

# Lecture 11

## Shortest Paths

Department of Computer Science  
Hofstra University

# Lecture Goals

- In this lecture we study **shortest-paths** problems. We begin by analyzing some basic properties of shortest paths and a generic algorithm for the problem.
- We introduce and analyze **Dijkstra's algorithm** for shortest-paths problems with nonnegative weights.
- Next, we consider **Topological Sort** for Edge-weighted DAG, which works even if the weights are negative.
- We conclude with the **Bellman–Ford** algorithm for edge-weighted digraphs with no negative cycles.
- We conclude with the Floyd Warshall Algorithm for all-pairs shortest path

# Shortest Paths in an Edge-weighted Digraph

Given an edge-weighted digraph, find the shortest path from  $s$  to  $t$ .

edge-weighted digraph

4→5	0.35
5→4	0.35
4→7	0.37
5→7	0.28
7→5	0.28
5→1	0.32
0→4	0.38
0→2	0.26
7→3	0.39
1→3	0.29
2→7	0.34
6→2	0.40
3→6	0.52
6→0	0.58
6→4	0.93



shortest path from 0 to 6

0→2	0.26
2→7	0.34
7→3	0.39
3→6	0.52

## Variants

### ❖ Which vertices?

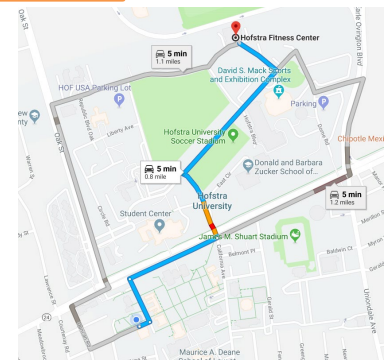
- Single source: from one vertex  $s$  to every other vertex.
- Source-sink: from one vertex  $s$  to another  $t$ .
- All pairs: between all pairs of vertices.

### ❖ Nonnegative weights?

### ❖ Cycles?

- Negative cycles.

Can we use BFS?



Simplifying assumption: Each vertex is reachable from  $s$ .

# Weighted Directed Edge API

```
public class DirectedEdge
```

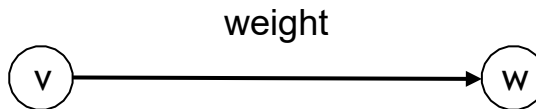
```
    DirectedEdge(int v, int w, double weight) //weighted edge v->w
```

```
    int from() // vertex v
```

```
    int to() // vertex w
```

```
    double weight() // the weight
```

```
    String toString() // string representation
```



Idiom for processing an edge *e*: `int v = e.from(), w = e.to();`

# Weighted Edge: Java Implementation

```
public class DirectedEdge
{
    private final int v, w;
    private final double weight;

    public DirectedEdge(int v, int w, double weight)
    {
        this.v = v;
        this.w = w;
        this.weight = weight;
    }

    public int from()
    { return v; }

    public int to()
    { return w; }

    public int weight()
    { return weight; }
}
```

# Edge-Weighted Graph API

```
public class EdgeWeightedDigraph
```

```
    EdgeWeightedDigraph(int V) // edge-weighted digraph with V vertices
```

```
    void addEdge(DirectedEdge e) // add weighted directed edge e
```

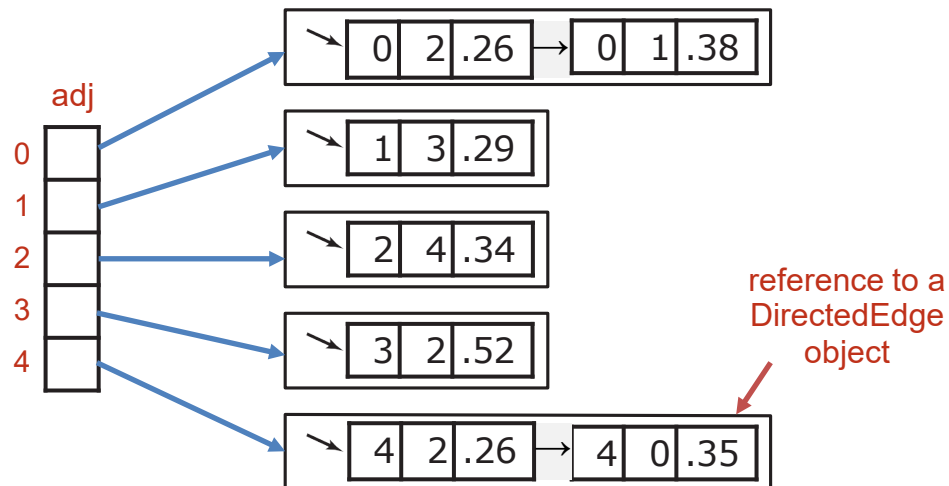
```
    Iterable<DirectedEdge> adj(int v) // edges pointing from v
```

```
    Iterable<DirectedEdge> edges() // all edges in this graph
```

```
    int V() // number of vertices
```

```
    int E() // number of edges
```

```
    String toString() // string representation
```



# Edge-Weighted Digraph: Adjacency-Lists Implementation

```
public class EdgeWeightedDigraph
{
    private final int V;
    private final List<DirectedEdge>[] adj;

    public EdgeWeightedDigraph (int V)
    {
        this.V = V;
        adj = (List<DirectedEdge>[]) new ArrayList[V];
        for (int v = 0; v < V; v++)
            adj[v] = new ArrayList<DirectedEdge>();
    }

    public void addEdge(DirectedEdge e)
    {
        int v = e.from();
        adj[v].add(e);
    }

    public Iterable<DirectedEdge> adj(int v)
    {
        return adj[v];
    }
}
```

← add edge  $e = v \rightarrow w$  to  
only  $v$ 's adjacency lists

# Single-source Shortest Paths API

**Goal.** Find the shortest path from  $s$  to every other vertex.

```
public class SP
```

```
    SP(EdgeWeightedGraph G, int s) // shortest paths from s in graph G
```

```
    double distTo(int v) // length of shortest path from s to v
```

```
    Iterable<DirectedEdge> pathTo(int v) // shortest path from s to v
```

```
SP sp = new SP(G, s);
for (int v = 0; v < G.V(); v++)
{
    StdOut.printf("%d to %d (0.2f): ", s, v, sp.distTo(v));
    for (DirectedEdge e : sp.pathTo(v)) {
        StdOut.print(e + " ");
        StdOut.println();
    }
}
```

```
0 to 0 (0.00):
0 to 1 (1.05): 0->4 0.38 4->5 0.35 5->1 0.32
0 to 2 (0.26): 0->2 0.26
0 to 3 (0.99): 0->2 0.26 2->7 0.34 7->3 0.39
0 to 4 (0.38): 0->4 0.38
0 to 5 (0.73): 0->4 0.38 4->5 0.35
0 to 6 (1.51): 0->2 0.26 2->7 0.34 7->3 0.39 3->6 0.52
0 to 7 (0.60): 0->2 0.26 2->7 0.34
```



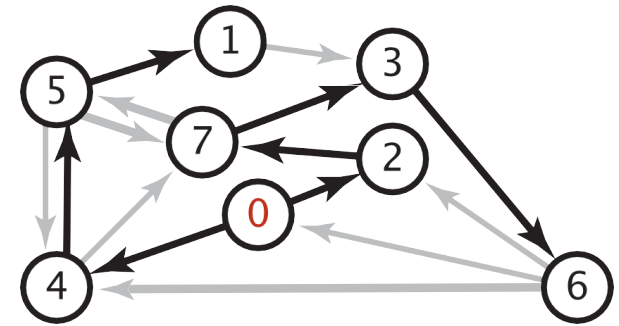
# Data Structures for Single-source Shortest Paths

**Goal.** Find the shortest path from  $s$  to every other vertex.

**Observation.** A shortest-paths tree (SPT) solution exists.

**Consequence.** Can represent the SPT with two vertex-indexed arrays:

- $\text{distTo}[v]$  is length of shortest path from  $s$  to  $v$ .
- $\text{edgeTo}[v]$  is last edge on shortest path from  $s$  to  $v$ .



shortest-paths tree from 0

	edgeTo[]	distTo[]
0	null	0
1	5->1 0.32	1.05
2	0->2 0.26	0.26
3	7->3 0.37	0.97
4	0->4 0.38	0.38
5	4->5 0.35	0.73
6	3->6 0.52	1.49
7	2->7 0.34	0.60

parent-link representation

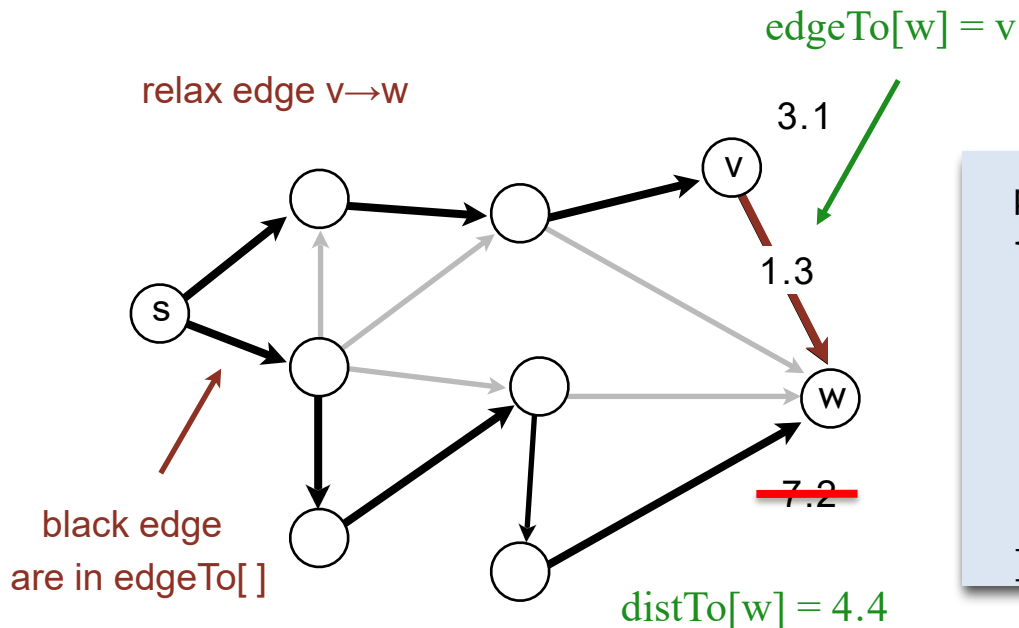
```
public double distTo(int v)
{ return distTo[v]; }

public Iterable<DirectedEdge> pathTo(int v)
{
    Stack<DirectedEdge> path = new Stack<DirectedEdge>();
    for (DirectedEdge e = edgeTo[v]; e != null; e = edgeTo[e.from()])
        path.push(e);
    return path;
}
```

# Edge Relaxation

Relax edge  $e = v \rightarrow w$ . (basic of building SPT)

- $\text{distTo}[v]$  is length of shortest **known** path from  $s$  to  $v$ .
- $\text{distTo}[w]$  is length of shortest **known** path from  $s$  to  $w$ .
- $\text{edgeTo}[w]$  is last edge on shortest **known** path from  $s$  to  $w$ .
- If  $e = v \rightarrow w$  gives shorter path to  $w$  through  $v$ , update  $\text{distTo}[w]$  and  $\text{edgeTo}[w]$ .



```
private void relax(DirectedEdge e)
{
    int v = e.from(), w = e.to();
    if (distTo[w] > distTo[v] + e.weight())
    {
        distTo[w] = distTo[v] + e.weight();
        edgeTo[w] = e;
    }
}
```

# Generic Shortest-paths Algorithm

## Generic algorithm (to compute SPT from $s$ )

---

**For each vertex  $v$ :  $\text{distTo}[v] = \infty$ .**

**For each vertex  $v$ :  $\text{edgeTo}[v] = \text{null}$ .**

**$\text{distTo}[s] = 0$ .**

**Repeat until done:**

**- Relax any edge.**

---

**Proposition.** Generic algorithm computes SPT (if it exists) from  $s$ .

**Pf.**

- Throughout algorithm,  $\text{distTo}[v]$  is the length of a simple path from  $s$  to  $v$  (and  $\text{edgeTo}[v]$  is last edge on path).
- Each successful relaxation decreases  $\text{distTo}[v]$  for some  $v$ .
- The entry  $\text{distTo}[v]$  can decrease at most a finite number of times.

**Efficient implementations.** How to choose which edge to relax?

- Ex 1. Dijkstra's algorithm. (**nonnegative weights, directed cycles**).
- Ex 2. Topological sort algorithm. (**no directed cycles**).
- Ex 3. Bellman–Ford algorithm. (**no negative cycles**).

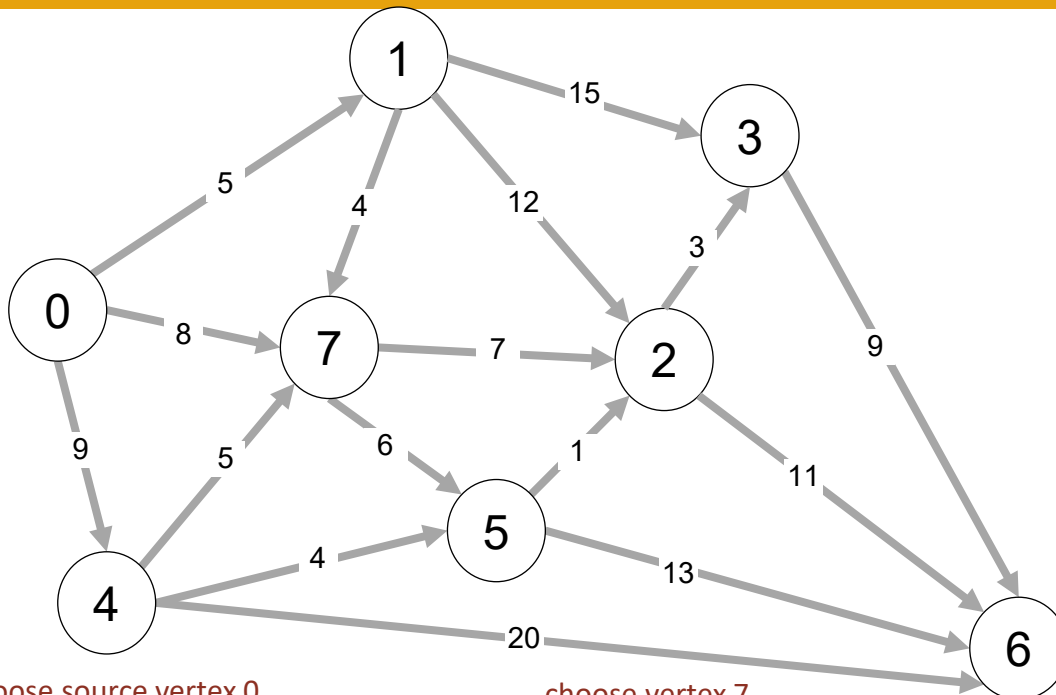
# Dijkstra's Algorithm

1. Mark the source node with a current distance of 0 and the rest with infinity.
2. Set the non-visited node with the smallest current distance as the current node.
3. For each neighbor, N of the current node add the current distance of the adjacent node with the weight of the edge connecting 0- $\rightarrow$ 1. If it is smaller than the current distance of Node, set it as the new current distance of N.
4. Mark the current node 1 as visited.
5. Go to step 2 if there are any nodes are unvisited.

# Dijkstra's Algorithm

- Consider vertices in increasing order of distance from s
  - (non-tree vertex with the lowest distTo[ ] value).
- Add vertex to tree and relax all edges pointing from that vertex.

choose vertex 5  
 relax all edges adjacent from 5  
 choose vertex 2  
 relax all edges adjacent from 2  
 choose vertex 3  
 relax all edges adjacent from 3  
 choose vertex 6  
 relax all edges adjacent from 6



choose source vertex 0  
 relax all edges adjacent from 0  
 choose vertex 1  
 relax all edges adjacent from 1

choose vertex 7  
 relax all edges adjacent from 7  
 choose vertex 4  
 relax all edges adjacent from 4



v distTo[]			
0	<del>∞</del>	0	
1	<del>∞</del>	5	
2	<del>∞</del>	<del>17</del>	<del>15</del> 14
3	<del>∞</del>	<del>20</del>	17
4	<del>∞</del>	9	
5	<del>∞</del>	<del>14</del>	13
6	<del>∞</del>	<del>29</del>	<del>26</del> 25
7	<del>∞</del>	8	

v edgeTo[]			
0	-		
1	<del>-</del>	0	
2	<del>-</del>	<del>1</del>	<del>7</del> 5
3	<del>-</del>	<del>1</del>	2
4	<del>-</del>	0	
5	<del>-</del>	<del>7</del>	4
6	<del>-</del>	<del>4</del>	<del>5</del> 2
7	<del>-</del>	0	

# Dijkstra's Algorithm: Correctness Proof

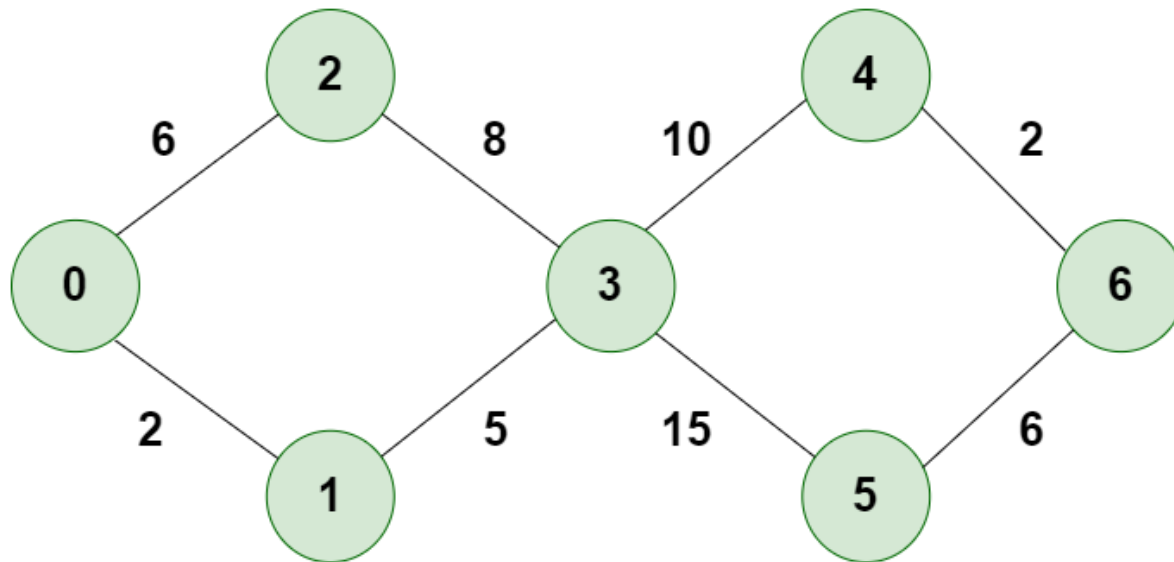
**Proposition.** Dijkstra's algorithm computes a SPT in any edge-weighted digraph with nonnegative weights.

**Pf.**

- Each edge  $e = v \rightarrow w$  is relaxed exactly once (when  $v$  is relaxed),
  - leaving  $\text{distTo}[w] \leq \text{distTo}[v] + e.\text{weight}()$ .
- Inequality holds until algorithm terminates because:
  - $\text{distTo}[w]$  cannot increase   $\text{distTo}[ ]$  values are monotone decreasing
  - $\text{distTo}[v]$  will not change  we choose lowest  $\text{distTo}[ ]$  value at each step (and edge weights are nonnegative)
- Thus, upon termination, shortest-paths optimality conditions hold.

# Dijkstra's Algorithm Example

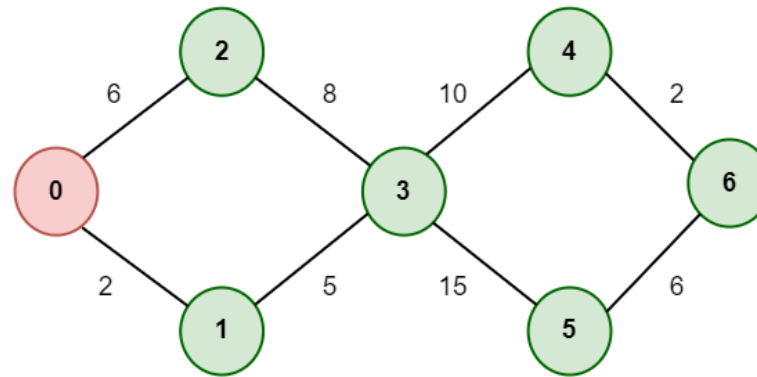
**Example Graph**



**Dijkstra's Algorithm**

## STEP 1

Start from Node 0 and mark Node 0 as Visited and check for adjacent nodes



Unvisited Nodes

{0,1,2,3,4,5,6}

Distance:

0: 0 ✓

1: ∞

2: ∞

3: ∞

4: ∞

5: ∞

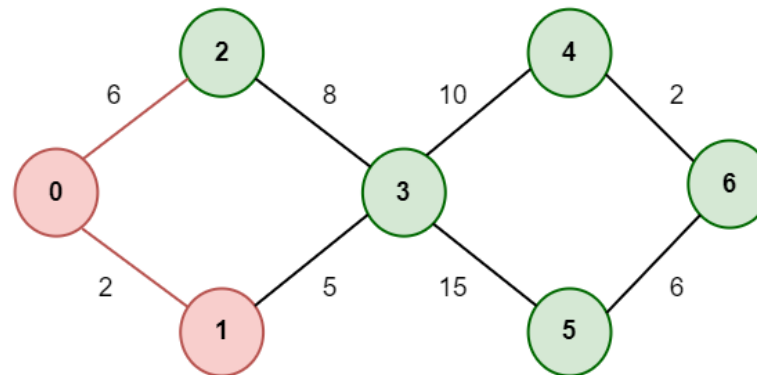
6: ∞

Dijkstra's Algorithm

Step 1: Start from Node 0 and mark Node as visited

## STEP 2

Mark Node 1 as Visited and add the Distance



Unvisited Nodes

{0,1,2,3,4,5,6}

Distance:

0: 0 ✓

1: 2 ✓

2: ∞

3: ∞

4: ∞

5: ∞

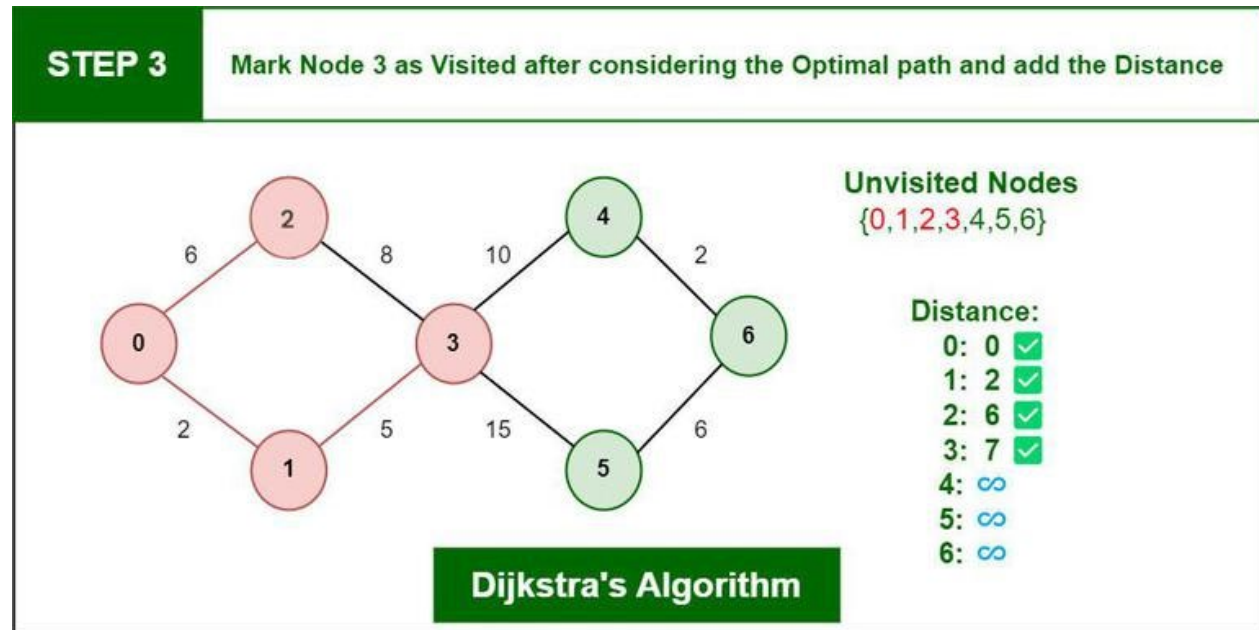
6: ∞

Dijkstra's Algorithm

Step 2: Check for adjacent Nodes, and choose Node with minimum distance. In this step Node 1 is Minimum distance adjacent Node, so marked it as visited and add up the distance.

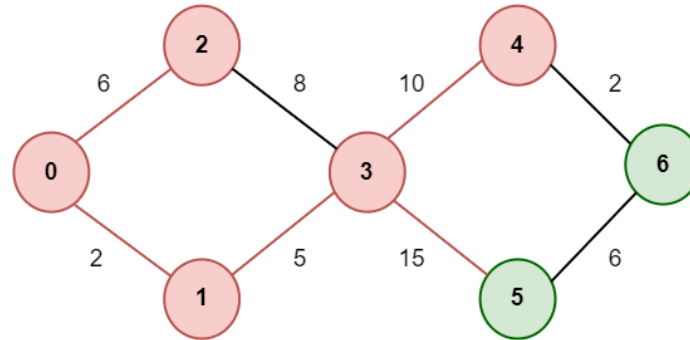


- Step 3: Then Move Forward and check for adjacent Node which is Node 3, so marked it as visited and add up the distance, Now the distance will be:
- Distance: Node 0  $\rightarrow$  Node 1  $\rightarrow$  Node 3 =  $2 + 5 = 7$



#### STEP 4

Mark Node 4 as Visited after considering the Optimal path and add the Distance



Unvisited Nodes

{0, 1, 2, 3, 4, 5, 6}

Distance:

0: 0 ✓

1: 2 ✓

2: 6 ✓

3: 7 ✓

4: 17 ✓

5: ∞

6: ∞

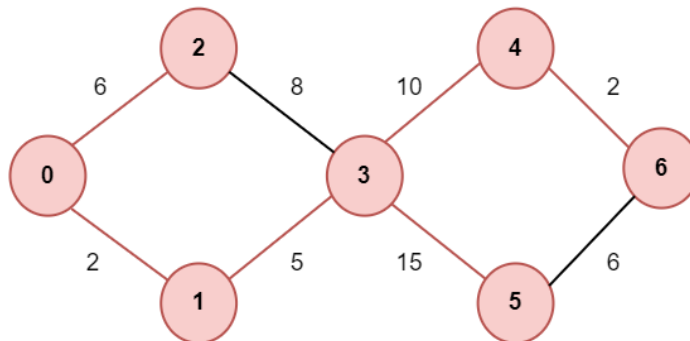
Dijkstra's Algorithm

Step 4: We choose Node 4 as Minimum distance adjacent Node, so marked it as visited and add up the distance.

Distance: Node 0 → Node 1 → Node 3 → Node 4 = 2 + 5 + 10 = 17

#### STEP 5

Mark Node 6 as Visited and add the Distance



Unvisited Nodes

{0, 1, 2, 3, 4, 5, 6}

Distance:

0: 0 ✓

1: 2 ✓

2: 6 ✓

3: 7 ✓

4: 17 ✓

5: 22 ✓

6: 19 ✓

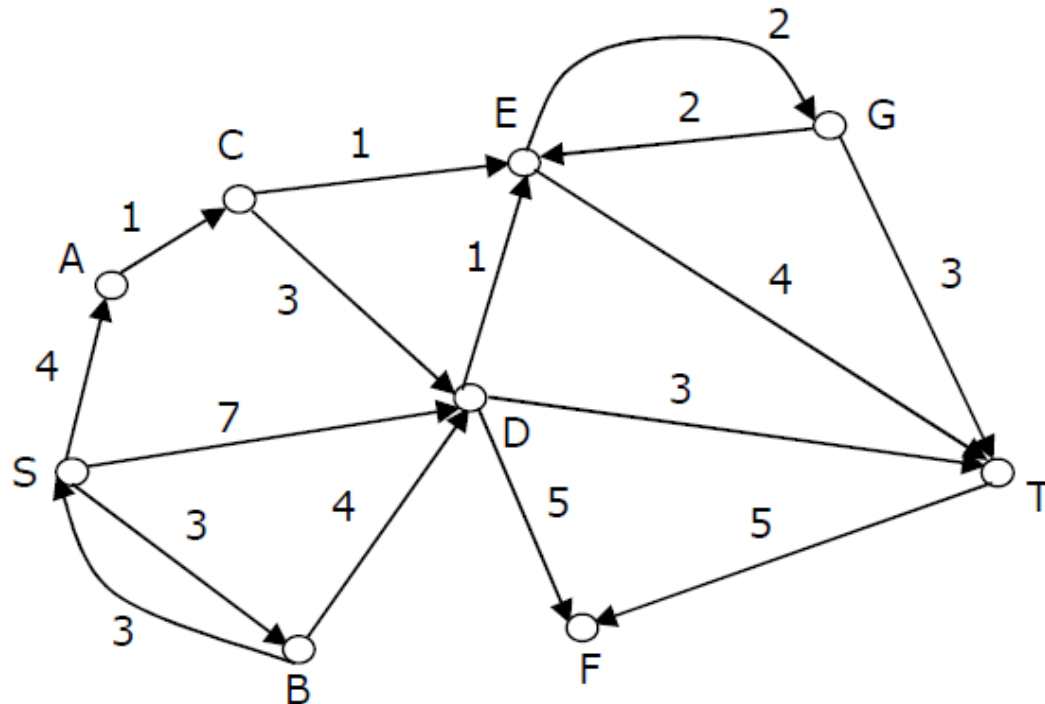
Dijkstra's Algorithm

Step 5: Check for adjacent Node 6, so marked it as visited and add up the distance, Now the distance will be:

Distance: Node 0 → Node 1 → Node 3 → Node 4 → Node 6 = 2 + 5 + 10 + 2 = 19

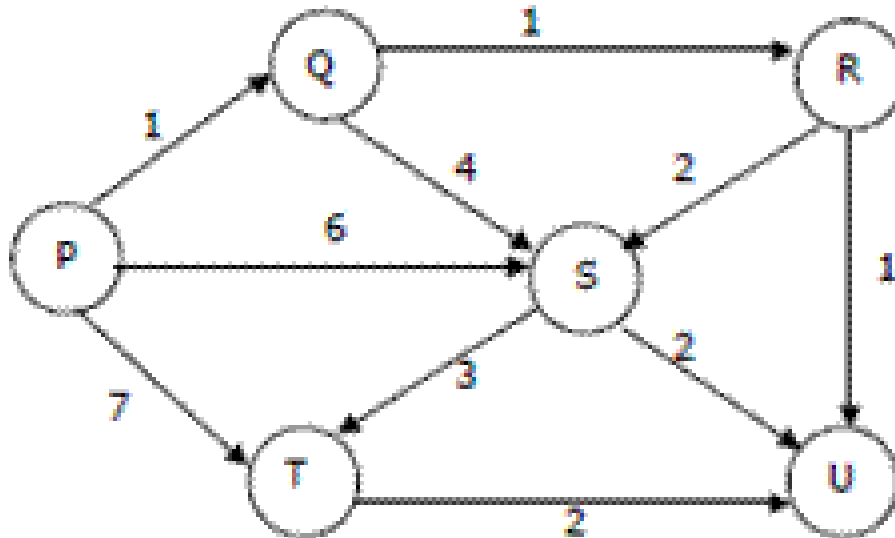
# Quiz: Dijkstra's Algorithm

- Consider the directed graph shown in the figure below. There are multiple shortest paths between vertices S and T. Which one will be reported by Dijkstra's shortest path algorithm? Assume that, in any iteration, the shortest path to a vertex  $v$  is updated only when a strictly shorter path to  $v$  is discovered.
- ANS: SACET
- See Dijkstra's shortest path algorithm When the algorithm reaches vertex 'C', the distance values of 'D' and 'E' would be 7 and 6 respectively. So the next picked vertex would be 'E'



# Quiz: Dijkstra's Algorithm

- Suppose we run Dijkstra's single source shortest-path algorithm on the following edge weighted directed graph with vertex P as the source. In what order do the nodes get included into the set of vertices for which the shortest path distances are finalized?
- ANS: P, Q, R, U, S, T



# Topological Sort for Shortest Paths in Edge-weighted DAG

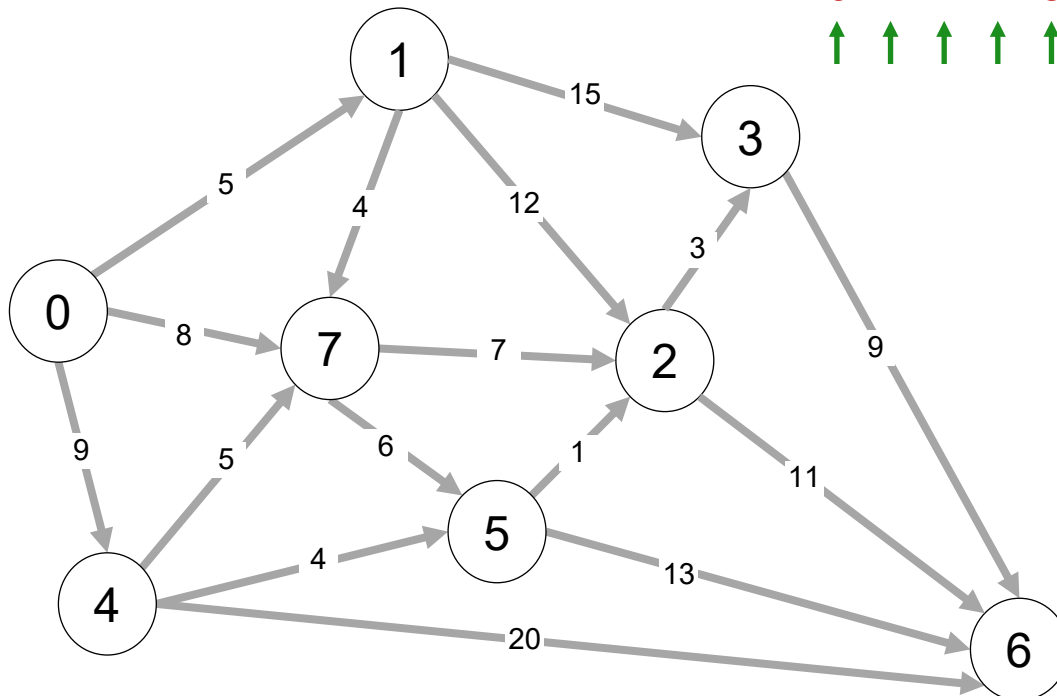
- Initialize  $\text{dist}[] = \{\text{INF}, \text{INF}, \dots\}$  and  $\text{dist}[s] = 0$  where  $s$  is the source vertex.
- Create a topological order of all vertices.
- Do following for every vertex  $u$  in topological order.
  - .....Do following for every adjacent vertex  $v$  of  $u$
  - .....if ( $\text{dist}[v] > \text{dist}[u] + \text{weight}(u, v)$ ) //relax edge  $uv$
  - ..... $\text{dist}[v] = \text{dist}[u] + \text{weight}(u, v)$
- Time Complexity: Time complexity of topological sorting is  $O(V+E)$ . After finding topological order, the algorithm process all vertices and for every vertex, it runs a loop for all adjacent vertices. Total adjacent vertices in a graph is  $O(E)$ . So the inner loop runs  $O(V+E)$  times. Therefore, overall time complexity of this algorithm is  $O(V+E)$ .

# Topological Sort for Shortest Paths in Edge-weighted DAG

Suppose that an edge-weighted digraph has no directed cycles. Is it easier to find shortest paths than in a general digraph?

Yes!

- Consider vertices in topological order.
- Relax all edges pointing from that vertex



0 1 4 7 5 2 3 6  
↑ ↑ ↑ ↑ ↑ ↑ ↑ ↑

v	distTo[]	
0	<del>∞</del>	0
1	<del>∞</del>	5
2	<del>∞</del>	<del>17</del> 15 14
3	<del>∞</del>	<del>20</del> 17
4	<del>∞</del>	9
5	<del>∞</del>	13
6	<del>∞</del>	<del>29</del> <del>26</del> 25
7	<del>∞</del>	8

v	edgeTo[]	
0	-	
1	<del>-</del>	0
2	<del>-</del>	<del>1</del> <del>7</del> 5
3	<del>-</del>	<del>1</del> 2
4	<del>-</del>	0
5	<del>-</del>	4
6	<del>-</del>	<del>4</del> <del>5</del> 2
7	<del>-</del>	0

# Shortest Paths in Edge-weighted DAG: Correctness Proof

**Proposition.** Topological sort algorithm computes SPT in any edge-weighted DAG in time proportional to  $E + V$ .

edge weights  
can be negative!

**Pf.**

- Each edge  $e = v \rightarrow w$  is relaxed exactly once (when  $v$  is relaxed),
  - leaving  $\text{distTo}[w] \leq \text{distTo}[v] + e.\text{weight}()$ .
- Inequality holds until algorithm terminates because:
  - $\text{distTo}[w]$  cannot increase      ←  $\text{distTo}[\ ]$  values are monotone decreasing
  - $\text{distTo}[v]$  will not change      ← because of topological order, no edge pointing to  $v$  will be relaxed after  $v$  is relaxed
- Thus, upon termination, shortest-paths optimality conditions hold.

# Longest Paths in Edge-weighted DAG

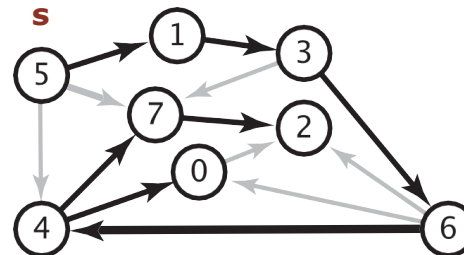
Formulate as a shortest paths problem in edge-weighted DAGs.

- Negate all weights.
  - Find shortest paths.
  - Negate weights in result.
- equivalent: reverse sense of equality in relax()

**longest paths input**

**shortest paths input**

5->4	0.35	5->4	-0.35
4->7	0.37	4->7	-0.37
5->7	0.28	5->7	-0.28
5->1	0.32	5->1	-0.32
4->0	0.38	4->0	-0.38
0->2	0.26	0->2	-0.26
3->7	0.39	3->7	-0.39
1->3	0.29	1->3	-0.29
7->2	0.34	7->2	-0.34
6->2	0.40	6->2	-0.40
3->6	0.52	3->6	-0.52
6->0	0.58	6->0	-0.58
6->4	0.93	6->4	-0.93

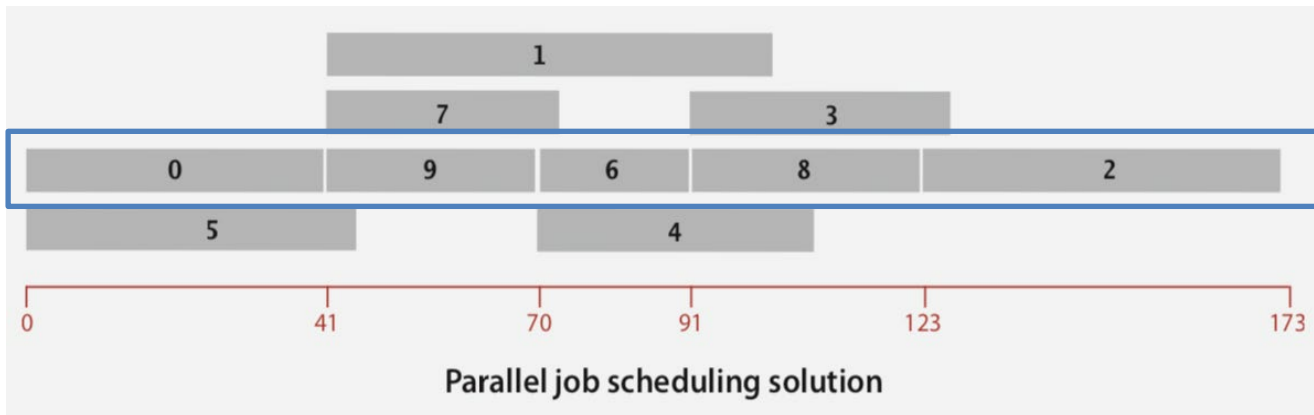


**Key point.** Topological sort algorithm works even with negative weights.



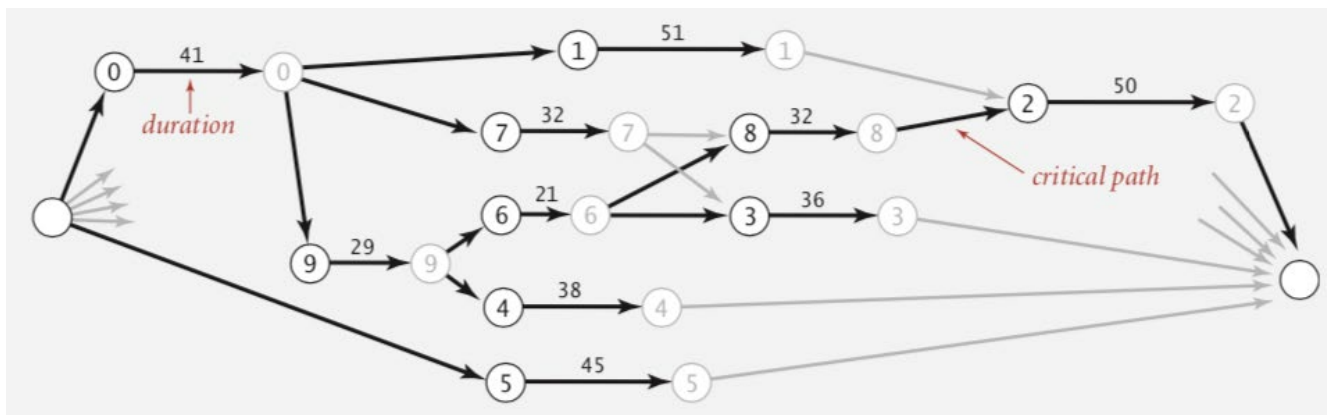
# Longest Paths in Edge-weighted DAG: Application

**Parallel job scheduling.** Given a set of jobs with durations and precedence constraints, schedule the jobs (by finding a start time for each) so as to achieve the minimum completion time, while respecting the constraints.



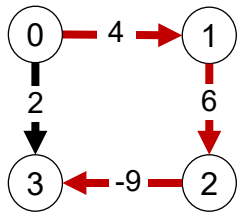
job	duration	must complete before		
0	41.0	1	7	9
1	51.0	2		
2	50.0			
3	36.0			
4	38.0			
5	45.0			
6	21.0	3	8	
7	32.0	3	8	
8	32.0	2		
9	29.0	4	6	

Use **longest path** from the source to schedule each job.

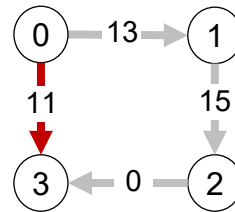


# Shortest Paths with Negative weights

**Dijkstra.** Doesn't work with negative edge weights.



Dijkstra selects vertex 3 immediately after 0.  
But shortest path from 0 to 3 is  $0 \rightarrow 1 \rightarrow 2 \rightarrow 3$ .

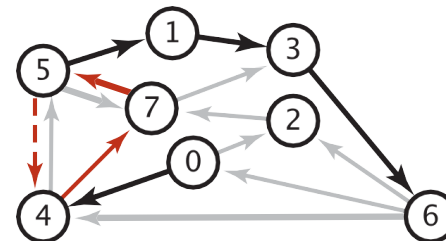


Adding 9 to each edge weight changes the  
shortest path from  $0 \rightarrow 1 \rightarrow 2 \rightarrow 3$  to  $0 \rightarrow 3$ .

**Conclusion.**  
Need a different algorithm.

**Re-weighting.** Add a constant to every edge weight doesn't work.

- A **negative cycle** is a directed cycle whose sum of edge weights is negative.
- A SPT exists **iff** no negative cycles, assuming all vertices reachable from s



**negative cycle**  $(-0.66 + 0.37 + 0.28)$

$5 \rightarrow 4 \rightarrow 7 \rightarrow 5$

**shortest path from 0 to 6**

$0 \rightarrow 4 \rightarrow 7 \rightarrow 5 \rightarrow 4 \rightarrow 7 \rightarrow 5 \dots \rightarrow 1 \rightarrow 3 \rightarrow 6$

4->5	0.35
5->4	-0.66
4->7	0.37
5->7	0.28
7->5	0.28
5->1	0.32
0->4	0.38
0->2	0.26
7->3	0.39
1->3	0.29
2->7	0.34
6->2	0.40
3->6	0.52
6->0	0.58
6->4	0.93

# Bellman-Ford Algorithm

## Bellman-Ford algorithm

**For each vertex  $v$ :  $\text{distTo}[v] = \infty$ .**

**For each vertex  $v$ :  $\text{edgeTo}[v] = \text{null}$ .**

**$\text{distTo}[s] = 0$ .**

**Repeat  $V-1$  times:**

**- Relax each edge.**

```
for (int i = 1; i < G.V(); i++)  
    for (int v = 0; v < G.V(); v++)  
        for (DirectedEdge e : G.adj(v))  
            relax(e);
```

relaxing edges  $V-1$  times in the Bellman-Ford algorithm guarantees that the algorithm has explored all possible paths of length up to  $V-1$ , which is the maximum possible length of a shortest path in a graph with  $V$  vertices. This allows the algorithm to correctly calculate the shortest paths from the source vertex to all other vertices, given that there are no negative-weight cycles.

Time complexity for connected graph: Best Case:  $O(E)$ , when distance array after 1st and 2nd relaxation are same, we can simply stop further processing

Average Case:  $O(V \cdot E)$

Worst Case:  $O(V \cdot E)$

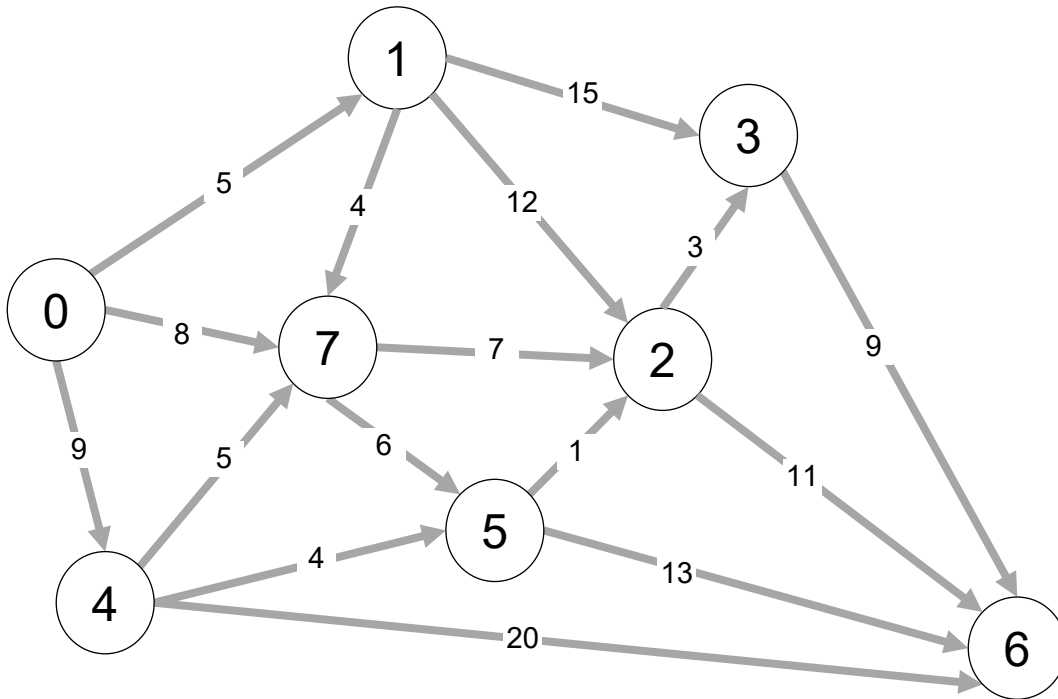
← pass  $i$  (relax each edge)

Bellman-Ford in 5 minutes — Step by step example

<https://www.youtube.com/watch?v=obWXjtg0L64>

# Bellman-Ford Algorithm

Repeat  $V - 1$  times: relax all  $E$  edges.



v	distTo[]		
0	<del>∞</del>	0	
1	<del>∞</del>	5	
2	<del>∞</del>	<del>17</del>	14
3	<del>∞</del>	<del>20</del>	17
4	<del>∞</del>	9	
5	<del>∞</del>	13	
6	<del>∞</del>	<del>28</del>	<del>26</del> 25
7	<del>∞</del>	8	

v	edgeTo[]		
0	-		
1	<del>-</del>	0	
2	<del>-</del>	<del>1</del>	5
3	<del>-</del>	<del>1</del>	2
4	<del>-</del>	0	
5	<del>-</del>	4	
6	<del>-</del>	<del>2</del>	<del>5</del> 2
7	<del>-</del>	0	

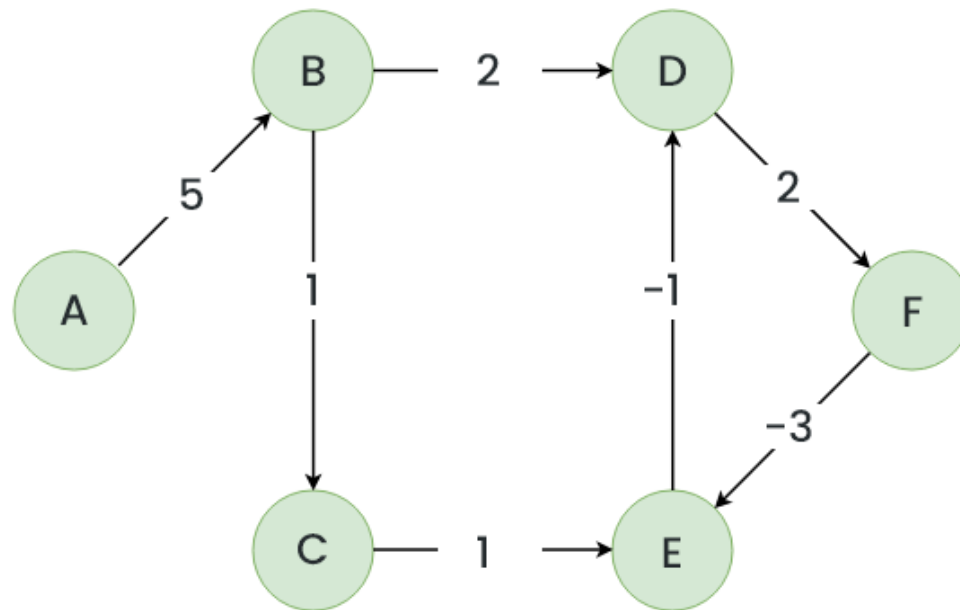
pass 1 pass 2 pass 3 (no further changes) pass 4-7 (no further changes)

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

# Algorithm to Find Negative Cycle in a Directed Weighted Graph Using Bellman-Ford

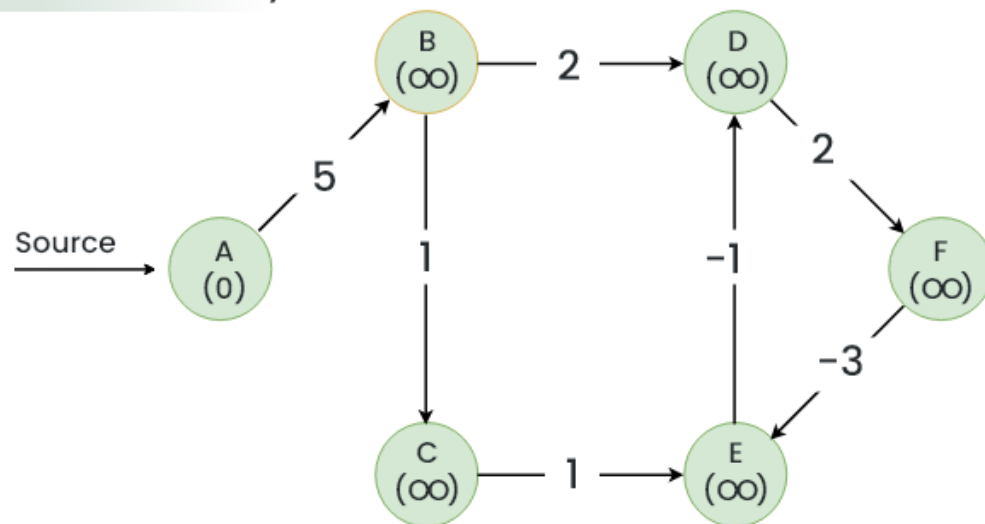
- Initialize distance array  $\text{dist}[]$  for each vertex ' $v$ ' as  **$\text{dist}[v] = \text{INFINITY}$** .
- Assume any vertex (let's say '0') as source and assign  **$\text{dist} = 0$** .
- Relax all the **edges( $u, v, \text{weight}$ )**  **$N-1$**  times as per the below condition:
  - **$\text{dist}[v] = \text{minimum}(\text{dist}[v], \text{distance}[u] + \text{weight})$**
- Now, Relax all the edges one more time i.e. the  **$N$ th** time and based on the below two cases we can detect the negative cycle:
  - Case 1 (Negative cycle exists): For any **edge( $u, v, \text{weight}$ )**, if  **$\text{dist}[u] + \text{weight} < \text{dist}[v]$**
  - Case 2 (No Negative cycle) : case 1 fails for all the edges.

# Bellman-Ford Example



- Step 1: Initialize a distance array  $\text{Dist}[]$  to store the shortest distance for each vertex from the source vertex. Initially distance of source will be 0 and Distance of other vertices will be INFINITY.

Initialize The Distance Array

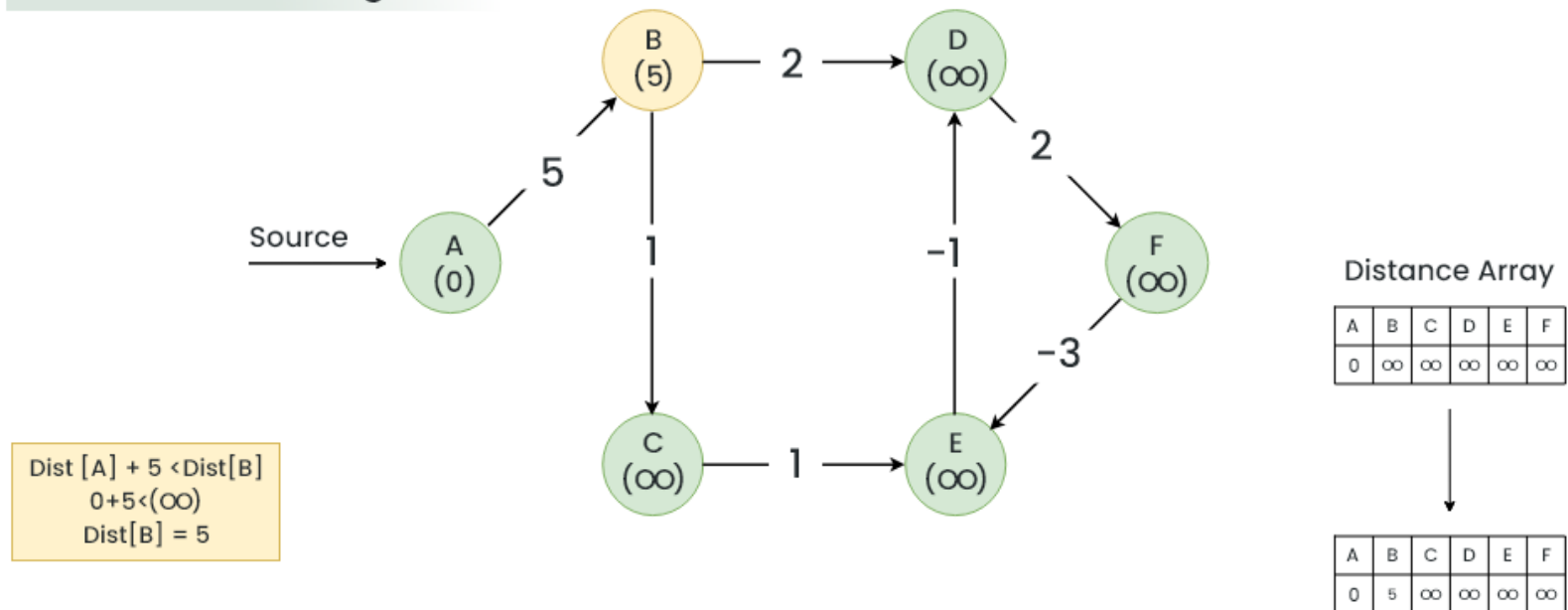


Distance Array  
 $\text{Dist}[]$

A	B	C	D	E	F
0	$\infty$	$\infty$	$\infty$	$\infty$	$\infty$

- Step 2: Start relaxing the edges, during 1st Relaxation:
- Current Distance of B  $>$  (Distance of A) + (Weight of A to B)  
i.e. Infinity  $>$   $0 + 5$
- Therefore,  $\text{Dist}[B] = 5$

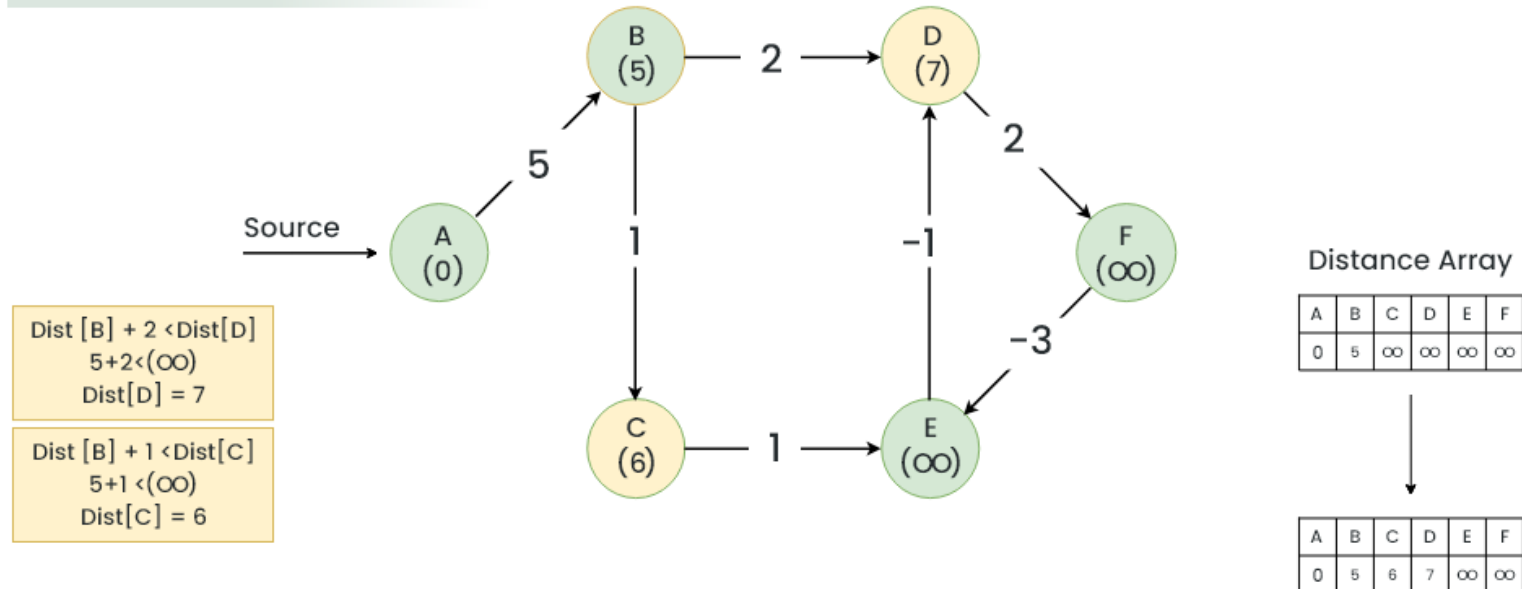
### 1st Relaxation Of Edges





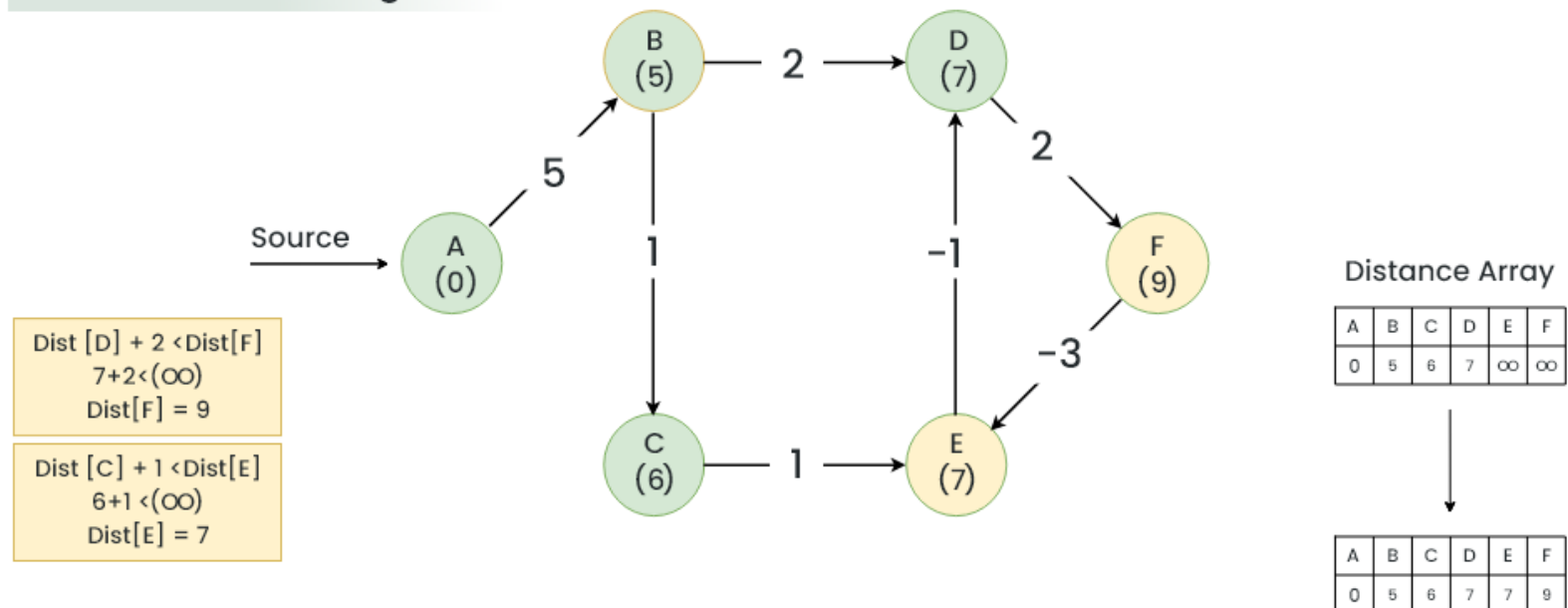
- Step 3: During 2nd Relaxation:
- Current Distance of D  $>$  (Distance of B) + (Weight of B to D)  
i.e. Infinity  $>$  5 + 2
- $\text{Dist}[D] = 7$
- Current Distance of C  $>$  (Distance of B) + (Weight of B to C)  
i.e. Infinity  $>$  5 + 1
- $\text{Dist}[C] = 6$

### 2nd Relaxation Of Edges



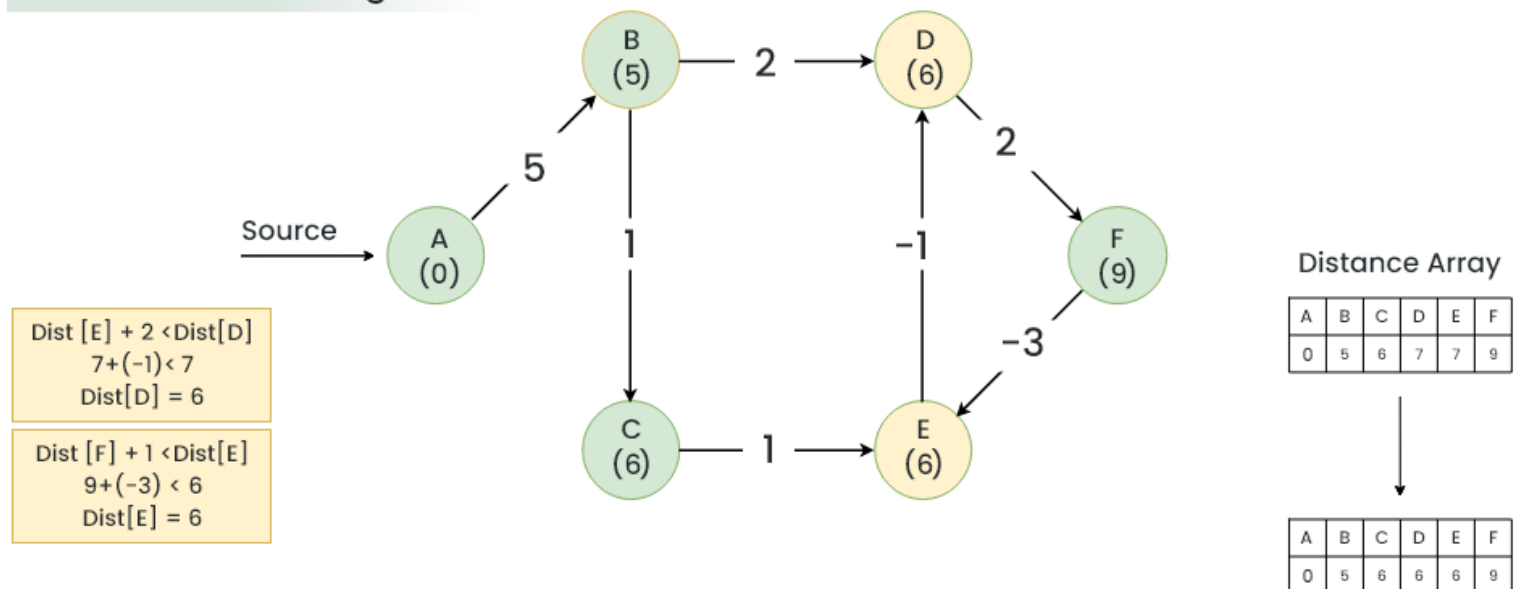
- Step 4: During 3rd Relaxation:
- Current Distance of F > (Distance of D ) + (Weight of D to F) i.e.  $\text{Infinity} > 7 + 2$
- $\text{Dist}[F] = 9$
- Current Distance of E > (Distance of C ) + (Weight of C to E) i.e.  $\text{Infinity} > 6 + 1$
- $\text{Dist}[E] = 7$

### 3rd Relaxation Of Edges



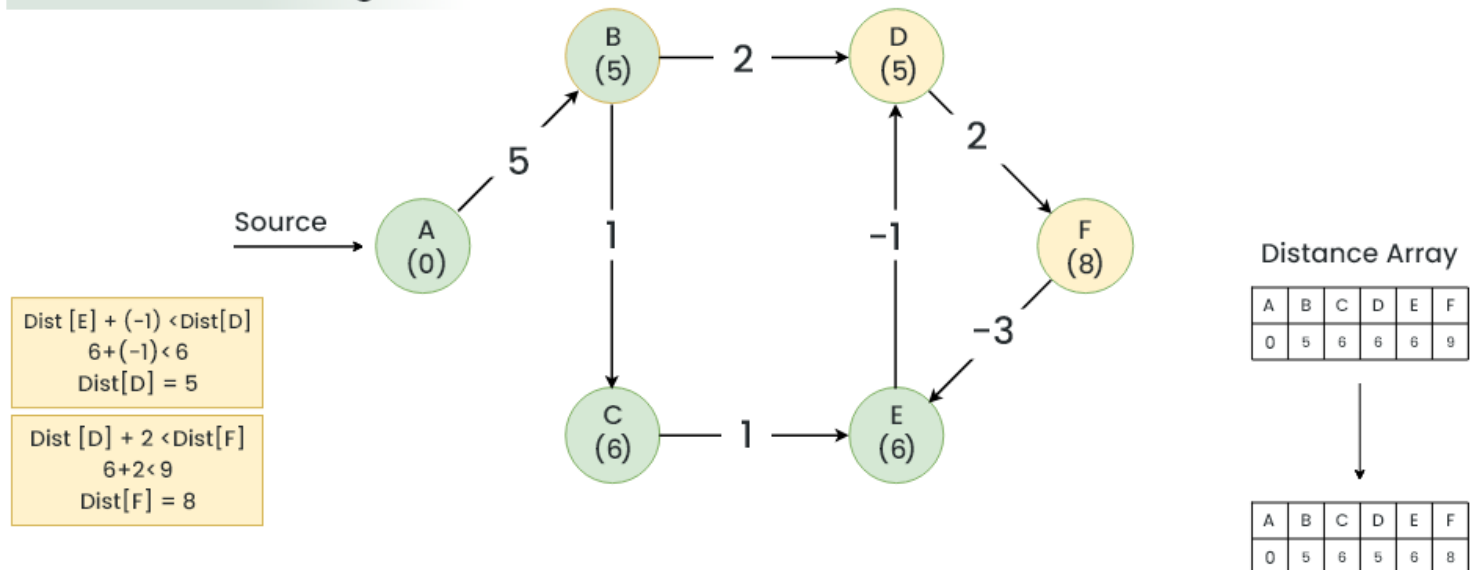
- Step 5: During 4th Relaxation:
- Current Distance of D > (Distance of E) + (Weight of E to D)  
i.e.  $7 > 7 + (-1)$
- $\text{Dist}[D] = 6$
- Current Distance of E > (Distance of F) + (Weight of F to E)  
i.e.  $7 > 9 + (-3)$
- $\text{Dist}[E] = 6$

#### 4th Relaxation Of Edges



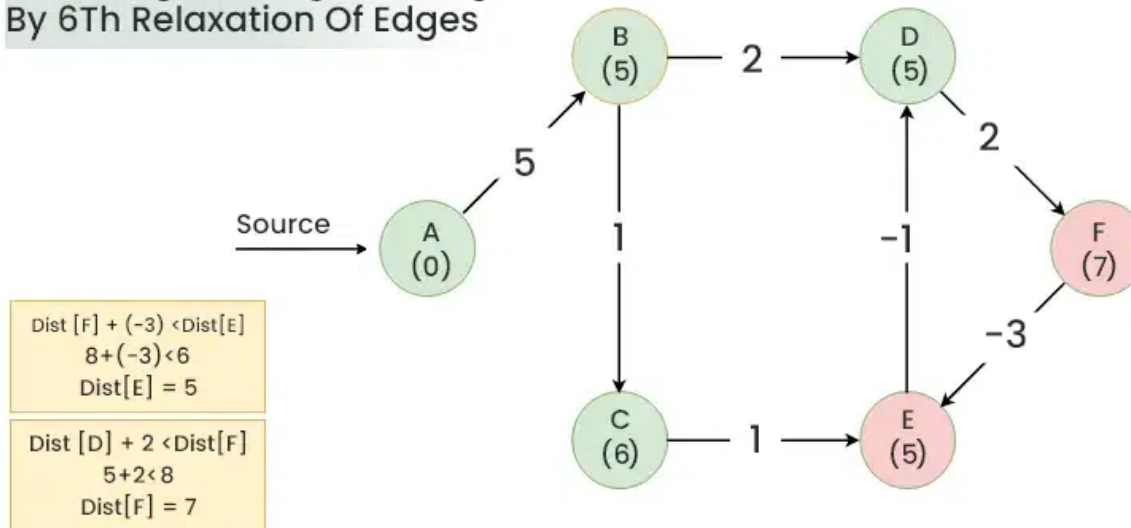
- Step 6: During 5th Relaxation:
- Current Distance of F > (Distance of D) + (Weight of D to F) i.e.  $9 > 6 + 2$
- $\text{Dist}[F] = 8$
- Current Distance of D > (Distance of E) + (Weight of E to D) i.e.  $6 > 6 + (-1)$
- $\text{Dist}[D] = 5$
- Since the graph has 6 vertices, So during the 5th relaxation the shortest distance for all the vertices should have been calculated.

#### 5th Relaxation Of Edges



- Step 7: Now the final relaxation i.e. the 6th relaxation should indicate the presence of negative cycle if there is any changes in the distance array of 5th relaxation.
- During the 6th relaxation, following changes can be seen:
- Current Distance of E > (Distance of F) + (Weight of F to E) i.e.  $6 > 8 + (-3)$
- Dist[E]=5
- Current Distance of F > (Distance of D) + (Weight of D to F) i.e.  $8 > 5 + 2$
- Dist[F]=7
- Since, we observe changes in the Distance array Hence, we can conclude the presence of a negative cycle in the graph (D->F->E).

#### Detecting The Negative Edge By 6Th Relaxation Of Edges



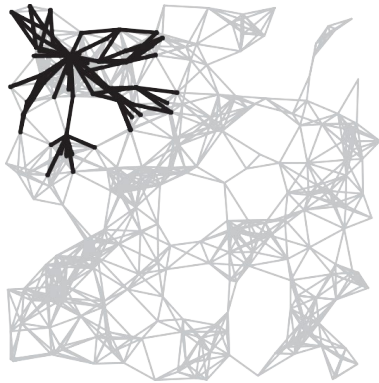
Distance Array

A	B	C	D	E	F
0	5	6	5	6	8

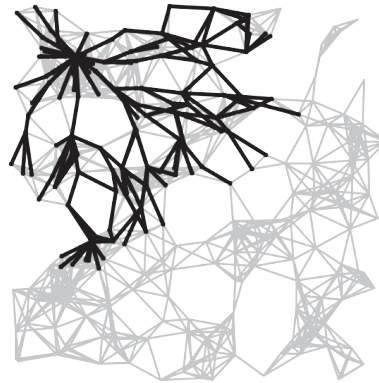
A	B	C	D	E	F
0	5	6	5	4	7

# Bellman-Ford Algorithm Visualization

passes 4



7



10



13



SPT



# Quiz

- Given a directed graph where weight of every edge is same, we can most efficiently find shortest path from a given source to destination using?
- A. Breadth First Traversal
- B. Dijkstra's Shortest Path Algorithm
- C. Neither Breadth First Traversal nor Dijkstra's algorithm can be used
- D. Depth First Search
- ANS: A
- With BFS, we first find explore vertices at one edge distance, then all vertices at 2 edge distance, and so on.

# Quiz

- Which of the following statement(s) is / are correct regarding Bellman-Ford shortest path algorithm?
- P: Always finds a negative weighted cycle, if one exist s.
- Q: Finds whether any negative weighted cycle is reachable from the source.
- ANS: Q
- Bellman-Ford shortest path algorithm is a single source shortest path algorithm. So it can only find cycles which are reachable from a given source, not any negative weight cycle. Consider a disconnected where a negative weight cycle is not reachable from the source at all. If there is a negative weight cycle, then it will appear in shortest path as the negative weight cycle will always form a shorter path when iterated through the cycle again and again.



# Quiz

- Let  $G = (V, E)$  be an undirected graph with a subgraph  $G_1 = (V_1, E_1)$ . Weights are assigned to edges of  $G$  as follows :
  - A single-source shortest path algorithm is executed on the weighted graph  $(V, E, w)$  with an arbitrary vertex  $v_1$  as the source. Which of the following can always be inferred from the path costs computed?
    - A. The number of edges in the shortest paths from  $v_1$  to all vertices of  $G$
    - B.  $G_1$  is connected or not
    - C.  $V_1$  forms a clique in  $G$
    - D.  $G_1$  is a tree
  - ANS: B
  - When shortest path from  $v_1$  (one of the vertices in  $V_1$ ) is computed.  $G_1$  is connected if the distance from  $v_1$  to any other vertex in  $V_1$  is greater than 0, otherwise  $G_1$  is disconnected.

# Quiz

- Let  $G = (V, E)$  be a simple undirected graph, and  $s$  be a particular vertex in it called the source. For  $x \in V$ , let  $d(x)$  denote the shortest distance in  $G$  from  $s$  to  $x$ . A breadth first search (BFS) is performed starting at  $s$ . Let  $T$  be the resultant BFS tree. If  $(u, v)$  is an edge of  $G$  that is not in  $T$ , then which one of the following CANNOT be the value of  $d(u) - d(v)$ ?
- A. -1 B. 0 C. 1 D. 2
- ANS: D
- Note that the given graph is undirected, so an edge  $(u, v)$  also means  $(v, u)$  is also an edge. Since a shorter path can always be obtained by using edge  $(u, v)$  or  $(v, u)$ , the difference between  $d(u)$  and  $d(v)$  can not be more than 1.

# Dijkstra's Algorithm vs. Bellman-Ford Algorithm

- Dijkstra's Algorithm:
  - Uses a priority queue to select the next vertex to process.
  - Greedily selects the vertex with the smallest tentative distance to source node.
  - Works only on graphs with non-negative edge weights.
  - Time complexity of  $O(V^2)$  for a dense graph and  $O(E \log V)$  for a sparse graph.
- Bellman-Ford Algorithm:
  - Iteratively relaxes all edges  $V-1$  times, where  $V$  is the number of vertices.
  - Does not use a priority queue.
  - Can handle graphs with negative edge weights, and can detect negative cycles.
  - Time complexity of  $O(VE)$ , where  $V$  is the number of vertices and  $E$  is the number of edges in the graph.
- Dijkstra's algorithm is faster and more efficient for graphs with non-negative weights, the Bellman-Ford algorithm is more versatile as it can handle negative weights and detect negative cycles, albeit at the cost of lower efficiency.

# Time complexity of Bellman-Ford

- What is the time complexity of Bellman-Ford single-source shortest path algorithm on a complete graph of  $n$  vertices?
- ANS:  $O(V^3)$
- Time complexity of Bellman-Ford algorithm is  $O(VE)$  where  $V$  is number of vertices and  $E$  is number edges . If the graph is complete, the value of  $E$  becomes  $O(V^2)$ . So overall time complexity becomes  $O(V^3)$

# Single Source Shortest-paths Implementation: Cost Summary

algorithm	restriction	typical case	worst case	extra space
<b>topological sort</b>	no directed cycles	$E + V$	$E + V$	$V$
<b>Dijkstra (binary heap)</b>	no negative weights	$E \log V$	$E \log V$	$V$
<b>Bellman-Ford</b>	no negative cycles	$E V$	$E V$	$V$
<b>Bellman-Ford (queue-based) (omitted)</b>		$E + V$	$E V$	$V$

- **Remark 1.** Directed cycles make the problem harder.
- **Remark 2.** Negative weights make the problem harder.
- **Remark 3.** Negative cycles makes the problem intractable.

# Quiz

- Which of the following algorithm can be used to efficiently calculate single source shortest paths in a Directed Acyclic Graph?
  - Dijkstra
  - Bellman-Ford
  - Topological Sort
  - Strongly Connected Component
- ANS: Topological Sort
- Using Topological Sort, we can find single source shortest paths in  $O(V+E)$  time which is the most efficient algorithm

# Quiz

- Given a graph where all edges have positive weights, the shortest paths produced by Dijkstra and Bellman Ford algorithm may be different but path weight would always be same.
- ANS: True
- Dijkstra and Bellman-Ford both work fine for a graph with all positive weights, but they are different algorithms and may pick different edges for shortest paths.

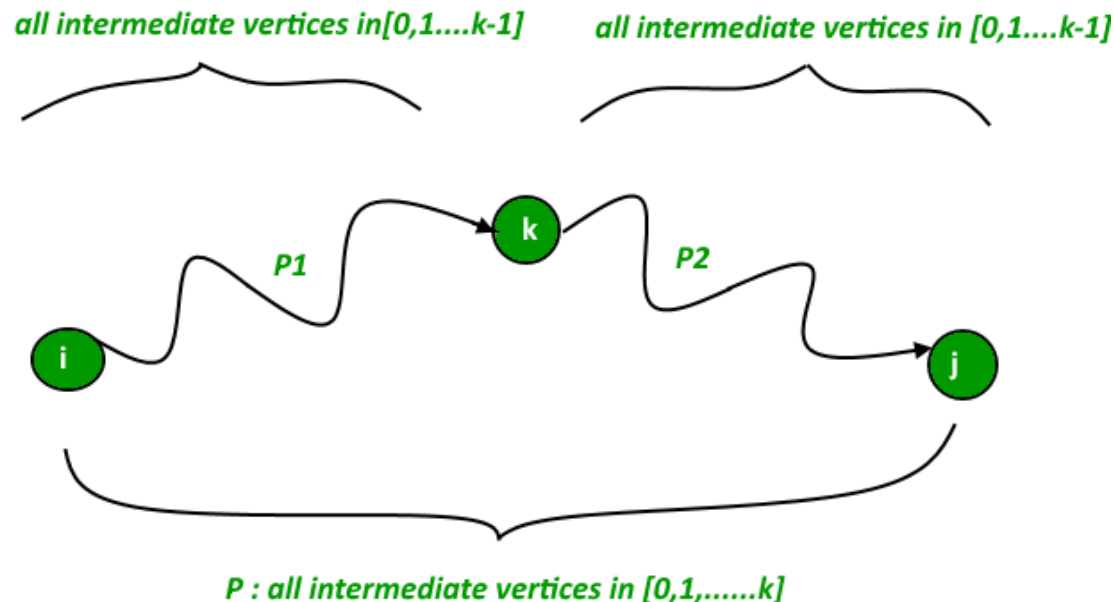
# Floyd Warshall Algorithm

- The Floyd Warshall Algorithm is an all pair shortest path algorithm unlike Dijkstra and Bellman Ford which are single source shortest path algorithms.
- It works for both the directed and undirected weighted graphs. But, it does not work for the graphs with negative cycles
- It follows Dynamic Programming approach to check every possible path going via every possible node in order to calculate shortest distance between every pair of nodes.
- *For  $k = 0$  to  $n - 1$   
    For  $i = 0$  to  $n - 1$   
        For  $j = 0$  to  $n - 1$   
             $Distance[i, j] = \min(Distance[i, j], Distance[i, k] + Distance[k, j])$*
- *where  $i$  = source Node,  $j$  = Destination Node,  $k$  = Intermediate Node*
- Time Complexity:  $O(V^3)$ , where  $V$  is the number of vertices in the graph and we run three nested loops each of size  $V$

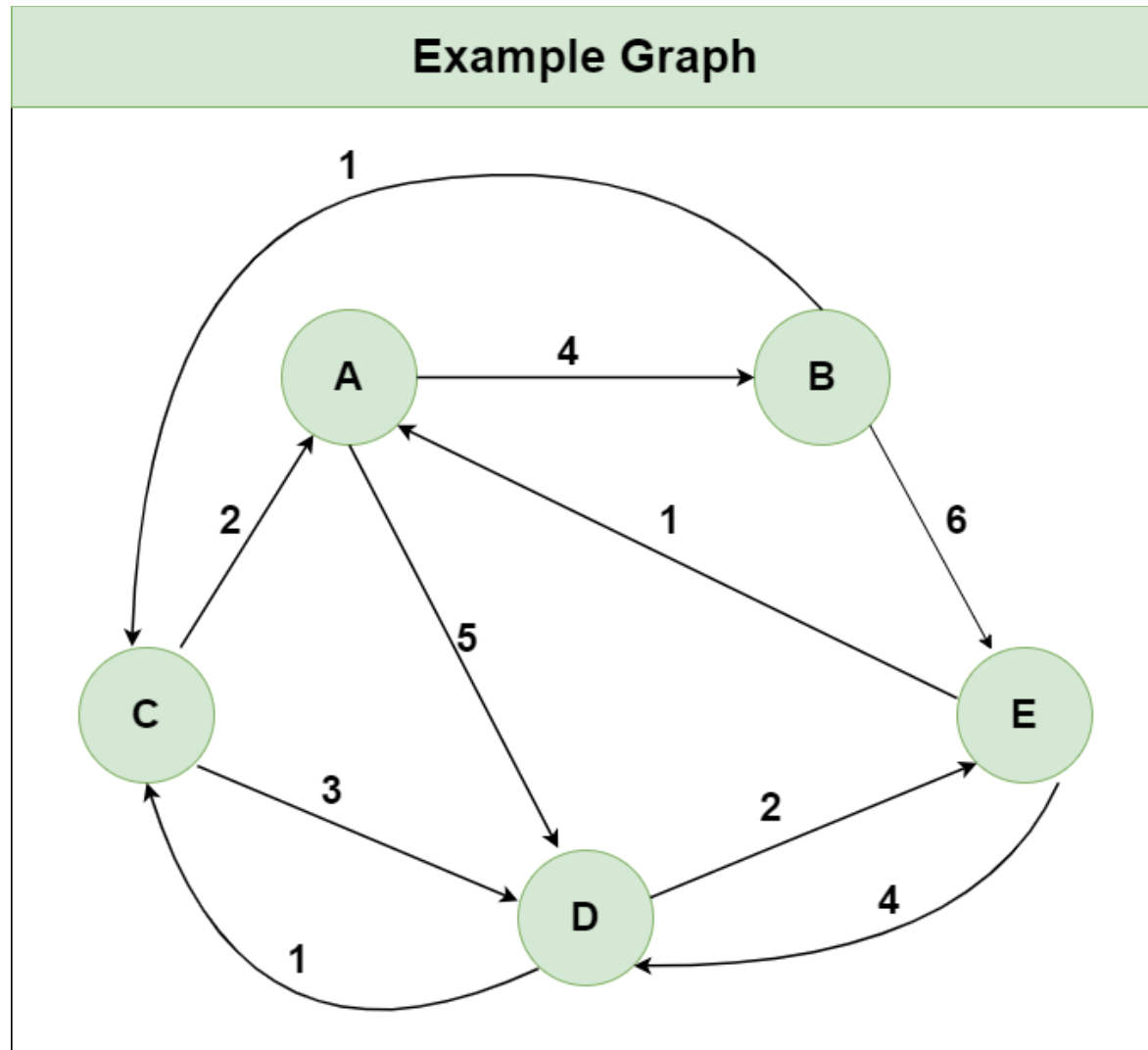


# Floyd Warshall Algorithm is Dynamic Programming

- Floyd Warshall Algorithm is a Dynamic Programming based algorithm. It finds all pairs shortest paths using following recursive nature of problem. For every pair  $(i, j)$  of source and destination vertices respectively, there are two possible cases. 1)  $k$  is not an intermediate vertex in shortest path from  $i$  to  $j$ . We keep the value of  $\text{dist}[i][j]$  as it is. 2)  $k$  is an intermediate vertex in shortest path from  $i$  to  $j$ . We update the value of  $\text{dist}[i][j]$  as  $\text{dist}[i][k] + \text{dist}[k][j]$ . The following figure is taken from the Cormen book. It shows the above optimal substructure property in the all-pairs shortest path problem.
- Since there are overlapping subproblems in recursion, it uses dynamic programming



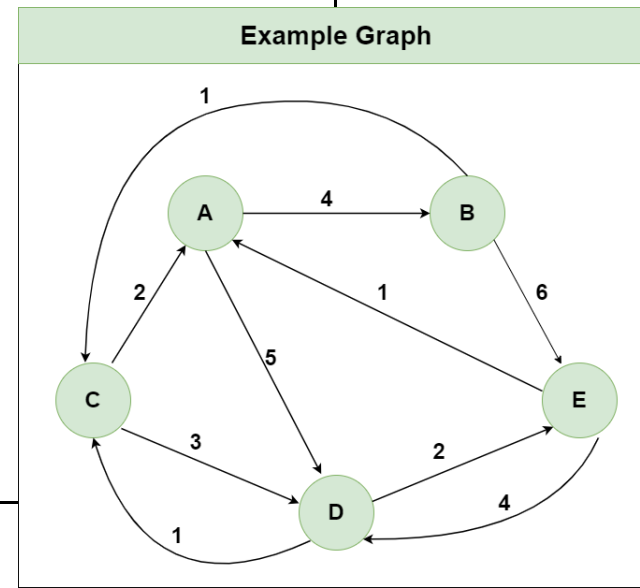
# Floyd Warshall Algorithm Example



- **Step 1:** Initialize the  $Distance[][]$  matrix using the input graph such that  $Distance[i][j]$  = weight of edge from  $i$  to  $j$ , also  $Distance[i][j] = \text{Infinity}$  if there is no edge from  $i$  to  $j$ .

### Step1: Initializing $Distance[ ][ ]$ using the Input Graph

	A	B	C	D	E
A	0	4	$\infty$	5	$\infty$
B	$\infty$	0	1	$\infty$	6
C	2	$\infty$	0	3	$\infty$
D	$\infty$	$\infty$	1	0	2
E	1	$\infty$	$\infty$	4	0



- **Step 2:** Treat node **A** as an intermediate node and calculate the  $Distance[i][j]$  for every  $\{i,j\}$  node pair using the formula:
- $= Distance[i][j] = \text{minimum} (Distance[i][j], Distance[i][A] + Distance[A][j])$

### Step 2: Using Node A as the Intermediate node

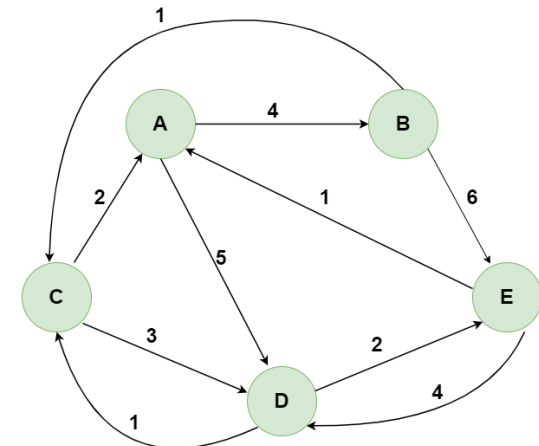
$$Distance[i][j] = \min (Distance[i][j], Distance[i][A] + Distance[A][j])$$

	A	B	C	D	E
A	0	4	$\infty$	5	$\infty$
B	$\infty$	?	?	?	?
C	2	?	?	?	?
D	$\infty$	?	?	?	?
E	1	?	?	?	?



	A	B	C	D	E
A	0	4	$\infty$	5	$\infty$
B	$\infty$	0	1	$\infty$	6
C	2	6	0	3	12
D	$\infty$	$\infty$	1	0	2
E	1	5	$\infty$	4	0

Example Graph



- Step 3:** Treat node **B** as an intermediate node and calculate the  $Distance[i][j]$  for every  $\{i,j\}$  node pair using the formula:  
 $Distance[i][j] = \text{minimum} (Distance[i][j], Distance[i][B] + Distance[B][j])$

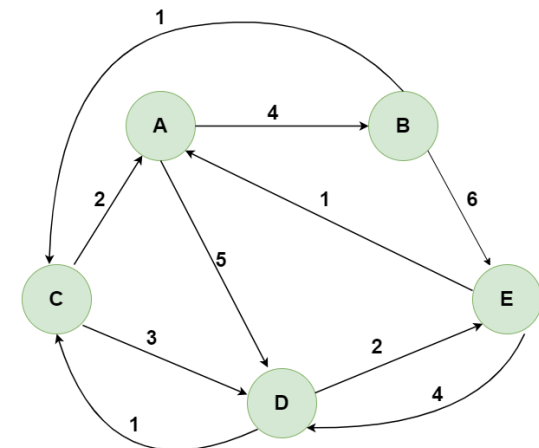
### Step 3: Using Node B as the Intermediate node

$$Distance[i][j] = \min (Distance[i][j], Distance[i][B] + Distance[B][j])$$

	A	B	C	D	E
A	?	4	?	?	?
B	$\infty$	0	1	$\infty$	6
C	?	6	?	?	?
D	?	$\infty$	?	?	?
E	?	5	?	?	?

	A	B	C	D	E
A	0	4	5	5	10
B	$\infty$	0	1	$\infty$	6
C	2	6	0	3	12
D	$\infty$	$\infty$	1	0	2
E	1	5	6	4	0

Example Graph



- **Step 4:** Treat node **C** as an intermediate node and calculate the  $Distance[i][j]$  for every  $\{i,j\}$  node pair using the formula:  
 $= Distance[i][j] = \text{minimum} (Distance[i][j], Distance[i][C] + Distance[C][j])$

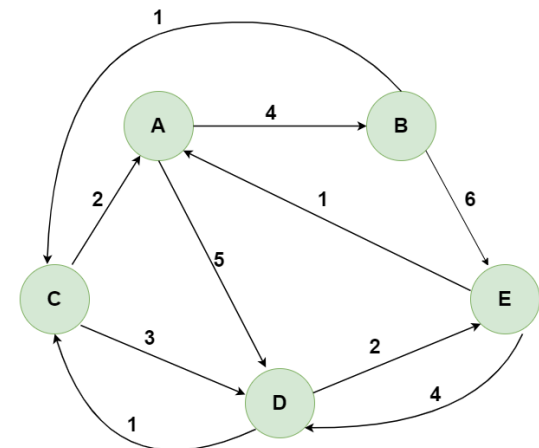
#### Step 4: Using Node C as the Intermediate node

$$Distance[i][j] = \min (Distance[i][j], Distance[i][C] + Distance[C][j])$$

	A	B	C	D	E
A	?	?	5	?	?
B	?	?	1	?	?
C	2	6	0	3	12
D	?	?	1	?	?
E	?	?	6	?	?

	A	B	C	D	E
A	0	4	5	5	10
B	3	0	1	4	6
C	2	6	0	3	12
D	3	7	1	0	2
E	1	5	6	4	0

Example Graph



- **Step 5:** Treat node **D** as an intermediate node and calculate the  $Distance[i][j]$  for every  $\{i,j\}$  node pair using the formula:  
 $= Distance[i][j] = \text{minimum} (Distance[i][j], Distance[i][D] + Distance[D][j])$

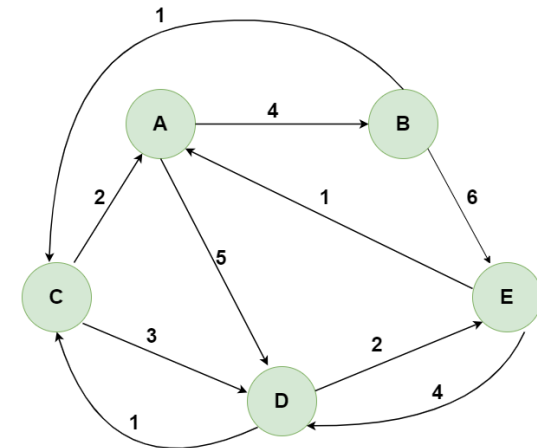
### Step 5: Using Node D as the Intermediate node

$$Distance[i][j] = \min (Distance[i][j], Distance[i][D] + Distance[D][j])$$

	A	B	C	D	E
A	?	?	?	5	?
B	?	?	?	4	?
C	?	?	?	3	?
D	3	7	1	0	2
E	?	?	?	4	?

	A	B	C	D	E
A	0	4	5	5	7
B	3	0	1	4	6
C	2	6	0	3	5
D	3	7	1	0	2
E	1	5	5	4	0

Example Graph



- Step 6:** Treat node **E** as an intermediate node and calculate the  $Distance[i][j]$  for every  $\{i,j\}$  node pair using the formula:  
 $= Distance[i][j] = \text{minimum} (Distance[i][j], Distance[i][E] + Distance[E][j])$

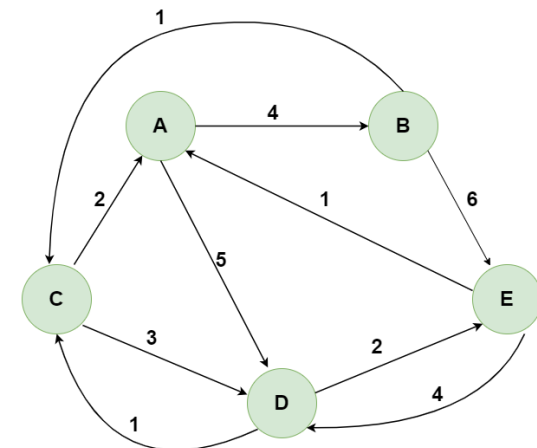
### Step 6: Using Node E as the Intermediate node

$$Distance[i][j] = \min (Distance[i][j], Distance[i][E] + Distance[E][j])$$

	A	B	C	D	E
A	?	?	?	?	7
B	?	?	?	?	6
C	?	?	?	?	5
D	?	?	?	?	2
E	1	5	5	4	0

	A	B	C	D	E
A	0	4	5	5	7
B	3	0	1	4	6
C	2	6	0	3	5
D	3	7	1	0	2
E	1	5	5	4	0

Example Graph



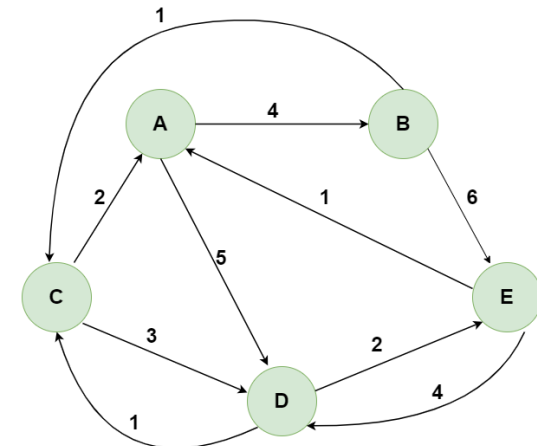


- **Step 7:** Since all the nodes have been treated as an intermediate node, we can now return the updated *Distance[][]* matrix as our answer matrix.

Step 7: Return Distance[ ][ ] matrix as the result

	A	B	C	D	E
A	0	4	5	5	7
B	3	0	1	4	6
C	2	6	0	3	5
D	3	7	1	0	2
E	1	5	5	4	0

Example Graph

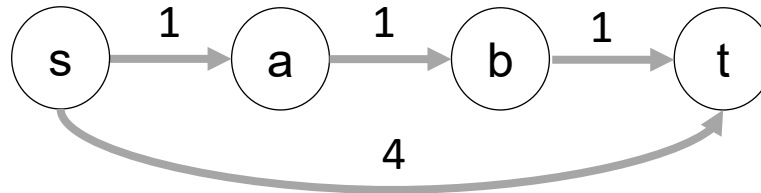


# Johnson's algorithm for All-pairs shortest paths

- Johnson's algorithm uses both Dijkstra and Bellman-Ford as subroutines. If we apply Dijkstra's Single Source shortest path algorithm for every vertex, considering every vertex as the source, we can find all pair shortest paths in  $O(V*V\text{Log}V)$  time.
- Dijkstra's algorithm doesn't work for negative weight edge. The idea of Johnson's algorithm is to re-weight all edges and make them all positive, then apply Dijkstra's algorithm for every vertex.
- How to transform a given graph into a graph with all non-negative weight edges?

# Quiz: increase weight of every edge by 1

- In a weighted graph, assume that the shortest path from a source 's' to a destination 't' is correctly calculated using a shortest path algorithm. Is the following statement true? If we increase weight of every edge by 1, the shortest path always remains same.
- See the following counterexample. There are 4 edges s-a, a-b, b-t and s-t of weights 1, 1, 1 and 4 respectively. The shortest path from s to t is s-a, a-b, b-t=3. If we increase weight of every edge by 1, the shortest path changes to s-t = 5.



# Quiz: Doubling of the original weight

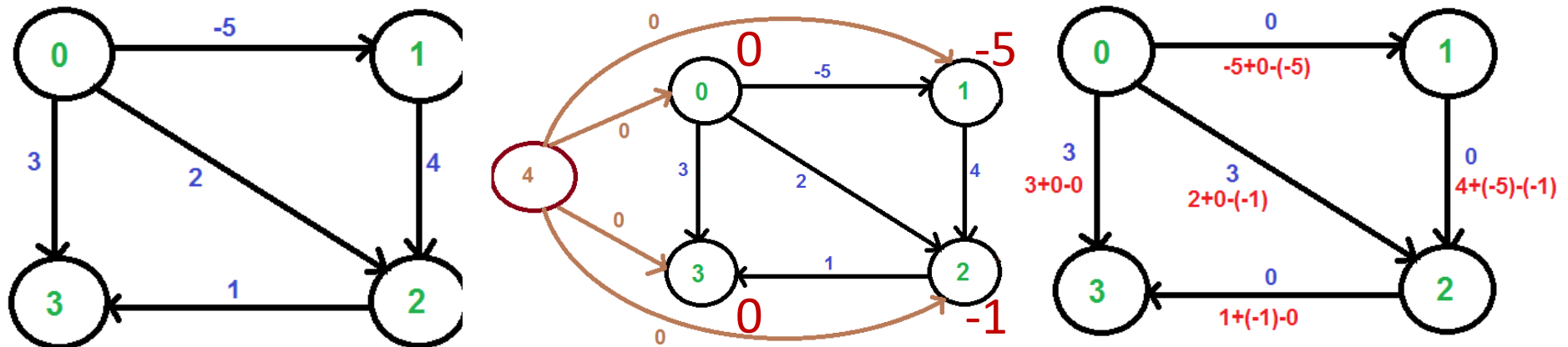
- Is the following statement valid about shortest paths? Given a graph, suppose we have calculated shortest path from a source to all other vertices. If we modify the graph such that weights of all edges becomes double of the original weight, then the shortest path remains same only the total weight of path changes.
- True.
- The shortest path remains same. It is like if we change unit of distance from meter to kilo meter, the shortest paths don't change. But this does not make weights positive.

# Johnson's algorithm for All-pairs shortest paths

1. Let the given graph be  $G$ . Add a new vertex  $s$  to the graph, add edges from the new vertex to all vertices of  $G$ . Let the modified graph be  $G'$ .
2. Run the Bellman-Ford algorithm on  $G'$  with  $s$  as the source. Let the distances calculated by Bellman-Ford be  $h[0]$ ,  $h[1]$ , ..  $h[V-1]$ . If we find a negative weight cycle, then return.
3. Reweight the edges of the original graph. For each edge  $(u, v)$ , assign the new weight as “original weight +  $h[u] - h[v]$ ”.
4. Remove the added vertex  $s$  and run Dijkstra's algorithm for every vertex.

# Johnson's Algorithm Example

- We add a source  $s$  and add edges from  $s$  to all vertices of the original graph. In the following diagram  $s$  is 4.
- We calculate the shortest distances from 4 to all other vertices using Bellman-Ford algorithm. The shortest distances from 4 to 0, 1, 2 and 3 are 0, -5, -1 and 0 respectively, i.e.,  $h[] = \{0, -5, -1, 0\}$ . Once we get these distances, we remove the source vertex 4 and reweight the edges using following formula.  $w(u, v) = w(u, v) + h[u] - h[v]$ .
- Since all weights are positive now, we can run Dijkstra's shortest path algorithm for every vertex as the source.



Distances from 4 to 0, 1, 2 and 3 are 0, -5, -1 and 0 respectively.

# Johnson's Algorithm: Proof

- The following property is always true about  $h[]$  values as they are the shortest distances.
- $h[v] \leq h[u] + w(u, v)$
- The property simply means that the shortest distance from  $s$  to  $v$  must be smaller than or equal to the shortest distance from  $s$  to  $u$  plus the weight of the edge  $(u, v)$ . The new weights are  $w(u, v) + h[u] - h[v]$ . The value of the new weights must be greater than or equal to zero because of the inequality " $h[v] \leq h[u] + w(u, v)$ ".
- After reweighting, all set of paths between any two vertices is increased by the same amount and all negative weights become non-negative. Consider any path between two vertices  $s$  and  $t$ , the weight of every path is increased by  $h[s] - h[t]$ , and all  $h[]$  values of vertices on the path from  $s$  to  $t$  cancel each other.
- Time complexity: The main steps in the algorithm are Bellman-Ford Algorithm called once and Dijkstra called  $V$  times. Time complexity of Bellman Ford is  $O(VE)$  and time complexity of Dijkstra is  $O(V \log V)$ . So overall time complexity is  $O(V^2 \log V + VE)$ .

- We add a source  $s$  and add edges from  $s$  to all vertices of the original graph. In the following diagram  $s$  is 4.

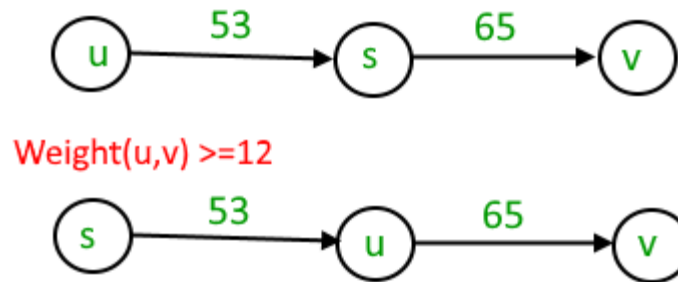


# Quiz

- Match the following
  - Group A
    - a) Dijkstra's single shortest path algo
    - b) Bellman Ford's single shortest path algo
    - c) Floyd Warshall's all pair shortest path algo
  - Group B
    - p) Dynamic Programming
    - q) Backtracking
    - r) Greedy Algorithm
- Dijkstra is a greedy algorithm where we pick the minimum distant vertex from not yet finalized vertices. Bellman Ford and Floyd Warshall both are Dynamic Programming algorithms where we build the shortest paths in bottom up manner.

# Quiz

- Consider a weighted undirected graph with positive edge weights and let  $uv$  be an edge in the graph. It is known that the shortest path from the source vertex  $s$  to  $u$  has weight 53 and the shortest path from  $s$  to  $v$  has weight 65. Which one of the following statements is always true?
- A.  $\text{weight}(u, v) < 12$
- B.  $\text{weight}(u, v) \leq 12$
- C.  $\text{weight}(u, v) > 12$
- D.  $\text{weight}(u, v) \geq 12$
- ANS:
- The minimum weight happens when  $(S,U) + (U,V) = (S,V)$  Else  $(S,U) + (U,V) \geq (S,V)$  Given  $(S,U) = 53$ ,  $(S,V) = 65$ ,  $53 + (U,V) \geq 65$ ,  $(U,V) \geq 12$ .





# Quiz

- Let  $G$  be a directed graph whose vertex set is the set of numbers from 1 to 100. There is an edge from a vertex  $i$  to a vertex  $j$  if either  $j = i + 1$  or  $j = 3i$ . The minimum number of edges in a path in  $G$  from vertex 1 to vertex 100 is
- A. 4 B. 7 C. 23 D. 99
- ANS: 7
- The task is to find minimum number of edges in a path in  $G$  from vertex 1 to vertex 100 such that we can move to either  $i+1$  or  $3i$  from a vertex  $i$ .
- Since the task is to minimize number of edges,
- we would prefer to follow  $3*i$ .
- Let us follow multiple of 3.
- $1 \Rightarrow 3 \Rightarrow 9 \Rightarrow 27 \Rightarrow 81$ , now we can't follow multiple
- of 3. So we will have to follow  $i+1$ . This solution gives
- a long path.
- What if we begin from end, and we reduce by 1 if
- the value is not multiple of 3, else we divide by 3.
- $100 \Rightarrow 99 \Rightarrow 33 \Rightarrow 11 \Rightarrow 10 \Rightarrow 9 \Rightarrow 3 \Rightarrow 1$
- So we need total 7 edges.

# Backup Slides