Networks: Optimizing Network Routing: Bellman-Ford, Dijkstra and Floyd-Warshall

Ali Benzerbadj

University of Ain Temouchent Belhadj Bouchaïb

Plan

1. A Quick Reminder
2. Bellman-Ford Algorithm
   - Pseudocode
   - Example
   - Time Complexity

Plan

3 Dijkstra Algorithm
- Pseudocode
- Example
- Time complexity

4 Floyd-Warshall Algorithm
- Pseudocode
- Example
- Time Complexity

Plan

5 Comparison of Algorithms

6 How Bellman-Ford algorithm is leveraged in the RIP

Optimizing Network Routing

A Quick Reminder

- **Dijkstra** : Shortest path from **one** node to **all** nodes, **negative** edge weights not allowed.
- **Bellman-Ford** : Shortest path from **one** node to **all** nodes, **negative** edge weights allowed.
- **Floyd-Warshall** : Shortest path between **all** pairs of vertices, **negative** edge weights allowed.

Optimizing Network Routing

Bellman-Ford Algorithm

The Bellman-Ford algorithm shares some characteristics with dynamic programming, however it is more accurately described as a relaxation-based algorithm.

Optimizing Network Routing

Bellman-Ford Algorithm

- The Bellman-Ford Algorithm is used to find the shortest paths from a single source vertex to all other vertices in a weighted graph.
- It can handle graphs with negative weight edges, unlike Dijkstra's algorithm, but cannot handle negative weight cycles.

Optimizing Network Routing

Bellman-Ford Algorithm (Cont.)

1. **Input of the Algorithm :**
   - A weighted graph $G = (V, E, w)$ where V is a set of vertices, E is a set of edges, and w represents the weight of the edges.
   - A source vertex *src* from which the shortest paths will be calculated.
2. **Output of the Algorithm :**
   - The shortest path distances from the source vertex to all other vertices, stored in a distance array.
   - A predecessor array to reconstruct the paths.

Optimizing Network Routing

Algorithm 1 Bellman-Ford : Initialization Pseudocode

Require: Graph $G(V, E, w)$ with vertices V, edges E and $w : E \to \mathbb{R}$ is a function that assigns a weight to each edge. Note that if there is no edge between vertex u and vertex v, weight(u, v) $= \infty$
Require: Source vertex $src \in V$
Ensure: Shortest path distances from *src* to all other vertices in G
Ensure: Predecessor information to reconstruct shortest paths

```
1: procedure BellmanFord(G, src)
2:     Initialize distance[] and predecessor[] arrays :
3:     for each vertex v in G except src do
4:         distance[v] = ∞
5:         predecessor[v] = null
6:     end for
7:     distance[src] = 0
8: end procedure
```

Optimizing Network Routing

Algorithm 2 Bellman-Ford : Relaxation Pseudocode

```
1:  procedure BellmanFord(G, src)
2:      for each vertex i from 1 to |V| − 1 do
3:          for each edge (u, v) in G do
4:              if distance[u] + weight(u, v) < distance[v] then
5:                  distance[v] = distance[u] + weight(u, v)
6:                  predecessor[v] = u
7:              end if
8:          end for
9:      end for
10: end procedure
```

Optimizing Network Routing

Algorithm 3 Bellman-Ford : Negative-Weight Cycles Pseudo-code

```
1: procedure BellmanFord(G, src)
2:     for each edge (u, v) in G do
3:         if distance[u] + weight(u, v) < distance[v] then
4:             Error : Negative-weight cycle detected
5:             Terminate the algorithm
6:         end if
7:     end for
8:     Return distance[], predecessor[]
9: end procedure
```

Optimizing Network Routing

Explanation

1. Step 1 : Initialization
   - In this step, the algorithm initilizes :
     1. The distances to all vertices as infinite (∞), which implies that the distance from the source vertex to all other vertices is initially considered to be infinite
     2. The distance to the source vertex as 0, indicating that the shortest path from the source to itself is zero.
   - Additionally, it initializes an array to keep track of the predecessors of each vertex.

Optimizing Network Routing

Explanation (Cont.)

2. Step 2 : Relaxation
   - For each vertex, relax all the edges. This means checking if a shorter path exists to each vertex through another vertex and updating accordingly.

Optimizing Network Routing

Explanation (Cont.)

2. Step 2 : Relaxation
   - Mathematically, for an edge (u,v) with weight weight(u,v), we check if : distance$[v]$>distance$[u]$+weight(u,v), where :
     - distance$[u]$: Is the current known shortest distance from the source to vertex u
     - distance$[v]$: This is the current known shortest distance from the source to vertex v.
     - weight(u,v) : Is the weight of the edge from vertex u to vertex v.

Optimizing Network Routing

Explanation (Cont.)

2. Step 2 : Relaxation
   - If the condition is true : It means that going from the source vertex to vertex v through vertex u provides a shorter path. Therefore, you update the distance to vertex v and set the predecessor of v to u (indicating that the shortest path to v currently goes through u) :
     - distance[v]=distance[u]+weight(u,v)

Optimizing Network Routing

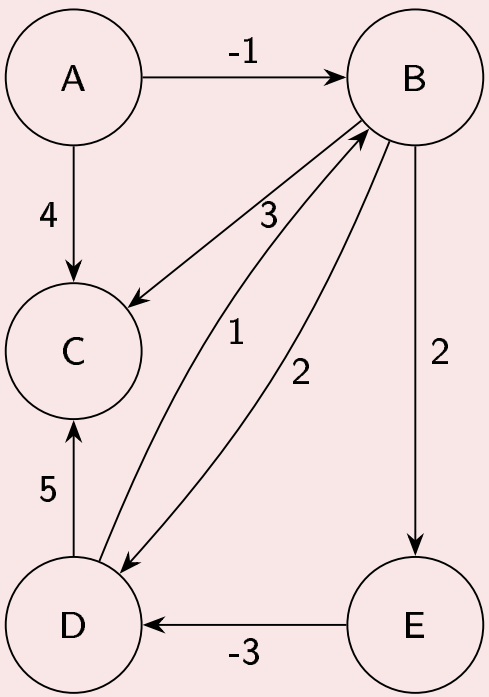
Explanation (Cont.)

3. Step 3 : Check for Negative-Weight Cycles
   - Finally, the algorithm checks for negative weight cycles.
   - If it finds a shorter path even after |V|-1 iterations, it indicates the presence of a negative weight cycle.

Optimizing Network Routing

Example

Consider the following graph with 5 vertices (A, B, C, D, E) and weighted edges :



Optimizing Network Routing

Example (Cont.)

Let's find the shortest paths from vertex A to all other vertices using the Bellman-Ford algorithm.

1. Initialization :
   - Set the distance to the source vertex A as 0.
   - Set the distance to all other vertices as infinity (∞)

Optimizing Network Routing

Table 1 – Initialization.

Vertex	Distance	Predecessor
A	0	-
B	∞	-
C	∞	-
D	∞	-
E	∞	-

Optimizing Network Routing

Example (Cont.)

2. Relaxation :
   - First Iteration (Relax all edges) :
   1. Edge (A, B, -1)
      - distance[B]=min(∞,0-1)=-1
      - Predecessor of B is A.
   2. Edge (A, C, 4) :
      - distance[C]=min(∞,0+4)=4
      - Predecessor of C is A
   3. Edge (B, C, 3) :
      - distance[C]=min(4,-1+3)=2
      - Predecessor of C is B

Optimizing Network Routing

Example (Cont.)

2. Relaxation :
   - First Iteration (Relax all edges) :
   1. Edge (B, D, 2) :
      - distance[D]=min(∞,-1+2)=1
      - Predecessor of D is B.
   2. Edge (B, E, 2) :
      - distance[E]=min(∞,-1+2)=1
      - Predecessor of E is B.
   3. Edge (D, B, 1) :
      - distance[B]=min(-1,1+1)= -1 (no update)

Optimizing Network Routing

Example (Cont.)

2. Relaxation :
   - First Iteration (Relax all edges) :
   1. Edge (D, C, 5) :
      - distance[C]=min(2,1+5)=2 (no update)
   2. Edge (E, D, -3) :
      - distance[D]=min(1,1-3)=-2
      - Predecessor of D is E.

Optimizing Network Routing

Table 2 – Relaxation : 1^{st} iteration.

Vertex	Distance	Predecessor
A	0	-
B	-1	A
C	2	B
D	-2	E
E	1	B

Optimizing Network Routing

Example (Cont.)

2. Relaxation :
   - Second Iteration (Relax all edges) :
   1. Edge (A, B, -1)
      - distance[B]=min(-1,0-1)=-1 (no update)
   2. Edge (A, C, 4) :
      - distance[C]=min(2,0+4)=2 (no update)
   3. Edge (B, C, 3) :
      - distance[C]=min(2,-1+3)=2 (no update)

Optimizing Network Routing

Example (Cont.)

2. Relaxation :
   - Second Iteration (Relax all edges) :
   1. Edge (B, D, 2) :
      - distance[D]=min(-2,-1+2)=-2 (no update)
   2. Edge (B, E, 2) :
      - distance[E]=min(1,-1+2)=1 (no update)
   3. Edge (D, B, 1) :
      - distance[B]=min(-1,-2+1)= -1 (no update)

Optimizing Network Routing

Example (Cont.)

2. Relaxation :
   - Second Iteration (Relax all edges) :
   1. Edge (D, C, 5) :
      - distance[C]=min(2,-2+5)=2 (no update)
   2. Edge (E, D, -3) :
      - distance[D]=min(-2,1-3)=-2 (no update)

Optimizing Network Routing

Table 3 – Relaxation : 2^{nd} iteration.

Vertex	Distance	Predecessor
A	0	-
B	-1	A
C	2	B
D	-2	E
E	1	B

Optimizing Network Routing

- Relaxing all edges three times, the table looks like this :

Table 4 – Relaxation : 3^{rd} iteration.

Vertex	Distance	Predecessor
A	0	-
B	-1	A
C	2	B
D	-2	E
E	1	B

Optimizing Network Routing

- Relaxing all edges one last time :

Table 5 – Relaxation : 4^{th} iteration.

Vertex	Distance	Predecessor
A	0	-
B	-1	A
C	2	B
D	-2	E
E	1	B

Optimizing Network Routing

Example (Cont.)

3. Checking for Negative-Weight Cycles :
   - In a graph with no negative-weight cycles, after $|V|$-1 iterations, the distances will stabilize.
   - If we perform another iteration and find that any distance can still be reduced, it indicates the presence of a negative-weight cycle.

Optimizing Network Routing

- Relaxing all edges one more time :

Table 6 – Relaxation : 5^{th} iteration.

Vertex	Distance	Predecessor
A	0	-
B	-1	A
C	2	B
D	-2	E
E	1	B

Optimizing Network Routing

Time Complexity

1. The time complexity of the Bellman-Ford algorithm is $\mathcal{O}(V * E)$, where V is the number of vertices and E is the number of edges in the graph :

Optimizing Network Routing

Time Complexity (Cont.)

- **Initialization** : Setting the initial distances takes $\mathcal{O}(V)$ time.
- **Relaxation** : The algorithm relaxes all edges $|V|$ - 1 times, resulting in a time complexity of $\mathcal{O}(V * E)$ for this step.
- **Negative Cycle Check** : In the final step, the algorithm checks for negative-weight cycles by iterating through all edges, which takes $\mathcal{O}(E)$ time.

Optimizing Network Routing

Time Complexity (Cont.)

- Therefore, the overall time complexity is dominated by the relaxation step, leading to $\mathcal{O}(V * E)$.

Optimizing Network Routing

Dijkstra's Algorithm

- Dijkstra's Algorithm is a greedy algorithm used to find the shortest path from a single source vertex to all other vertices in a graph with non-negative edge weights.
- Unlike the Floyd-Warshall algorithm, which computes shortest paths between all pairs of vertices, Dijkstra's algorithm focuses on paths starting from a single source.

Optimizing Network Routing

Dijkstra's Algorithm (Cont.)

1. **Input of the Algorithm :**
   - A weighted graph $G = (V, E, w)$ represented by an adjacency list or matrix, where V is a set of vertices, E is a set of edges, and w represents the non-negative weights of the edges.
   - A source vertex s from which the shortest paths to all other vertices are to be calculated.

Optimizing Network Routing

Dijkstra's Algorithm (Cont.)

2. **Output of the Algorithm :**
   - A distance array where $d[i]$ holds the shortest distance from the source vertex s to vertex i.
   - A predecessor array that helps reconstruct the shortest path from the source to any vertex.

Optimizing Network Routing

Algorithm 4 Dijkstra's Algorithm : Initialization

```
1:  procedure Dijkstra(graph, source)
2:      Input : graph, source
3:      Output : distance[], predecessor[]
4:      for each vertex v in graph do
5:          distance[v] = ∞                          ▷ Initialize distances to infinity
6:          predecessor[v] = null                    ▷ No predecessors yet
7:      end for
8:      distance[source] = 0                         ▷ Distance from source to itself is 0
9:      Create a priority queue Q with all vertices, prioritized by distance
10: end procedure
```

Optimizing Network Routing

Algorithm 5 Dijkstra's Algorithm : Main Loop

```
1:  procedure Dijkstra(graph, source)                          ▷ Continued
2:      while Q is not empty do
3:          u = Extract_Min(Q)                                 ▷ Vertex with the smallest distance
4:          for each neighbor v of u do
5:              alt = distance[u] + weight(u, v)
6:              if alt < distance[v] then
7:                  distance[v] = alt                          ▷ Update shortest distance to v
8:                  predecessor[v] = u                         ▷ Update predecessor of v
9:                  Decrease_Key(Q, v, alt)
10:             end if
11:         end for
12:     end while
13: end procedure
```

Optimizing Network Routing

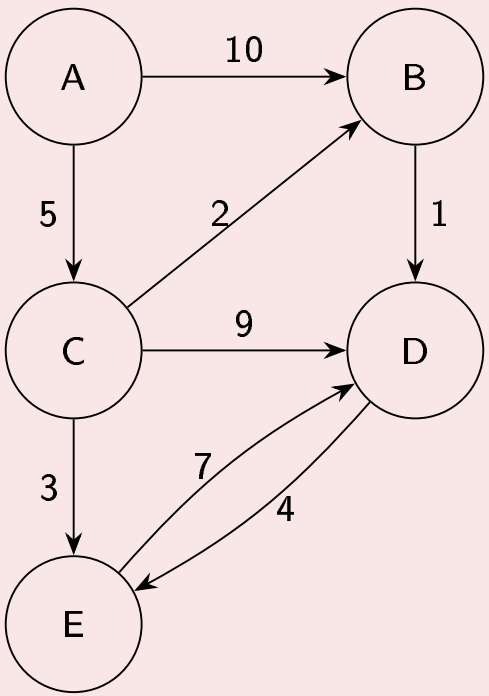
Algorithm 6 Dijkstra's Algorithm : Final Output

```
1: procedure Dijkstra(graph, source)                    ▷ Continued
2:     Return distance[], predecessor[]
3: end procedure
```

Optimizing Network Routing

Example

Consider a graph with 5 vertices (A, B, C, D, E) and the following edges with their weights :



Optimizing Network Routing

Example (Cont.)

- The adjacency matrix :

$$d = \begin{array}{c} \\ A \\ B \\ C \\ D \\ E \end{array} \begin{array}{c} \begin{array}{ccccc} A & B & C & D & E \end{array} \\ \begin{pmatrix} 0 & 10 & 5 & \infty & \infty \\ \infty & 0 & \infty & 1 & \infty \\ \infty & 2 & 0 & 9 & 3 \\ \infty & \infty & \infty & 0 & 4 \\ \infty & \infty & \infty & 7 & 0 \end{pmatrix} \end{array}$$

Optimizing Network Routing

Example (Cont.)

Let's find the shortest paths from vertex A to all other vertices using the Dijkstra's Algorithm :

1. Initialization :
   - Set the initial distances from A to all vertices :
     - distance(A) = 0 (starting vertex)
     - distance(B) = ∞
     - distance(C) = ∞
     - distance(D) = ∞
     - distance(E) = ∞
   - Set the processed set S= $\emptyset$.
   - Predecessor of each vertex is unknown initially.

Optimizing Network Routing

Example (Cont.)

Let's find the shortest paths from vertex A to all other vertices using the Dijkstra's Algorithm :

2. Step 1 (Process A) :
   - Select the unprocessed vertex with the smallest distance, which is A with distance 0.
   - Update the distances of A's neighbors (B and C) :
     - distance(B) = min(∞,0+10)=10
     - distance(C) = min(∞,0+5)=5
   - Predecessors : B $\rightarrow$ A, C $\rightarrow$ A
   - Mark A as processed : S={A}.

Optimizing Network Routing

Example (Cont.)

Let's find the shortest paths from vertex A to all other vertices using the Dijkstra's Algorithm :

3. Step 2 (Process C) :
   - Select the next unprocessed vertex with the smallest distance, which is C with distance 5.
   - Update the distances of C's neighbors (B, D, E) :
     - distance(B) = min(10,5+2)=7
     - distance(D) = min(∞,5+9)=14
     - distance(E) = min(∞,5+3)=8
   - Predecessors : B $\rightarrow$ C, D $\rightarrow$ C, E $\rightarrow$ C
   - Mark C as processed : S={A,C}.

Optimizing Network Routing

Example (Cont.)

Let's find the shortest paths from vertex A to all other vertices using the Dijkstra's Algorithm :

4. Step 3 (Process B) :
   - Select the next unprocessed vertex with the smallest distance, which is B with distance 7.
   - Update the distance of B's neighbor (D) :
     - distance(D) = min(14,7+1)=8
   - Predecessor : D $\rightarrow$ B
   - Mark B as processed : S={A,C, B}.

Optimizing Network Routing

Example (Cont.)

Let's find the shortest paths from vertex A to all other vertices using the Dijkstra's Algorithm :

5. Step 4 (Process E) :
   - Select the next unprocessed vertex with the smallest distance, which is E with distance 8.
   - Update the distance of E's neighbor (D) :
     - distance(D) = min(8,8+7)=8 (no update needed)
   - Mark E as processed : S={A,C, B, E}.

Optimizing Network Routing

Example (Cont.)

Let's find the shortest paths from vertex A to all other vertices using the Dijkstra's Algorithm :

6. Step 5 (Process D) :
   - The last unprocessed vertex is D with distance 8. No neighbors to update.
   - Mark D as processed : S={A,C, B, E, D}.

Optimizing Network Routing

Example (Cont.)

- Final Shortest Distances from A :
  - A to A : 0
  - A to B : 7
  - A to C : 5
  - A to D : 8
  - A to E : 8

Optimizing Network Routing

Remark

- Order of processing D and E : We can choose either vertex in any order since their distances are equal (8).
- For any vertex in Dijkstra's algorithm, if all of its neighbors are already in the set S (meaning they have already been processed), there is no need to update their distances. This is because the shortest path to each of those neighbors has already been determined when they were processed. Therefore, revisiting or updating these neighbors would not yield any shorter paths.

Optimizing Network Routing

Time Complexity

- $\mathcal{O}(V^2)$ with an adjacency matrix
- $\mathcal{O}((E + V) \log V)$ using adjacency list with a min-heap priority queue

Optimizing Network Routing

Floyd-Warshall Algorithm

- The Floyd-Warshall Algorithm is a dynamic programming algorithm used to find the shortest paths between all pairs of vertices in a weighted graph.
- Unlike the Bellman-Ford algorithm, which finds the shortest path from a single source to all other vertices, the Floyd-Warshall algorithm computes the shortest paths between every pair of vertices.

Optimizing Network Routing

Floyd-Warshall Algorithm (Cont.)

1. Input of the Algorithm :
   - A weighted graph G=(V,E,w) represented by an adjacency matrix, where V is a set of vertices, E is a set of edges, and w represents the weight of the edges. The graph can have positive or negative edge weights, but no negative cycles (where a cycle's total weight is negative).
2. Output of the Algorithm :
   - A matrix D where D[i][j] holds the shortest distance from vertex i to vertex j.

Optimizing Network Routing

Floyd-Warshall Algorithm (Cont.)

How the Algorithm Works :

1. Initialization :
   - Start by initializing the distance matrix D with the given weights :
     - If there is an edge from vertex i to vertex j, set D[i][j]=w(i,j).
     - If there is no edge between i and j, set D[i][j]= ∞.
     - The distance from any vertex to itself is zero, so D[i][i]=0 for all i.

Optimizing Network Routing

Floyd-Warshall Algorithm (Cont.)

2. Iterative Update :
   - The algorithm iteratively updates the distance matrix by considering each vertex as an intermediate point on the path between any two vertices.
   - For each pair of vertices (i,j), check if a path through vertex k is shorter than the direct path from i to j. Update D[i][j] if a shorter path is found :
     - D[i][j]=min(D[i][j],D[i][k]+D[k][j])
   - This process is repeated for every vertex k in the graph.

Optimizing Network Routing

Floyd-Warshall Algorithm (Cont.)

3. Final Matrix :
   - After all iterations, the matrix D contains the shortest path distances between all pairs of vertices.

Optimizing Network Routing

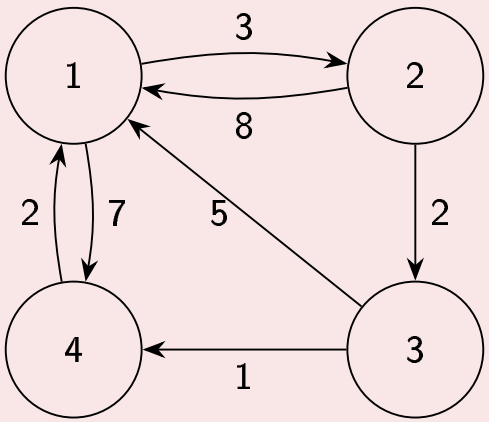
Algorithm 7 Pseudocode of Floyd-Warshall Algorithm

```
1:  Input : A graph G represented by an adjacency matrix d where d[i][j] is the weight of the edge
    from vertex i to vertex j. If there is no edge, d[i][j] is set to infinity.
2:  Output : The shortest path distances between all pairs of vertices, stored in the matrix d.
3:  procedure FloydWarshall(G)
4:      n ← |V|
5:      for k ← 1 to n do
6:          for i ← 1 to n do
7:              for j ← 1 to n do
8:                  if d[i][j] > d[i][k] + d[k][j] then
9:                      d[i][j] ← d[i][k] + d[k][j]
10:                 end if
11:             end for
12:         end for
13:     end for
14: end procedure
```

Optimizing Network Routing

Example

Consider the following graph with 4 vertices (1, 2, 3, 4) and weighted edges :



Optimizing Network Routing

Example (Cont.)

1. Initialisation :

$$d = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{pmatrix} 0 & 3 & \infty & 7 \\ 8 & 0 & 2 & \infty \\ 5 & \infty & 0 & 1 \\ 2 & \infty & \infty & 0 \end{pmatrix} \end{array}$$

- The diagonal elements are 0, representing the distance from any vertex to itself.
- Non-diagonal elements show the weights of the edges between vertices.
- ∞ signifies that there is no edge between those vertices.

Optimizing Network Routing

Example (Cont.)

2. Iterations :

- The algorithm iterates over each vertex as an intermediate vertex.
- For each pair of vertices (i,j), it checks if going through the intermediate vertex provides a shorter path.
- If so, it updates the distance matrix.

$$d = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{pmatrix} 0 & 3 & \infty & 7 \\ 8 & 0 & 2 & \infty \\ 5 & \infty & 0 & 1 \\ 2 & \infty & \infty & 0 \end{pmatrix} \end{array}$$

Optimizing Network Routing

Example (Cont.)

1. Vertex 1 as intermediate vertex $\Rightarrow$ 1^{st} Column and 1^{st} Line remain unchanged.

$$\begin{array}{lcll} d(2,3) & & d(2,1) & + \quad d(1,3) \\ 2 & < & 8 & + \quad \infty \\ d(2,4) & & d(2,1) & + \quad d(1,4) \\ \infty & > & 8 & + \quad 7 \\ d(3,2) & & d(3,1) & + \quad d(1,2) \\ \infty & > & 5 & + \quad 3 \\ \ldots & \ldots & \ldots & \ldots \quad \ldots \end{array}$$

$$\mathrm{d} = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{pmatrix} 0 & 3 & \infty & 7 \\ 8 & 0 & 2 & 15 \\ 5 & 8 & 0 & 1 \\ 2 & 5 & \infty & 0 \end{pmatrix} \end{array}$$

Optimizing Network Routing

Example (Cont.)

2. Vertex 2 as intermediate vertex $\Rightarrow$ 2^{nd} Column and 2^{nd} Line remain unchanged.

$$\begin{array}{ccccc} d(1,3) & & d(1,2) & + & d(2,3) \\ \infty & > & 3 & + & 2 \\ d(1,4) & & d(1,2) & + & d(2,4) \\ 7 & < & 3 & + & 15 \\ \dots & \dots & \dots & \dots & \dots \end{array}$$

$$\mathrm{d} = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{pmatrix} 0 & 3 & 5 & 7 \\ 8 & 0 & 2 & 15 \\ 5 & 8 & 0 & 1 \\ 2 & 5 & 7 & 0 \end{pmatrix} \end{array}$$

Optimizing Network Routing

Example (Cont.)

3. Vertex 3 as intermediate vertex $\Rightarrow$ 3^{rd} Column and 3^{rd} Line remain unchanged.

$$\begin{array}{ccccc} d(1,2) & & d(1,3) & + & d(3,2) \\ 3 & < & 5 & + & 8 \\ \cdots & \cdots & \cdots & \cdots & \cdots \end{array}$$

$$\mathrm{d} = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{pmatrix} 0 & 3 & 5 & 6 \\ 7 & 0 & 2 & 3 \\ 5 & 8 & 0 & 1 \\ 2 & 5 & 7 & 0 \end{pmatrix} \end{array}$$

Optimizing Network Routing

Example (Cont.)

4. Vertex 4 as intermediate vertex $\Rightarrow$ 4^{th} Column and 4^{th} Line remain unchanged.

$$d = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{pmatrix} 0 & 3 & 5 & 6 \\ 5 & 0 & 2 & 3 \\ 3 & 6 & 0 & 1 \\ 2 & 5 & 7 & 0 \end{pmatrix} \end{array}$$

Optimizing Network Routing

Example (Cont.)

- Final Distance Matrix : After the algorithm completes, the distance matrix contains the shortest paths between all pairs of vertices.
- The final matrix tells you the shortest distance, and we can reconstruct the path by manually checking possible intermediate vertices that lead to the shortest distance or by using a predecessor matrix for easier path reconstruction.

$$d = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{pmatrix} 0 & 3 & 5 & 6 \\ 5 & 0 & 2 & 3 \\ 3 & 6 & 0 & 1 \\ 2 & 5 & 7 & 0 \end{pmatrix} \end{array}$$

Optimizing Network Routing

Time Complexity

- $\mathcal{O}(V^3)$, where V is the number of vertices.

Optimizing Network Routing

Comparison of Algorithms (Part 1)

Feature	Bellman-Ford	Dijkstra	Floyd-Warshall
Purpose	Shortest paths from a single source vertex	Shortest paths from a single source vertex	Shortest paths between all pairs of vertices
Graph Type	Can handle negative weights and detect negative-weight cycles	Assumes non-negative edge weights	Can handle negative weights but not negative-weight cycles

Optimizing Network Routing

Comparison of Algorithms (Part 2)

Feature	Bellman-Ford	Dijkstra	Floyd-Warshall
Time Complexity	$\mathcal{O}(V \cdot E)$	$\mathcal{O}(V^2)$ with an adjacency matrix or $\mathcal{O}((E+V)\log V)$ using adjacency list with a min-heap priority queue	$\mathcal{O}(V^3)$

Optimizing Network Routing

Comparison of Algorithms (Part 3)

Feature	Bellman-Ford	Dijkstra	Floyd-Warshall
Relaxation Process	Relax all edges up to $V - 1$ times	Relax edges based on the vertex with the smallest distance	Relax all edges for each vertex
Handling of Negative Weights	Yes	No	Yes

Optimizing Network Routing

Comparison of Algorithms (Part 4)

Feature	Bellman-Ford	Dijkstra	Floyd-Warshall
Negative Cycle Detection	Yes	No	No
Algorithm Type	Dynamic Programming	Greedy Algorithm	Dynamic Programming

Optimizing Network Routing

How Bellman-Ford algorithm is leveraged in the RIP

RIP Protocol Overview :

- Distance Vector Routing : RIP is based on the Bellman-Ford algorithm.
- It is a distance-vector routing protocol.
- Each router maintains a routing table with the best-known distances to each destination and the next hop to reach that destination.

Optimizing Network Routing

How Bellman-Ford algorithm is leveraged in the RIP

RIP Protocol Overview (Cont) :

- Periodic Updates : Routers share their routing tables with their neighbors periodically (every 30 seconds by default) or when there is a significant change in the network topology.
- Please note that each router periodically shares its entire routing table with its neighbors.
- Hop Count as Metric : RIP uses hop count as a metric to measure distance, with a maximum allowable hop count of 15 (to prevent routing loops).

Optimizing Network Routing

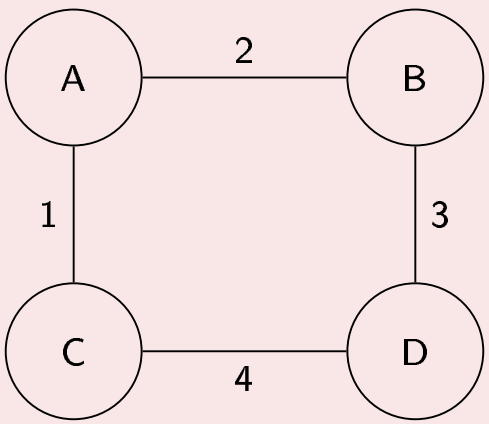
How Bellman-Ford algorithm is leveraged in the RIP

- A-B has a cost of 2
- A-C has a cost of 1
- B-D has a cost of 3
- C-D has a cost of 4

Optimizing Network Routing

How Bellman-Ford algorithm is leveraged in the RIP

Let's consider a small network of routers with the following topology :



Optimizing Network Routing

How Bellman-Ford algorithm is leveraged in the RIP

1. Initial State :
   - We are going to focus on router B and try to fill in its routing table.
   - Initially, routers A, B, C, and D only know the direct connections to their respective neighbors :

Optimizing Network Routing

How Bellman-Ford algorithm is leveraged in the RIP

Table 7 – A's Table

Destination	Cost	Next hop
A	0	-
B	2	B (Direct)
C	1	C (Direct)
D	∞	-

Optimizing Network Routing

How Bellman-Ford algorithm is leveraged in the RIP

Table 8 – B's Table

Destination	Cost	Next hop
A	2	A (Direct)
B	0	-
C	∞	-
D	3	D

Optimizing Network Routing

How Bellman-Ford algorithm is leveraged in the RIP

Table 9 – C's Table

Destination	Cost	Next hop
A	1	A (Direct)
B	∞	-
C	0	-
D	4	D

Optimizing Network Routing

How Bellman-Ford algorithm is leveraged in the RIP

Table 10 – D's Table

Destination	Cost	Next hop
A	∞	-
B	3	D
C	4	C
D	0	-

Optimizing Network Routing

How Bellman-Ford algorithm is leveraged in the RIP

2. Step 1 : First Exchange of Tables. Receiving Information from Neighbors.

   Router B will receive updates from its neighbors A and D about their respective costs to reach other routers.

   - A's table :
     - Destination A : Cost 0, Next Hop -
     - Destination B : Cost 2, Next Hop B
     - Destination C : Cost 1, Next Hop C
     - Destination D : Cost ∞, Next Hop -

Optimizing Network Routing

How Bellman-Ford algorithm is leveraged in the RIP

2. Step 1 : First Exchange of Tables. Receiving Information from Neighbors.

   Router B will receive updates from its neighbors A and D about their respective costs to reach other routers.

   - D's table :
     - Destination A : Cost ∞, Next Hop -
     - Destination B : Cost 3, Next Hop B
     - Destination C : Cost 4, Next Hop C
     - Destination D : Cost 0 Next Hop -

Optimizing Network Routing

How Bellman-Ford algorithm is leveraged in the RIP

3. Step 2 : Router B receives Router A's table and updates its own table. Router B uses the Bellman-Ford algorithm to update its table based on the information received. It processes the information from Router A as follows :
   - Destination A :
     - Current Cost in B's table : 2 (directly connected to A)
     - New Cost via A : 2 (Cost from B to A) + 0 (A's cost to A) = 2
     - No Update Needed (already has a cost of 2 via direct link).

Optimizing Network Routing

How Bellman-Ford algorithm is leveraged in the RIP

3. Step 2 : Router B receives Router A's table and updates its own table. Router B uses the Bellman-Ford algorithm to update its table based on the information received. It processes the information from Router A as follows :
   - Destination B :
     - Current Cost in B's table : 0 (itself)
     - New Cost via A : 2 (Cost from B to A) + 2 (A's cost to B) = 4
     - No Update Needed (B's cost to itself remains 0).

Optimizing Network Routing

How Bellman-Ford algorithm is leveraged in the RIP

3. Step 2 : Router B receives Router A's table and updates its own table. Router B uses the Bellman-Ford algorithm to update its table based on the information received. It processes the information from Router A as follows :
   - Destination C :
     - Current Cost in B's table : ∞ (initially unknown)
     - New Cost via A : 2 (Cost from B to A) + 1 (A's cost to C) = 3
     - Update : B learns that it can reach C via A with a cost of 3.

Optimizing Network Routing

How Bellman-Ford algorithm is leveraged in the RIP

3. Step 2 : Router B receives Router A's table and updates its own table. Router B uses the Bellman-Ford algorithm to update its table based on the information received. It processes the information from Router A as follows :
   - Destination D :
     - Current Cost in B's table : 3 (directly connected to D)
     - New Cost via A : 2 (Cost from B to A) + ∞ (A's cost to D) = ∞
     - No Update Needed (B's direct connection to D with a cost of 3 remains better).

Optimizing Network Routing

How Bellman-Ford algorithm is leveraged in the RIP

Table 11 – Updated Routing Table for B (After receiving A's table)

Destination	Cost	Next hop
A	2	A
B	0	-
C	3	A
D	3	D

Optimizing Network Routing

How Bellman-Ford algorithm is leveraged in the RIP

3. Step 2 : Router B receives Router D's table and updates its own table. Router B uses the Bellman-Ford algorithm to update its table based on the information received. It processes the information from Router D as follows :
   - Destination A :
     - Current Cost in B's table : 2 (directly connected to A)
     - New Cost via D : 3 (Cost from B to D) + *infty* (D's cost to A) = ∞
     - No Update Needed (B's direct connection to A with a cost of 2 remains better.

Optimizing Network Routing

How Bellman-Ford algorithm is leveraged in the RIP

3. Step 2 : Router B receives Router D's table and updates its own table. Router B uses the Bellman-Ford algorithm to update its table based on the information received. It processes the information from Router D as follows :
   - Destination B :
     - Current Cost in B's table : 0 (itself)
     - New Cost via D : 3 (Cost from B to D) + 3 (D's cost to B) = 6
     - No Update Needed (B's cost to itself remains 0).

Optimizing Network Routing

How Bellman-Ford algorithm is leveraged in the RIP

3. Step 2 : Router B receives Router D's table and updates its own table. Router B uses the Bellman-Ford algorithm to update its table based on the information received. It processes the information from Router D as follows :
   - Destination C :
     - Current Cost in B's table : 2 (through A)
     - New Cost via D : 3 (Cost from B to D) + 4 (D's cost to C) = 7
     - B learns that it can reach C via D with a cost of 7. (No update, as $3 < 7$)

Optimizing Network Routing

How Bellman-Ford algorithm is leveraged in the RIP

3. Step 2 : Router B receives Router D's table and updates its own table. Router B uses the Bellman-Ford algorithm to update its table based on the information received. It processes the information from Router D as follows :
   - Destination D :
     - Current Cost in B's table : 3 (directly connected to D)
     - New Cost via D : 3 (Cost from B to D) + 0 (D's cost to D) = 3
     - No Update Needed (already has a cost of 3 via direct link).

Optimizing Network Routing

How Bellman-Ford algorithm is leveraged in the RIP

Table 12 – Updated Routing Table for B (After receiving D's table)

Destination	Cost	Next hop
A	2	A
B	0	-
C	3	A
D	3	D

Optimizing Network Routing

How Bellman-Ford algorithm is leveraged in the RIP

- **Iteration and Convergence :** This process repeats at each router, and each router continually updates its table as it receives new information from its neighbors. Eventually, all routers in the network will converge on a stable set of paths (i.e., the routing tables will no longer change).
- **Convergence Detection :** The convergence point is reached when all routers' tables stabilize and no longer update based on the distance vectors exchanged by neighbors.

Optimizing Network Routing

How OSPF uses Dijkstra's algorithm

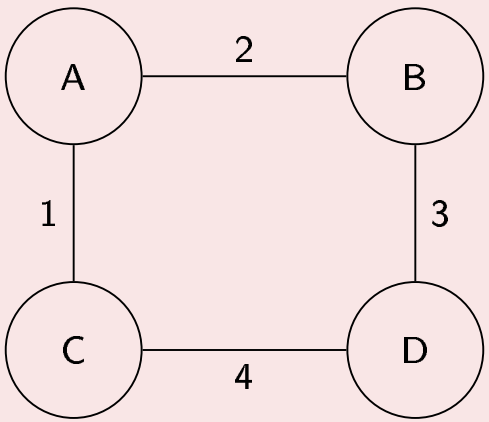
- Open Shortest Path First (OSPF) is a widely used link-state routing protocol in IP networks.
- OSPF uses Dijkstra's algorithm to calculate the shortest path tree (SPT) for each router, which helps determine the best route to each destination within the network.

Optimizing Network Routing

How OSPF uses Dijkstra's algorithm

Consider a small OSPF network with four routers :

- Router A is connected to Router B with a cost of 2.
- Router A is connected to Router C with a cost of 1.
- Router B is connected to Router D with a cost of 3.
- Router C is connected to Router D with a cost of 4.



Optimizing Network Routing

How OSPF uses Dijkstra's algorithm

1. Step 1 : Link-State Advertisement (LSA) Each router broadcasts its link-state information to all other routers in the OSPF area. The information includes the router's directly connected neighbors and the cost of each link.
   - Router A advertises :
     - A-B with a cost of 2
     - A-C with a cost of 1
   - Router B advertises :
     - B-A with a cost of 2
     - B-D with a cost of 3

Optimizing Network Routing

How OSPF uses Dijkstra's algorithm

1. Step 1 : Link-State Advertisement (LSA) Each router broadcasts its link-state information to all other routers in the OSPF area. The information includes the router's directly connected neighbors and the cost of each link.
   - Router C advertises :
     - C-A with a cost of 1
     - C-D with a cost of 4
   - Router D advertises :
     - D-B with a cost of 3
     - D-C with a cost of 4

Optimizing Network Routing

How OSPF uses Dijkstra's algorithm

2. Step 2 : Building the Link-State Database (LSDB) Each router collects all LSAs and builds the Link-State Database (LSDB), which represents the network's topology.

Optimizing Network Routing

How OSPF uses Dijkstra's algorithm

3. Step 3 : Running Dijkstra's Algorithm Now, each router runs Dijkstra's algorithm on the LSDB to calculate the shortest path tree (SPT) to all other routers.

Optimizing Network Routing

How OSPF uses Dijkstra's algorithm

Example : Dijkstra's Algorithm on Router A

1. Initial Setup :
   - Router A initializes the distance to itself as 0.
   - Distances to all other routers are set to ∞.
   - The set S (processed routers) is initially empty.

Optimizing Network Routing

How OSPF uses Dijkstra's algorithm

Initial Distances :

- Distance to A : 0
- Distance to B : ∞
- Distance to C : ∞
- Distance to D : ∞

Optimizing Network Routing

How OSPF uses Dijkstra's algorithm

1. Step 1 : Process Router A
   - Current Router : A
   - Neighbors : B, C
   - Update the distance estimates :
     - Distance to B : $\min(\infty, 0 + 2) = 2$ (via A)
     - Distance to C : $\min(\infty, 0 + 1) = 1$ (via A)
     - Add A to the set S.

Optimizing Network Routing

How OSPF uses Dijkstra's algorithm

1. Step 1 : Process Router A
   - Updated Distances :
     - Distance to A : 0
     - Distance to B : 2
     - Distance to C : 1
     - Distance to D : ∞
     - Set S : A

Optimizing Network Routing

How OSPF uses Dijkstra's algorithm

2. Step 2 : Process Router C (Next closest unprocessed router)
   - Current Router : C
   - Neighbors : A, D
   - Update the distance estimates :
     - Distance to D : $\min(\infty, 1 + 4) = 5$ (via C)
     - Add C to the set S.

Optimizing Network Routing

How OSPF uses Dijkstra's algorithm

2. Step 2 : Process Router C (Next closest unprocessed router)
   - Updated Distances :
     - Distance to A : 0
     - Distance to B : 2
     - Distance to C : 1
     - Distance to D : 5
     - Set S : A, C

Optimizing Network Routing

How OSPF uses Dijkstra's algorithm

3. Step 3 : Process Router B (Next closest unprocessed router)
   - Current Router : B
   - Neighbors : A, D
   - Update the distance estimates :
     - Distance to D : $\min(5, 2 + 3) = 5$ (via B) (no change)
     - Add B to the set S.

Optimizing Network Routing

How OSPF uses Dijkstra's algorithm

3. Step 3 : Process Router B (Next closest unprocessed router)
   - Updated Distances :
     - Distance to A : 0
     - Distance to B : 2
     - Distance to C : 1
     - Distance to D : 5
     - Set S : A, C, B

Optimizing Network Routing

How OSPF uses Dijkstra's algorithm

4. Step 4 : Process Router D
   - Current Router : D
   - Neighbors : B, C
   - No further updates are needed as all shortest paths have been found.
   - Final Shortest Path Tree from Router A :
     - Router A to Router B : Cost 2 (via A)
     - Router A to Router C : Cost 1 (via A)
     - Router A to Router D : Cost 5 (via either C or B)

Optimizing Network Routing

How OSPF uses Dijkstra's algorithm

4. Step 4 : Updating the Routing Table
   - Router A will now have the following routing table :
     - Destination B : Next hop B, Cost 2
     - Destination C : Next hop C, Cost 1
     - Destination D : Next hop C or B, Cost 5