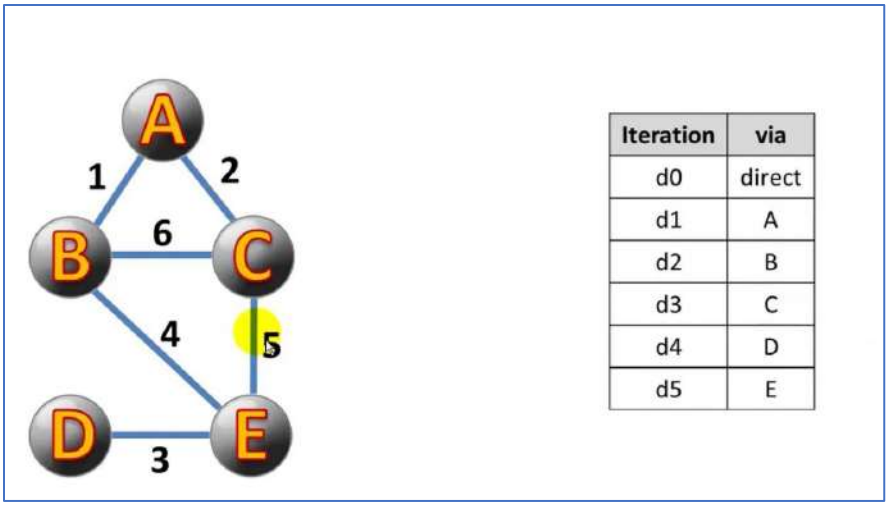
Undirected Graph

1.



d0	A	B	C	D	E
A	0	1	2	∞	∞
B	1	0	6	∞	4
C	2	6	0	∞	5
D	∞	∞	∞	0	3
E	∞	4	5	3	0

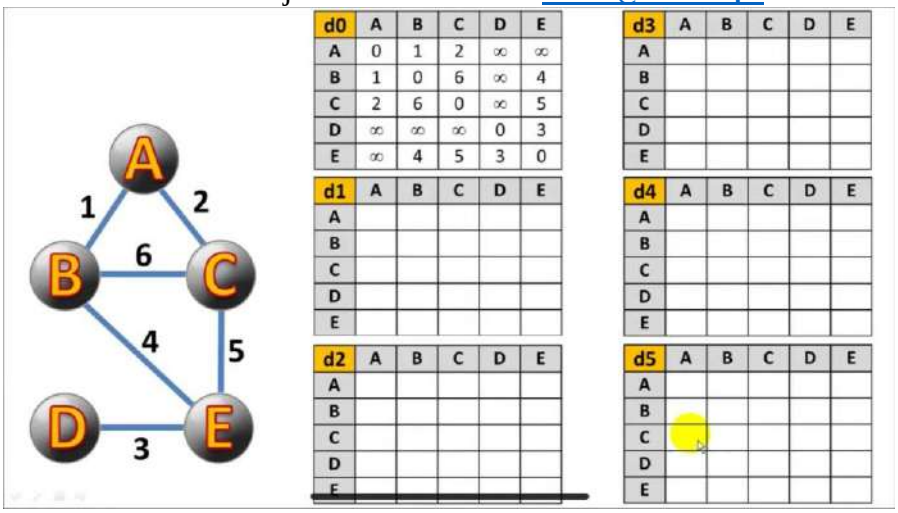
2.



Iteration	via
d0	direct
d1	A
d2	B
d3	C
d4	D
d5	E

3.

Dr. Naseer Ahmed Sajid email id: naseer@biit.edu.pk WhatsApp# 0346-5100010



d0	A	B	C	D	E
A	0	1	2	∞	∞
B	1	0	6	∞	4
C	2	6	0	∞	5
D	∞	∞	∞	0	3
E	∞	4	5	3	0

d1	A	B	C	D	E
A					
B					
C					
D					
E					

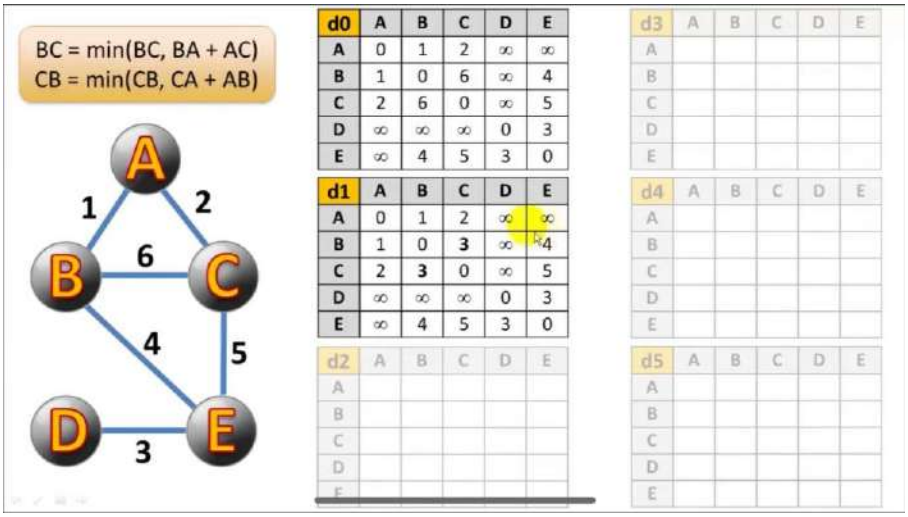
d2	A	B	C	D	E
A					
B					
C					
D					
E					

d3	A	B	C	D	E
A					
B					
C					
D					
E					

d4	A	B	C	D	E
A					
B					
C					
D					
E					

d5	A	B	C	D	E
A					
B					
C					
D					
E					

4.



d0	A	B	C	D	E
A	0	1	2	∞	∞
B	1	0	6	∞	4
C	2	6	0	∞	5
D	∞	∞	∞	0	3
E	∞	4	5	3	0

d1	A	B	C	D	E
A	0	1	2	∞	∞
B	1	0	**3**	∞	4
C	2	**3**	0	∞	5
D	∞	∞	∞	0	3
E	∞	4	5	3	0

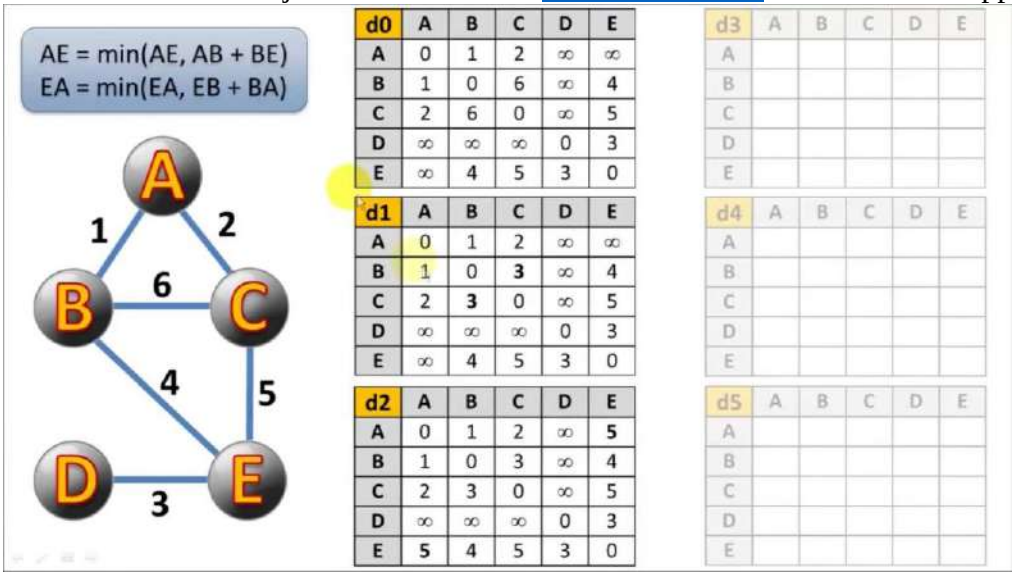
d2	A	B	C	D	E
A					
B					
C					
D					
E					

d3	A	B	C	D	E
A					
B					
C					
D					
E					

d4	A	B	C	D	E
A					
B					
C					
D					
E					

d5	A	B	C	D	E
A					
B					
C					
D					
E					

5.



d0	A	B	C	D	E
A	0	1	2	∞	∞
B	1	0	6	∞	4
C	2	6	0	∞	5
D	∞	∞	∞	0	3
E	∞	4	5	3	0

d1	A	B	C	D	E
A	0	1	2	∞	∞
B	1	0	**3**	∞	4
C	2	**3**	0	∞	5
D	∞	∞	∞	0	3
E	∞	4	5	3	0

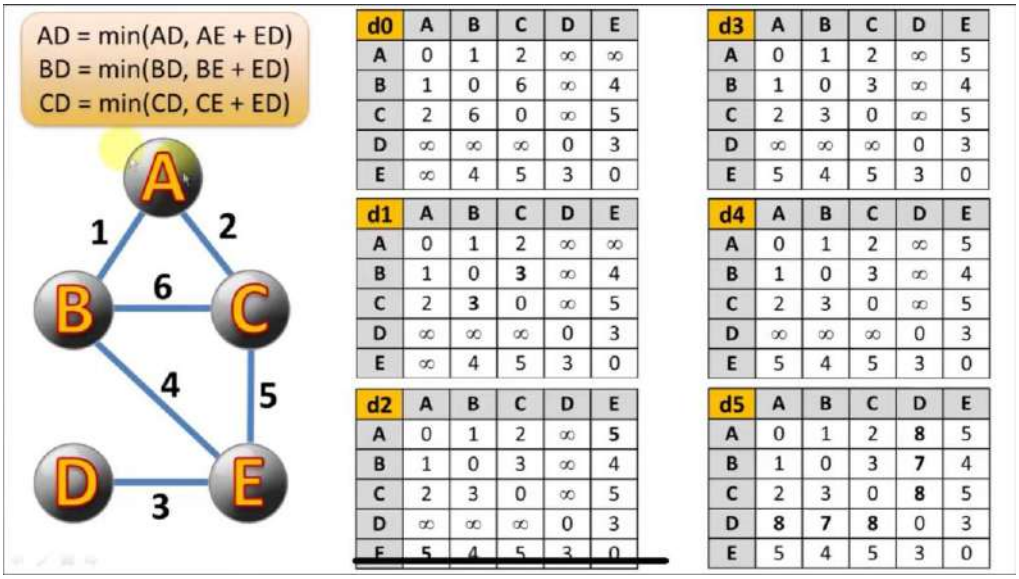
d2	A	B	C	D	E
A	0	1	2	∞	**5**
B	1	0	3	∞	4
C	2	3	0	∞	5
D	∞	∞	∞	0	3
E	**5**	4	5	3	0

d3	A	B	C	D	E
A					
B					
C					
D					
E					

d4	A	B	C	D	E
A					
B					
C					
D					
E					

d5	A	B	C	D	E
A					
B					
C					
D					
E					

6.



d0	A	B	C	D	E
A	0	1	2	∞	∞
B	1	0	6	∞	4
C	2	6	0	∞	5
D	∞	∞	∞	0	3
E	∞	4	5	3	0

d1	A	B	C	D	E
A	0	1	2	∞	∞
B	1	0	**3**	∞	4
C	2	**3**	0	∞	5
D	∞	∞	∞	0	3
E	∞	4	5	3	0

d2	A	B	C	D	E
A	0	1	2	∞	**5**
B	1	0	3	∞	4
C	2	3	0	∞	5
D	∞	∞	∞	0	3
E	**5**	4	5	3	0

d3	A	B	C	D	E
A	0	1	2	∞	5
B	1	0	3	∞	4
C	2	3	0	∞	5
D	∞	∞	∞	0	3
E	5	4	5	3	0

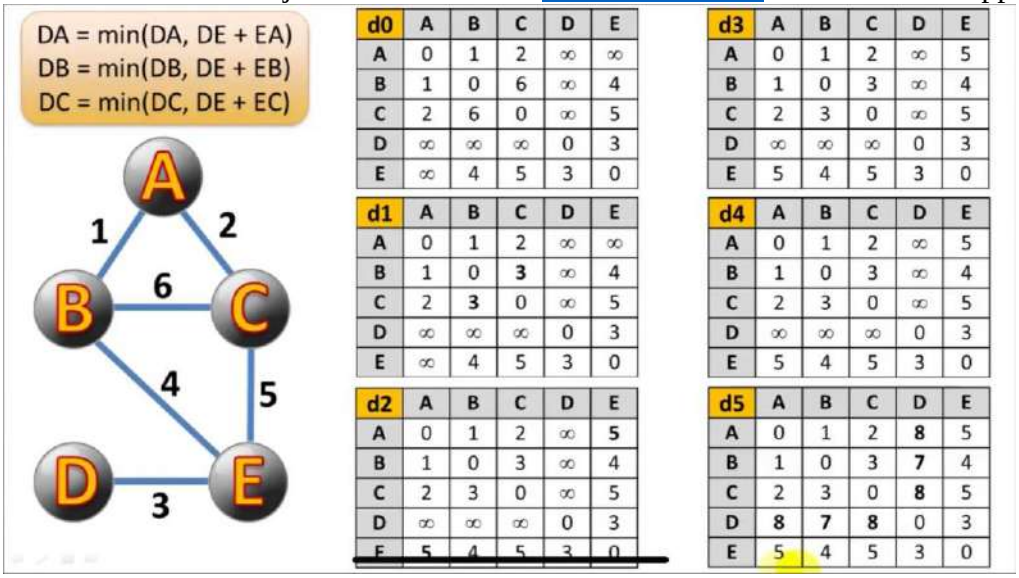
d4	A	B	C	D	E
A	0	1	2	∞	5
B	1	0	3	∞	4
C	2	3	0	∞	5
D	∞	∞	∞	0	3
E	5	4	5	3	0

d5	A	B	C	D	E
A	0	1	2	**8**	5
B	1	0	3	**7**	4
C	2	3	0	**8**	5
D	**8**	**7**	**8**	0	3
E	5	4	5	3	0

7.

Analysis of Algorithm

Dr. Naseer Ahmed Sajid email id: naseer@biit.edu.pk WhatsApp# 0346-5100010



d0	A	B	C	D	E
A	0	1	2	∞	∞
B	1	0	6	∞	4
C	2	6	0	∞	5
D	∞	∞	∞	0	3
E	∞	4	5	3	0

d1	A	B	C	D	E
A	0	1	2	∞	∞
B	1	0	**3**	∞	4
C	2	**3**	0	∞	5
D	∞	∞	∞	0	3
E	∞	4	5	3	0

d2	A	B	C	D	E
A	0	1	2	∞	**5**
B	1	0	3	∞	4
C	2	3	0	∞	5
D	∞	∞	∞	0	3
E	5	4	5	3	0

d3	A	B	C	D	E
A	0	1	2	∞	5
B	1	0	3	∞	4
C	2	3	0	∞	5
D	∞	∞	∞	0	3
E	5	4	5	3	0

d4	A	B	C	D	E
A	0	1	2	∞	5
B	1	0	3	∞	4
C	2	3	0	∞	5
D	∞	∞	∞	0	3
E	5	4	5	3	0

d5	A	B	C	D	E
A	0	1	2	**8**	5
B	1	0	3	**7**	4
C	2	3	0	**8**	5
D	**8**	**7**	**8**	0	3
E	5	4	5	3	0

Recurrence Relation and Implementations

Here's a comprehensive breakdown of the **recurrence relation**, **recursive definition**, **recursive algorithm**, and **memoized recursive algorithm** for the Floyd-Warshall algorithm.

1. Recurrence Relation

The Floyd-Warshall algorithm solves the all-pairs shortest path problem using the following recurrence relation:

$$d[i][j](k) = \begin{cases} w[i][j], & \text{if } k = 0 \\ \min(d[i][j](k-1), d[i][k](k-1) + d[k][j](k-1)), & \text{if } k > 0 \end{cases}$$

Where:

- d[i][j](k) is the shortest distance from node I to node j using only the first k intermediate nodes.
- w[i][j] is the weight of the direct edge from i to j, or infinity (∞\infty) if no edge exists.
- k is the current intermediate node being considered.

2. Recursive Definition

The shortest path between two nodes i and j, considering up to the k^{th} node as an intermediate, is recursively defined as:

1. If no intermediate nodes are allowed (k = 0):
    - The shortest path is the direct edge weight w[i][j].
2. If intermediate nodes are allowed (k>0):
    - The shortest path is either:

- The shortest path without using the k^{th} node (d[i][j](k−1)), or
- The shortest path passing through the k^{th} node (d[i][k](k−1)+d[k][j](k−1).

3. Recursive Algorithm

Here's a recursive implementation of the Floyd-Warshall algorithm:

```
def floyd_warshall_recursive(i, j, k, weights):

    if k == 0:
        return weights[i][j]

    without_k = floyd_warshall_recursive(i, j, k-1, weights)
    with_k = (floyd_warshall_recursive(i, k, k-1, weights) +
              floyd_warshall_recursive(k, j, k-1, weights))

    return min(without_k, with_k)

# Example: Initialize weights matrix with graph edges
weights = [
    [0, 3, float('inf'), 7],
    [8, 0, 2, float('inf')],
    [5, float('inf'), 0, 1],
    [2, float('inf'), float('inf'), 0]
]

# Find shortest distance from node 0 to node 3 using all nodes as intermediates
result = floyd_warshall_recursive(0, 3, len(weights) - 1, weights)
print(result)
```

4. Memoized Recursive Algorithm

The recursive approach can be inefficient due to repeated calculations. Using memoization, we can store already computed results to optimize performance.

```
def floyd_warshall_memoized(i, j, k, weights, memo):
    # Base case: No intermediate nodes allowed
    if k == 0:
        return weights[i][j]

    # Check if the result is already computed
    if (i, j, k) in memo:
        return memo[(i, j, k)]

    # Recursive case: Consider intermediate node k
    without_k = floyd_warshall_memoized(i, j, k-1, weights, memo)
    with_k = (floyd_warshall_memoized(i, k, k-1, weights, memo) +
```

```
        floyd_warshall_memoized(k, j, k-1, weights, memo))

    # Store the result in memo and return
    memo[(i, j, k)] = min(without_k, with_k)
    return memo[(i, j, k)]

# Example: Initialize weights matrix with graph edges
weights = [
    [0, 3, float('inf'), 7],
    [8, 0, 2, float('inf')],
    [5, float('inf'), 0, 1],
    [2, float('inf'), float('inf'), 0]
]

# Initialize memoization dictionary
memo = {}

# Find shortest distance from node 0 to node 3 using all nodes as intermediates
result = floyd_warshall_memoized(0, 3, len(weights) - 1, weights, memo)
print(result)
```

Key Notes:

- **Recursive Approach**: Simple to understand but inefficient due to redundant computations.
- **Memoization**: Improves efficiency by avoiding repeated calculations, reducing complexity significantly.
- **For practical purposes**, the iterative approach is usually preferred over the recursive one because of its clarity and efficiency.
- **Time Complexity O (n^3) and Space Complexity O (n^2)**

❖ Dijkstra's Algorithm

Key Characteristics

1. **Type of Graph**: Works on directed and undirected graphs with non-negative edge weights.
2. **Problem Solved**: Finds the shortest paths from a source vertex to all other vertices in the graph.
3. **Algorithm Paradigm**: Greedy.
4. **Data Structures Used**:
   Priority Queue (typically implemented using a Min-Heap or Fibonacci Heap).
   Adjacency List or Adjacency Matrix to represent the graph.

Dr. Naseer Ahmed Sajid email id: naseer@biit.edu.pk WhatsApp# 0346-5100010

Steps in the Algorithm

1. Initialization:

Assign a tentative distance of 0 to the source vertex and infinity () to all other vertices. Mark all vertices as unvisited.

2. Processing:

Start with the source vertex.
Update the distances of all its neighboring vertices.
Select the unvisited vertex with the smallest tentative distance as the next vertex to process.
Repeat until all vertices are visited or the smallest tentative distance among unvisited vertices is infinity.

3. Termination:

Once all vertices have been visited, the shortest path from the source to all vertices is determined.

Analysis of Algorithm

1. Correctness:

Based on the greedy approach, Dijkstra's Algorithm ensures the shortest path is found for each vertex once it's added to the finalized set.

Relies on the property that once a vertex's shortest path is determined, it won't change.

2. Time Complexity:

Using Adjacency Matrix: , where is the number of vertices.

Using Min-Heap and Adjacency List: , where is the number of edges.

With Fibonacci Heap: , though it is more complex to implement.

3. Space Complexity:

, for storing the graph and the priority queue.

4. Limitations:

Does not handle negative edge weights (e.g., Bellman-Ford Algorithm is used for such cases).

Efficiency is reduced in dense graphs when using an adjacency matrix.

Applications

Routing Algorithms: Used in network routing protocols like OSPF (Open Shortest Path First).

Pathfinding: In GPS navigation systems.

Game Development: Finding paths in game maps.

Robotics: Motion planning in robot navigation.

For Examples

Dr. Naseer Ahmed Sajid email id: naseer@biit.edu.pk WhatsApp# 0346-5100010