Module 5
Graph Algorithms

Dr. Natarajan Meghanathan
Professor of Computer Science
Jackson State University
Jackson, MS 39217
E-mail: natarajan.meghanathan@jsums.edu

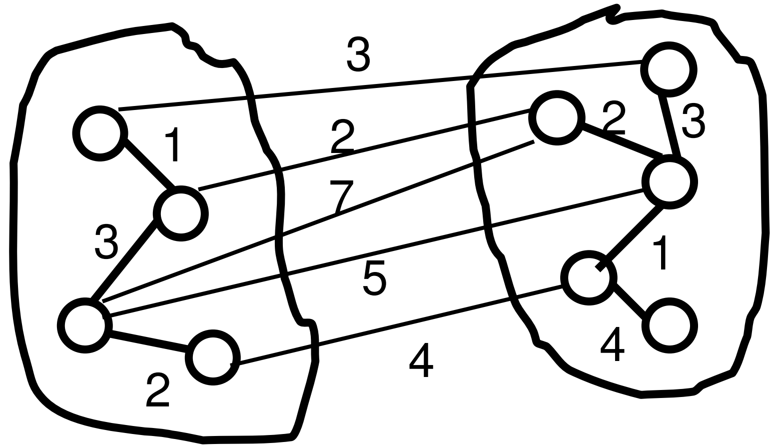
Minimum Spanning Trees

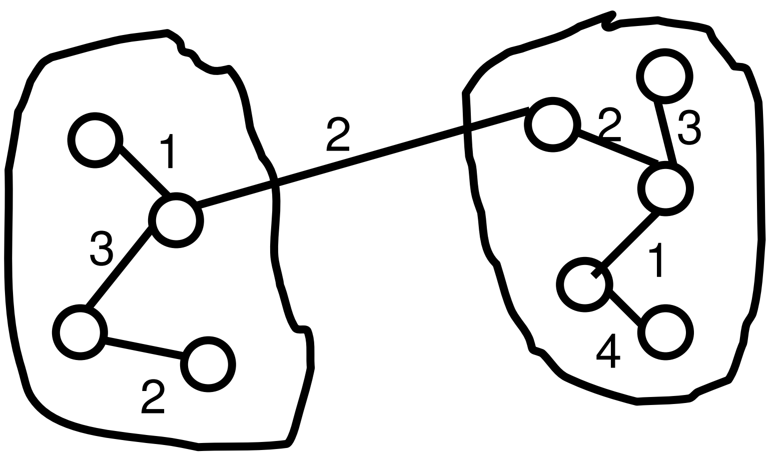
Minimum Spanning Tree Problem

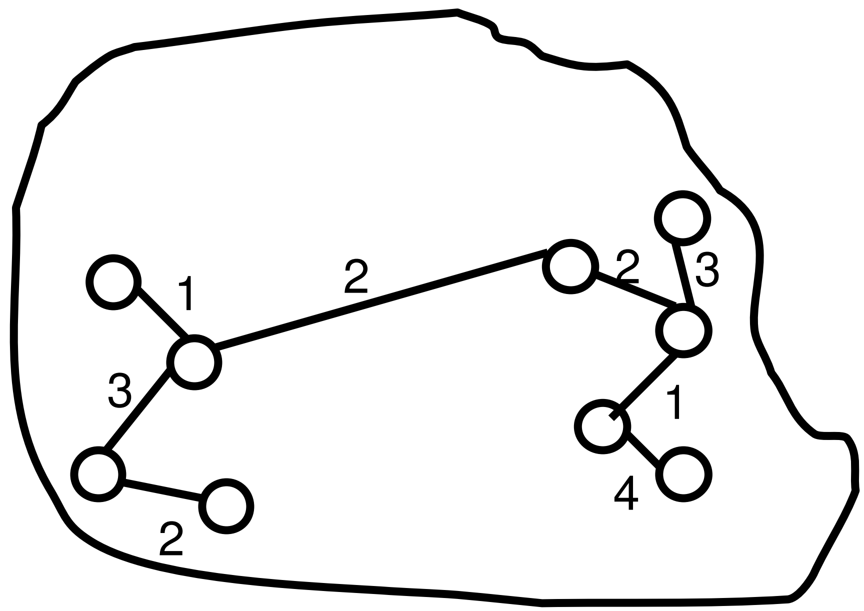
- Given a weighted graph, we want to determine a tree that spans all the vertices in the tree and the sum of the weights of all the edges in such a spanning tree should be minimum.
- <u>Kruskal algorithm:</u> Consider edges in the increasing order of their weights and include an edge in the tree, if and only if, by including the edge in the tree, we do not create a cycle!!
  - For a graph of E edges, we spend $\Theta(E*\log E)$ time to sort the edges and this is the most time consuming step of the algorithm.
- To start with, each vertex is in its own component.
- In each iteration, we merge two components using an edge of minimum weight connecting the vertices across the two components.
  - The merged component does not have a cycle and the sum of all the edge weights within a component is the minimum possible.
- To detect a cycle, the vertices within a component are identified by a component ID. If the edge considered for merging two components comprises of end vertices with the same component ID, then the edge is not considered for the merger.
  - An edge is considered for merging two components only if its end vertices are identified with different component IDs.

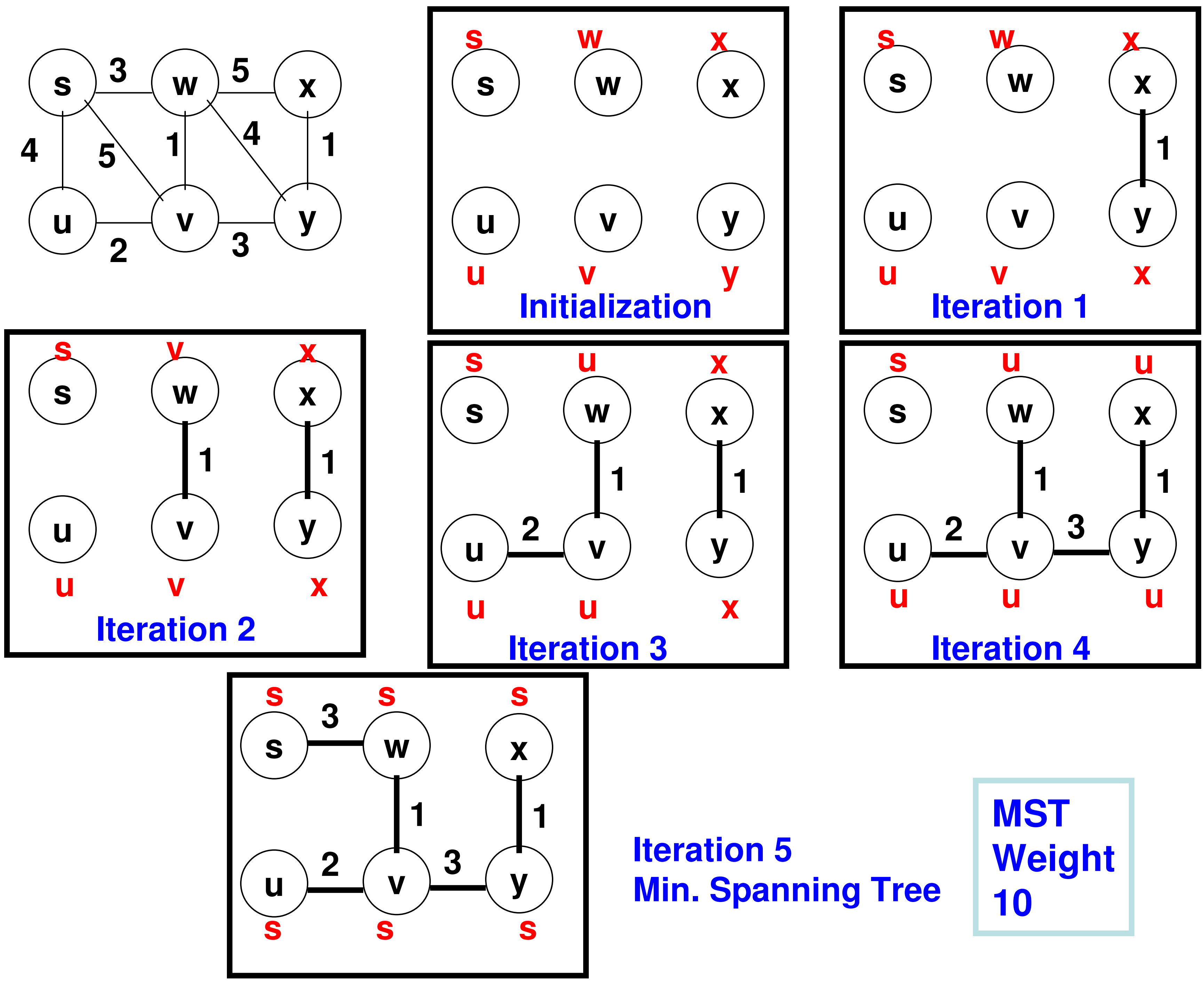
Property of any MST Algorithm

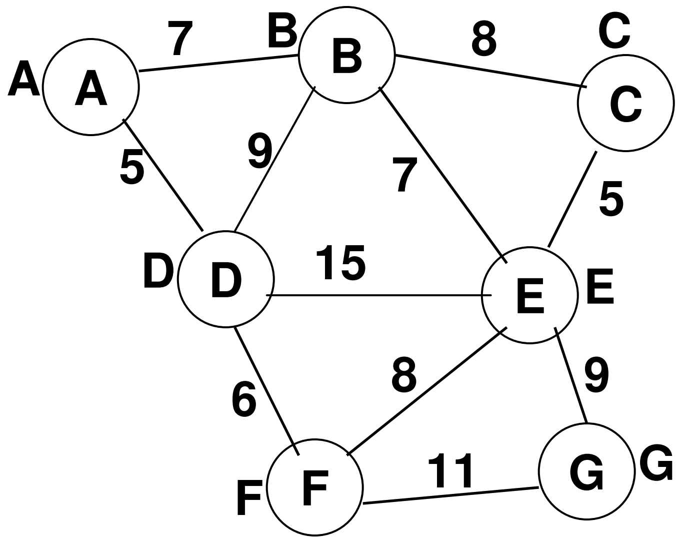
- Given two components of vertices (that are a tree by themselves of the smallest possible weights), any MST algorithm would choose an edge of the smallest weight that could connect the two components such that the merger of the two components is also a tree and is of the smallest possible weight.

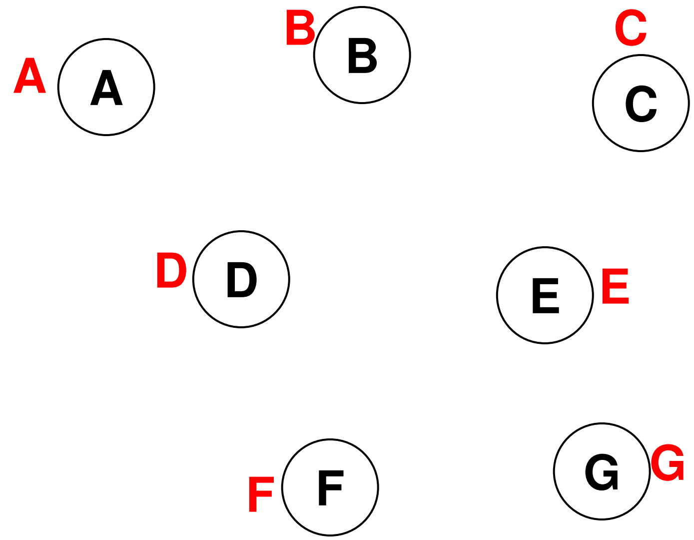




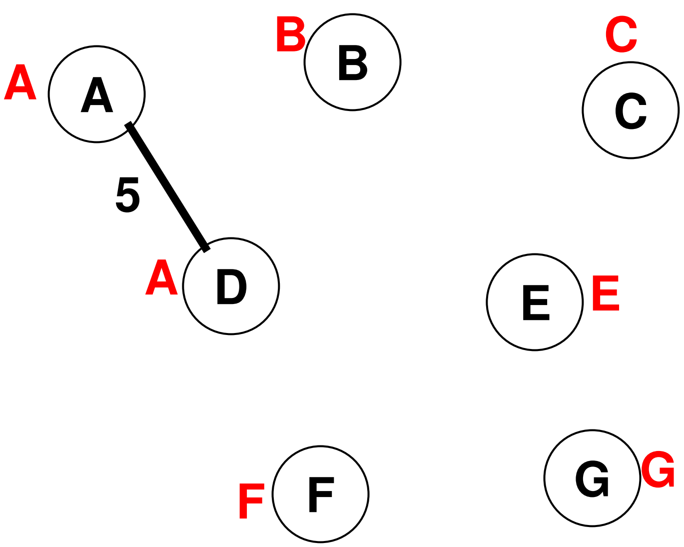



s
w
x
u
v
y
3
5
4
5
1
4
1
2
3
Initialization
Iteration 1
Iteration 2
Iteration 3
Iteration 4
Iteration 5
Min. Spanning Tree
MST
Weight
10

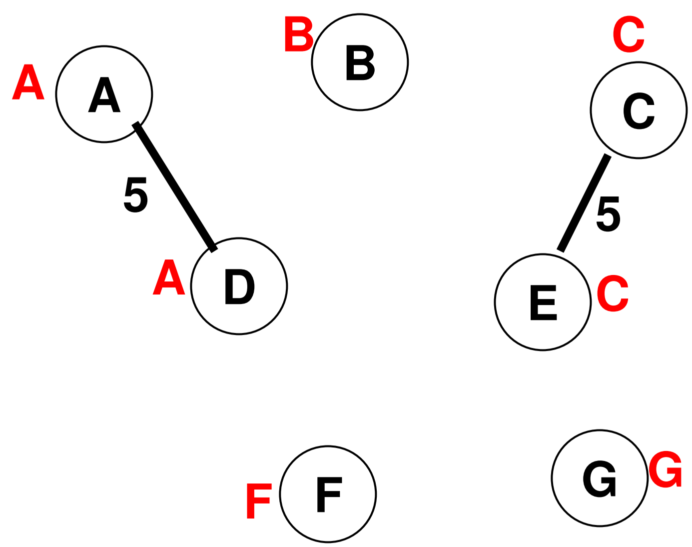




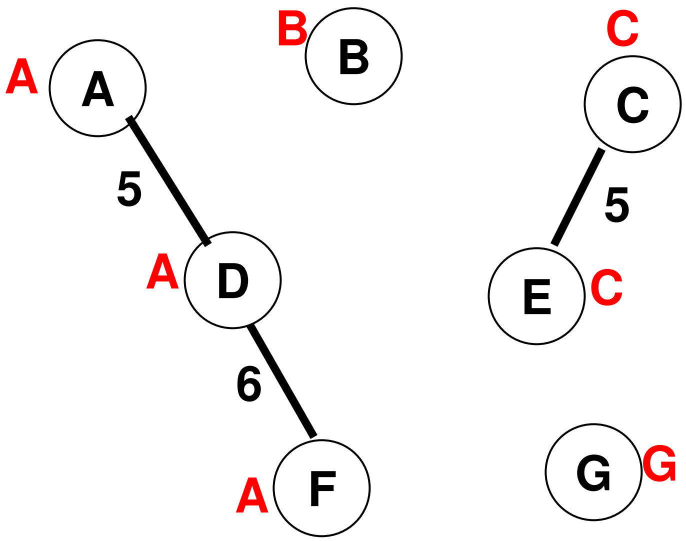
Initialization



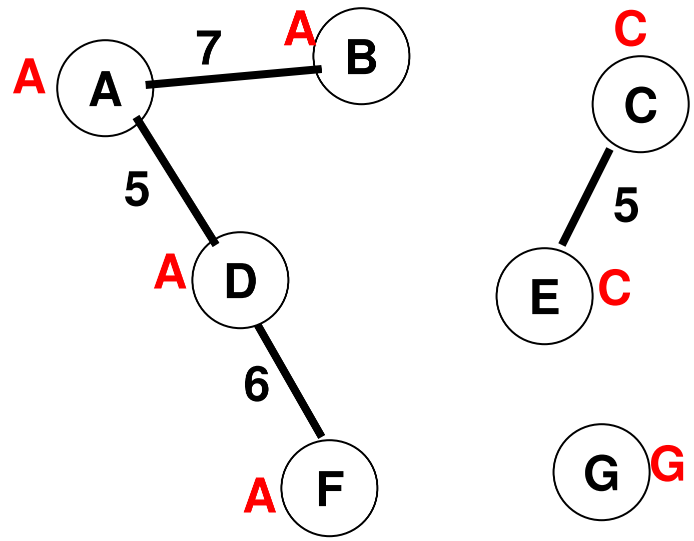
Iteration 1



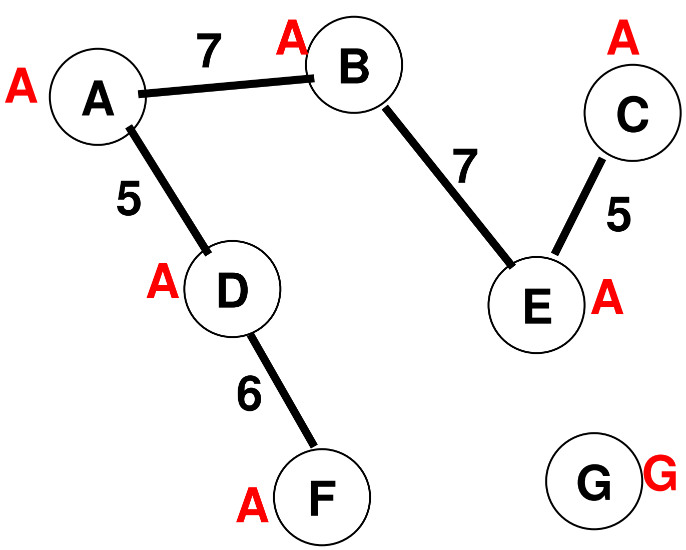
Iteration 2



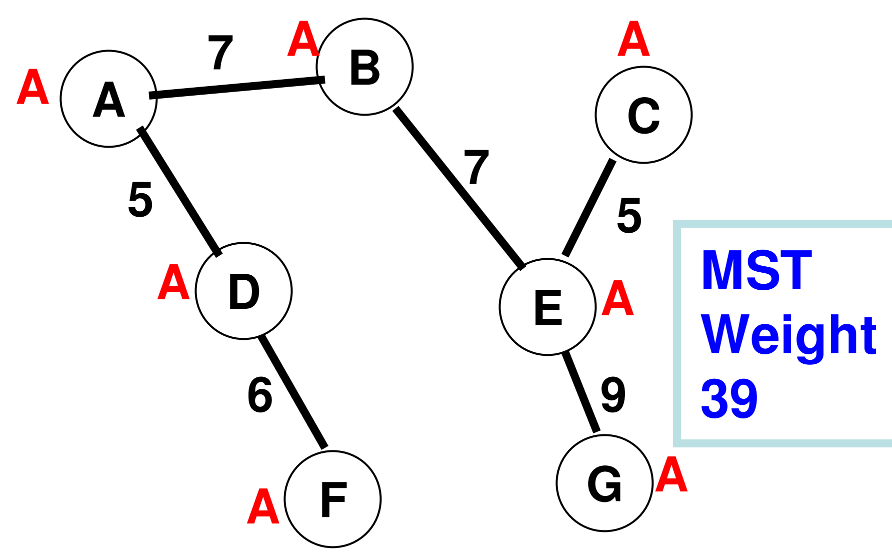
Iteration 3



Iteration 4



Iteration 5



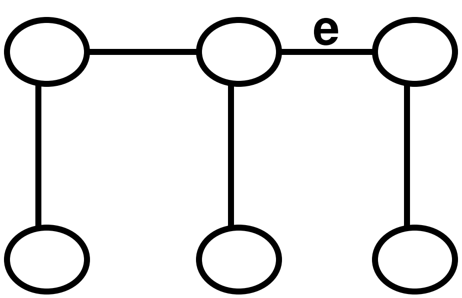
Iteration 6: Min. Sp Tree

Proof of Correctness: Kruskal's Algorithm

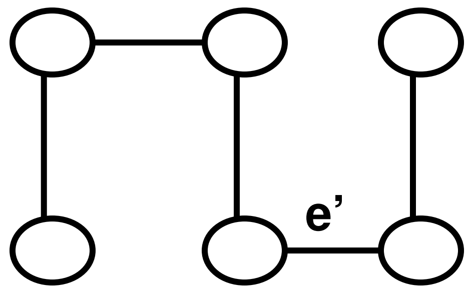
- Let *T* be the spanning tree generated by Kruskal's algorithm for a graph *G*. Let *T'* be a minimum spanning tree for *G*. We need to show that both *T* and *T'* have the same weight.
- Assume that wt(*T'*) < wt(*T*).
- Hence, there should be an edge *e* in *T* that is not in *T* ' and likewise there should be an edge e' in T' that is not in T. Because, if every edge of *T* is in *T'*, then *T* = *T'* and wt(*T*) = wt(*T'*).
- Remove the edge e' that is in T'. This would disconnect the T' to two components. The edge e that was in T and not in T' should be one of the edges (along with e') that cross the two split components of T'.
- Depending on how Kruskal's algorithm works, wt(e) ≤ wt(e'). Hence, the two components of T' could be merged using edge e (instead of e') and this would only lower the weight of T' from what it was before (and not increase it).
- That is, wt(modified T') = wt(T' – {e'} U {e}) ≤ wt(T').
- We could repeat the above procedure for all edges that are in T' and not in T, and eventually transform *T'* to *T*, without increasing the cost of the spanning tree.
- Hence, *T* is a minimum spanning tree.

Proof of Correctness

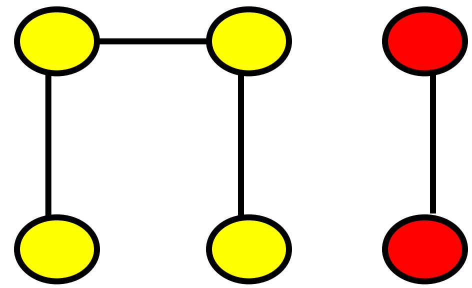
Let T be the spanning tree determined using Kruskal's
Let T' be a hypothetical spanning tree that is a MST such that W(T') < W(T)



T

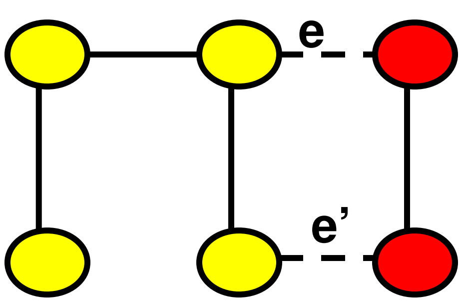


T'

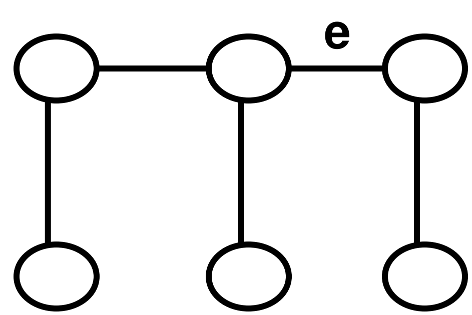


Wt(e) ≤ Wt(e')

Wt(T' – {e'} U {e}) ≤ Wt(T'). Hence, by reducing the edge difference and making T' approach T, we are able to only decrease the weight of T' further, if possible, making T' not a MST to start with, a contradiction.



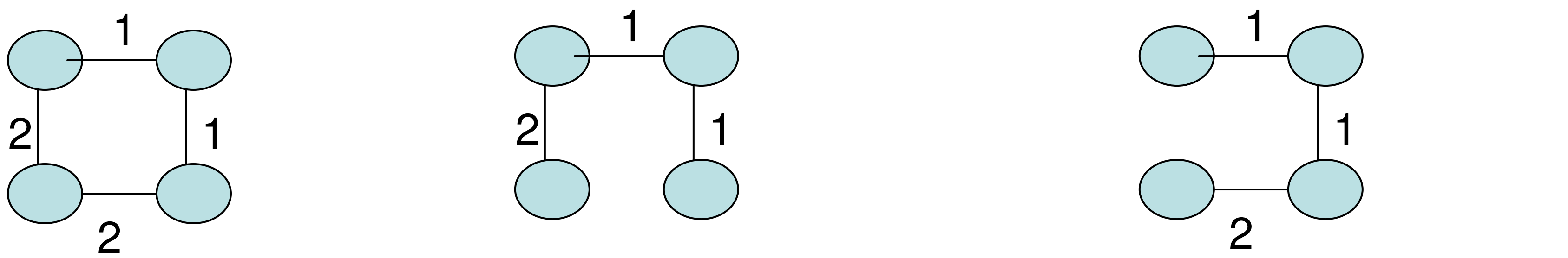
Candidate edges to merge the two components



Modified T ' = T ' – {e'} U {e}

Properties of Minimum Spanning Tree

- **<u>Property 2:</u>** If a graph does not have unique edge weights, there could be more than one minimum spanning tree for the graph.
- **Proof (by Example)**



Graph One Min. Spanning Tree Another Min. Spanning Tree

- **<u>Property 3:</u>** If all the edges in a weighted graph have unique weights, then there can be only one minimum spanning tree of the graph.
- **Proof:** Consider a graph G whose edges are of distinct weights. Assume there are two different spanning trees *T* and T', both are of minimum weight; but have at least one edge difference. Let e' be an edge in T' that is not in T. Removing e' from T' will split the latter into two components. There should be an edge e that is not part of T' but part of T and should also be a candidate edge to connect the two components of the split T'.

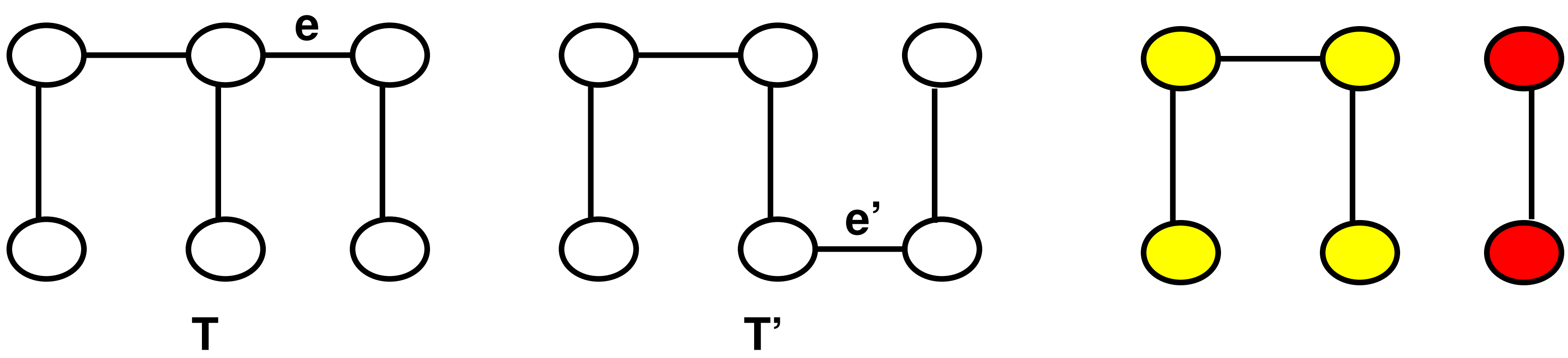
Properties of Minimum Spanning Tree

- **Property 3:** If all the edges in a weighted graph have unique weights, then there can be only one minimum spanning tree of the graph.
- **Proof (continued..):** If wt(e) < wt(e'), then we could merge the two components of T' using e and this would lower the weight of T' from what it was before. Hence, wt(e) ≥ wt(e').
- However, since the graph has unique edge weights, wt(e) > wt(e'). But, if this is the case, then we could indeed remove e from T and have e' to merge the two components of T resulting from the removal of e. This would only lower the weight of T from what it was before.
- So, if T and T' have to be two different MSTs ➔ wt(e) = wt(e').
  - This is a contradiction to the given statement that the graph has unique edge weights.
- Not (wt(e) = wt(e')) ➔ Not (T and T' have to be two different MSTs)
- That is, wt(e) ≠ wt(e') ➔ T and T' have to be the same MST.

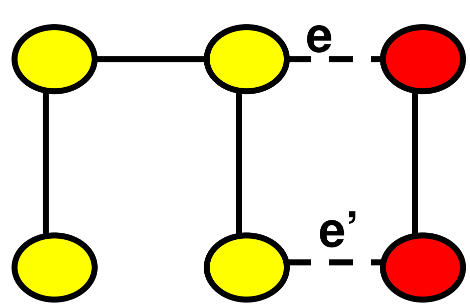
- Hence, if a graph has unique edge weights, there can be only one MST for the graph.

Property 3

Assume that both T and T' are MSTs, but different MSTs to start with.



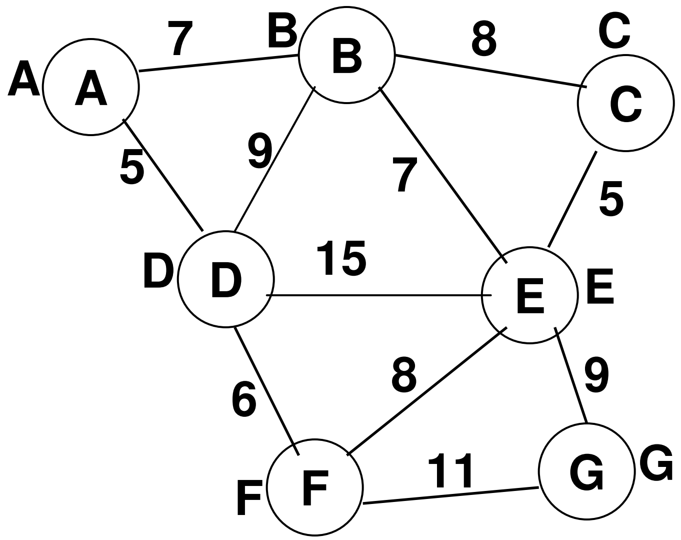
W(e) < W(e') => T' is not a MST
W(e) > W(e') => T is not a MST
Hence, for both T and T' to be two different MSTs ➔ W(e) = W(e').
But the graph has unique edge weights.
W(e) ≠ W(e) ➔ Both T and T' have to be the same.

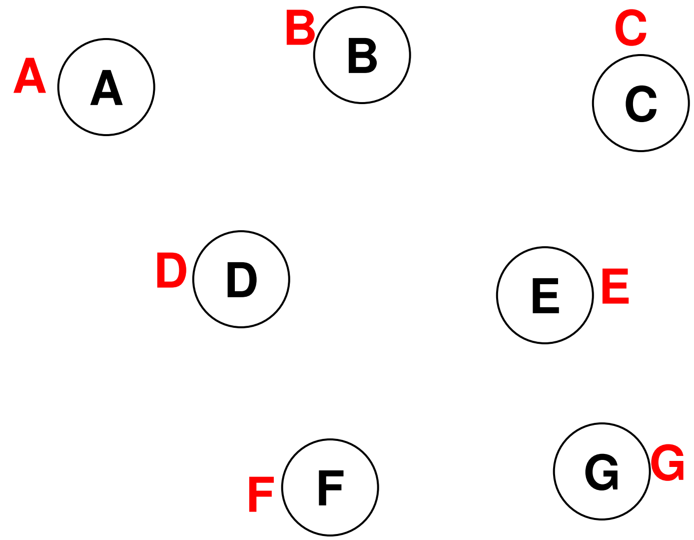


Candidate edges to merge the two components

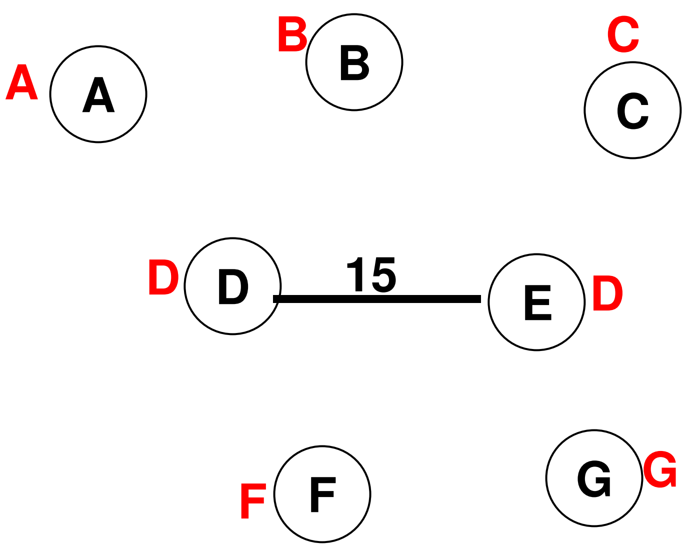
Maximum Spanning Tree

- A Maximum Spanning Tree is a spanning tree such that the sum of its edge weights is the maximum.
- We can find a Maximum Spanning Tree through any one of the following ways:
  - Straightforward approach: Run Kruskal's algorithm by selecting edges in the decreasing order of edge weights (i.e., edge with the largest weight is chosen first) as long as the end vertices of an edge are in two different components
  - Alternate approach (Example for Transform and Conquer): Given a weighted graph, set all the edge weights to be negative, run a minimum spanning tree algorithm on the negative weight graph, then turn all the edge weights to positive on the minimum spanning tree to get a maximum spanning tree.

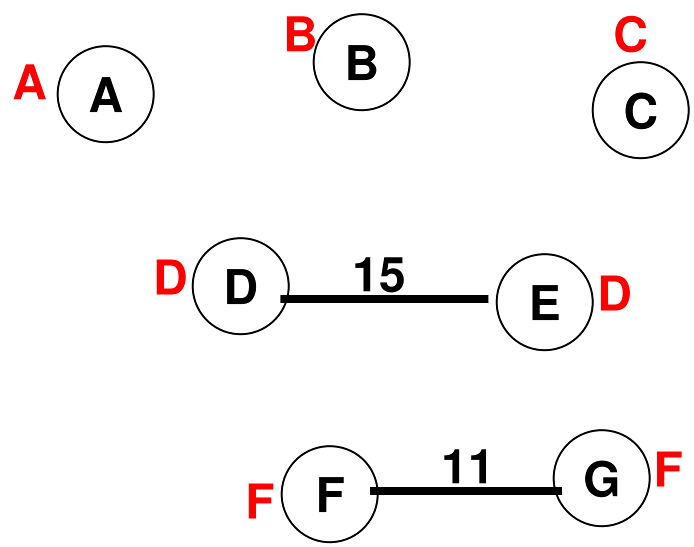




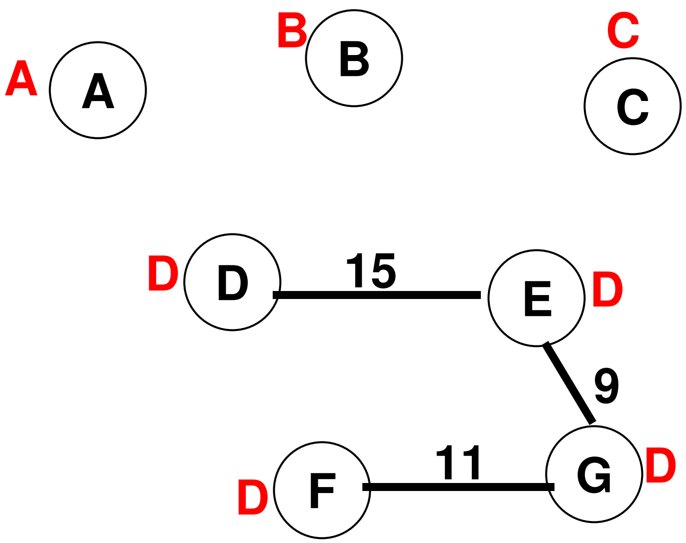
Initialization



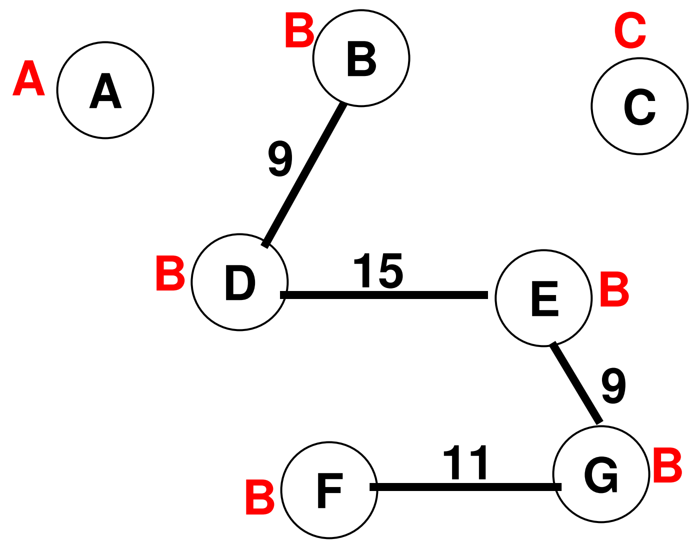
Iteration 1



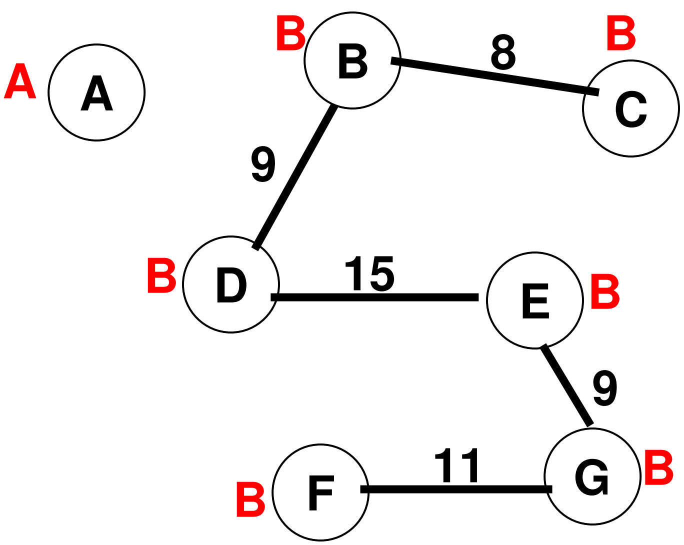
Iteration 2



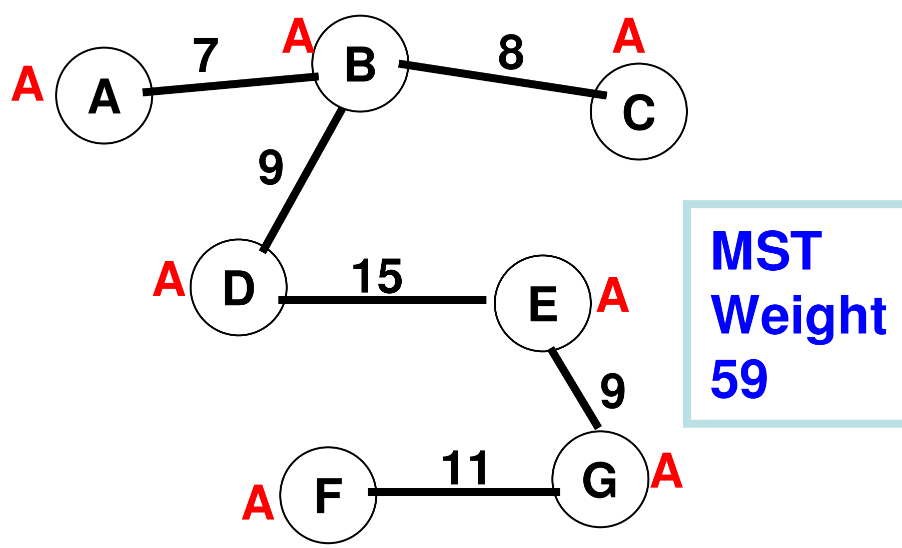
Iteration 3



Iteration 4



Iteration 5



Iteration 6: Max. Sp Tree

Practice Proofs

- Similar to the proof of correctness that we saw for the Minimum Spanning Trees, write the proof of correctness for the Kruskal's algorithm to find Maximum Spanning Trees.

- Prove the following property: If all the edges in a weighted graph have unique weights, then there can be only one maximum spanning tree of the graph.

Dijkstra’s Shortest Path Algorithm

Shortest Path (Min. Wt. Path) Problem

- Path p of length k from a vertex s to a vertex d is a sequence $(v_0, v_1, v_2, \ldots, v_k)$ of vertices such that $v_0 = s$ and $v_k = d$ and $(v_{i-1}, v_i) \in E$, for $i = 1, 2, \ldots, k$
- Weight of a path $p = (v_0, v_1, v_2, \ldots, v_k)$ is $w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i)$
- The weight of a shortest path from s to d is given by

$$\delta(s, d) = \min \{w(p) : s \xrightarrow{p} d\} \text{ if there is a path from } s \text{ to } d$$

$$= \infty \quad \text{otherwise}$$

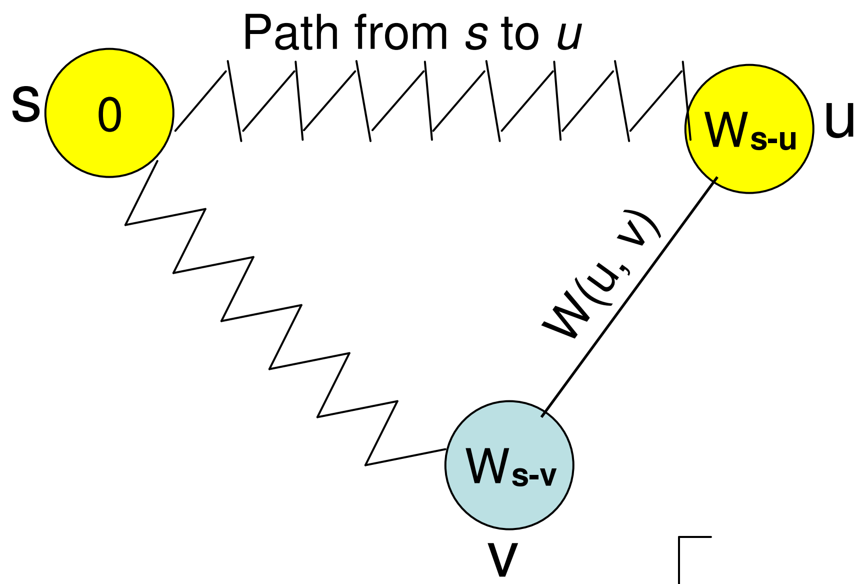
:

Dijkstra Algorithm

- **Assumption:** $w(u, v) > 0$ for each edge $(u, v) \in E$ (i.e., the edge weights are positive)
- **Objective:** Given $G = (V, E, w)$, find the shortest weight path between a given source s and destination d
- **Principle:** Greedy strategy
- Maintain a minimum weight path estimate $d[v]$ from s to each other vertex v.
- At each step, pick the vertex that has the smallest minimum weight path estimate
- **Output:** After running this algorithm for $|V|$ iterations, we get the shortest weight path from s to all other vertices in G
- **Time Complexity:** Dijkstra algorithm – $\Theta(|E| * log|V|)$

Dr. Meg's YouTube Video Explanation:
https://www.youtube.com/watch?v=V8VxK1cr0x0

Principle of Dijkstra Algorithm

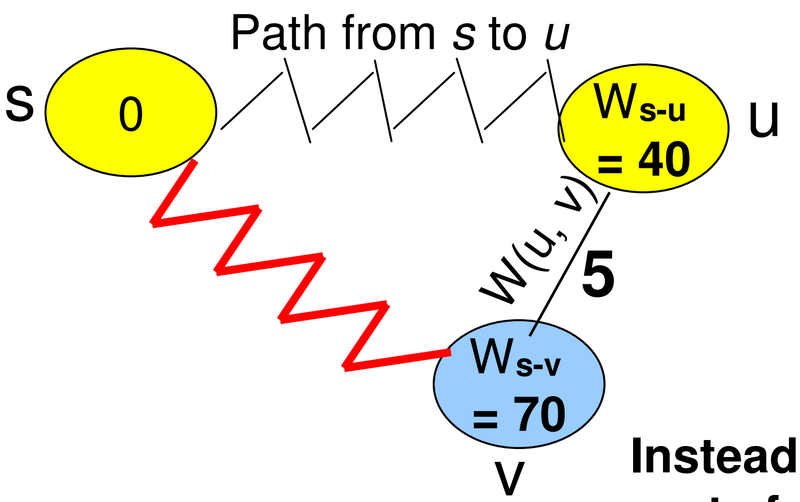


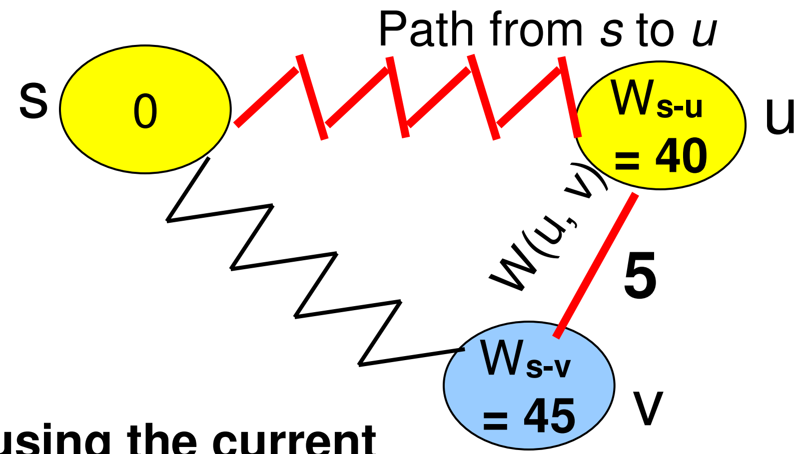
Principle in a nutshell

During the beginning of each iteration we will pick a vertex *u* that has the minimum weight path to *s*. We will then explore the neighbors of *u* for which we have not yet found a minimum weight path. We will try to see if by going through *u*, we can reduce the weight of path from *s* to *v*, where *v* is a neighbor of *u*.

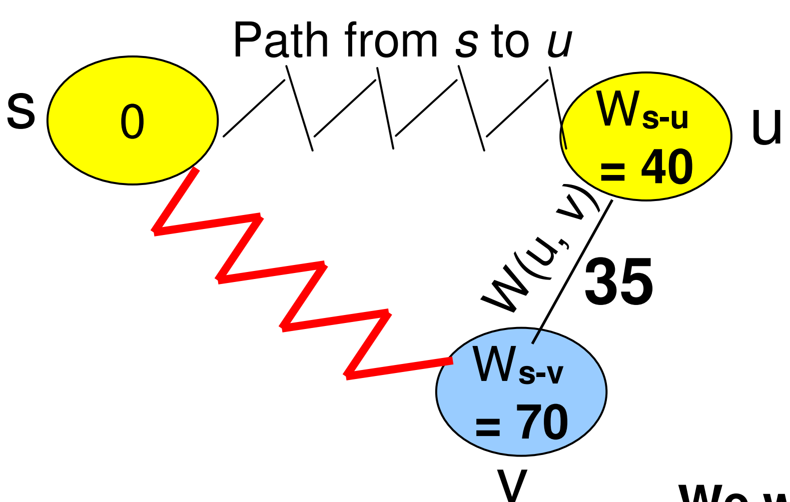
Relaxation Condition

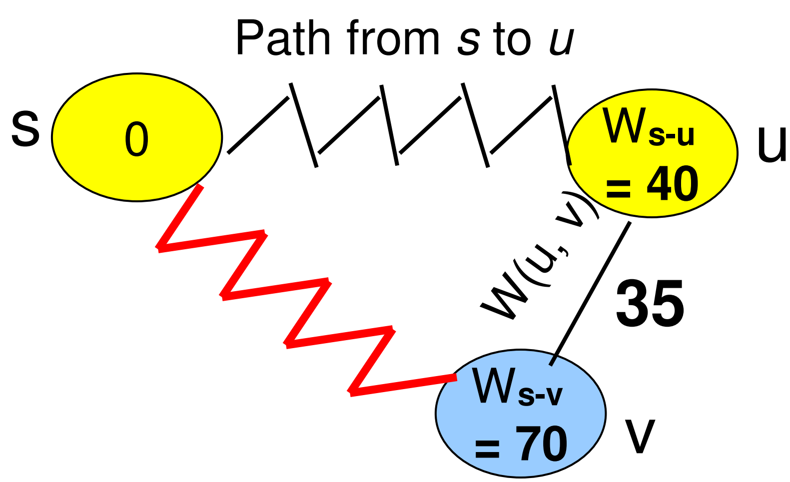
If $W_{s-v} > W_{s-u} + W(u, v)$ then
  $W_{s-v} = W_{s-u} + W(u, v)$
  Predecessor $(v) = u$
else
  Retain the current path from *s* to *v*



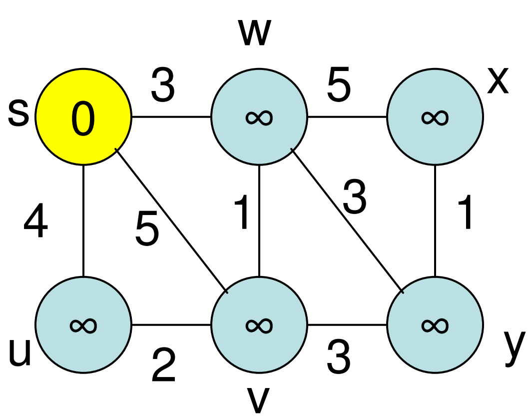


Instead of using the current route from s to v, we will go through u to reach v from s

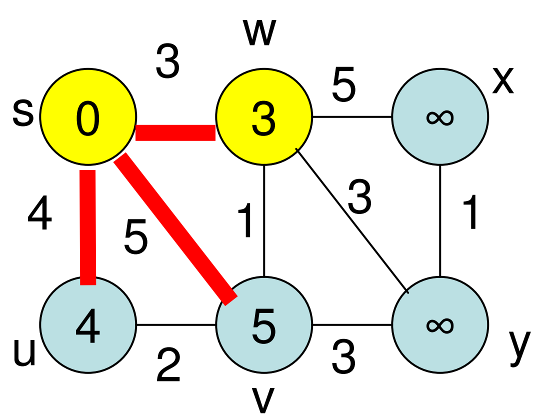




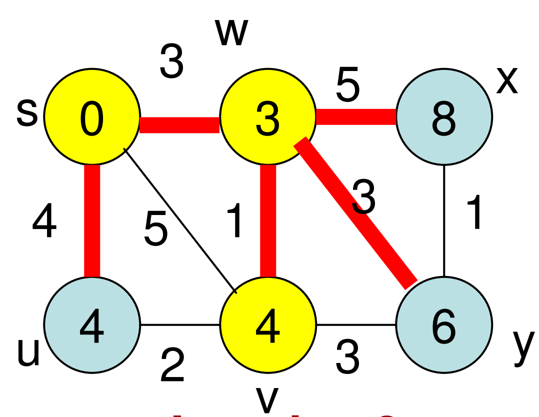
We will stay with the current route we know from s to v.



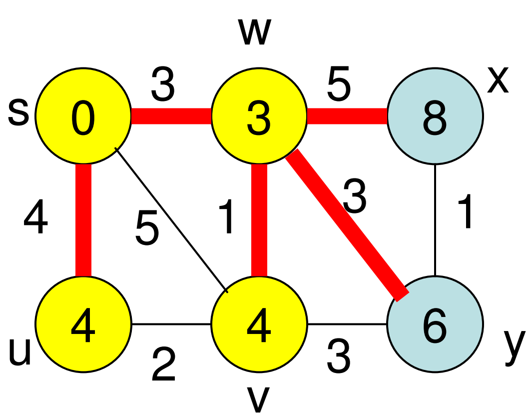
Given Graph, Initialization



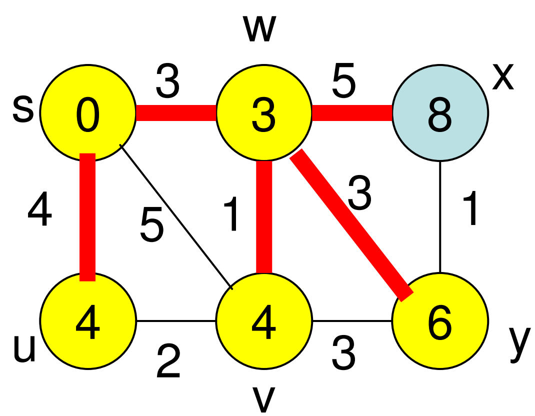
Iteration 1



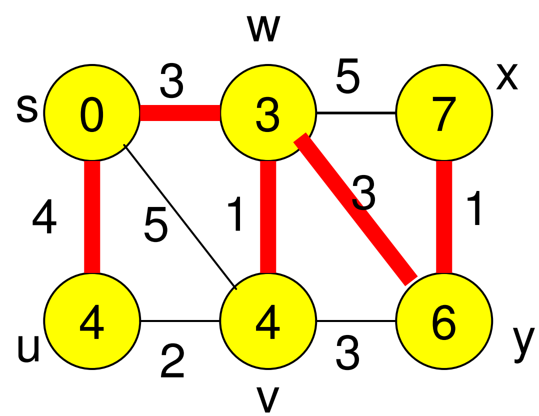
Iteration 2



Iteration 3



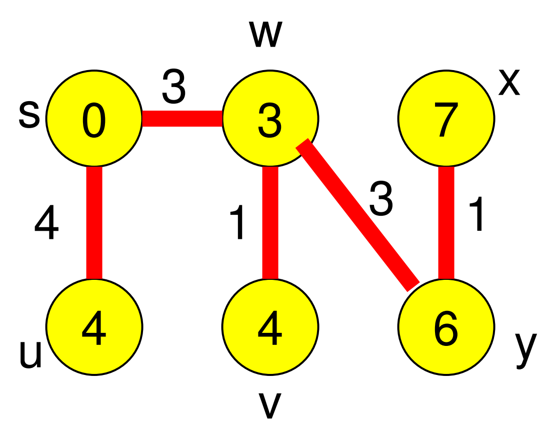
Iteration 4

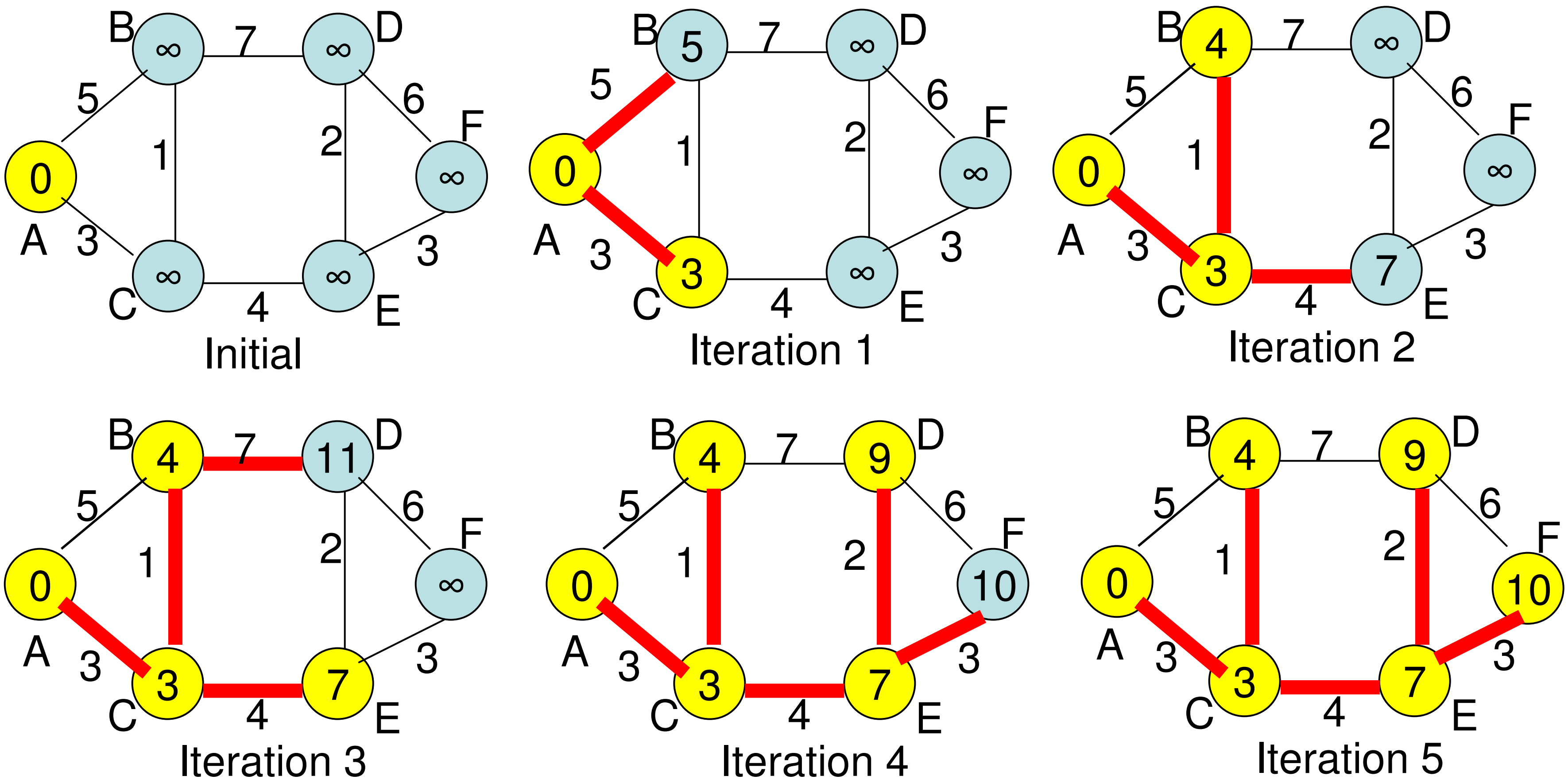


Iteration 5

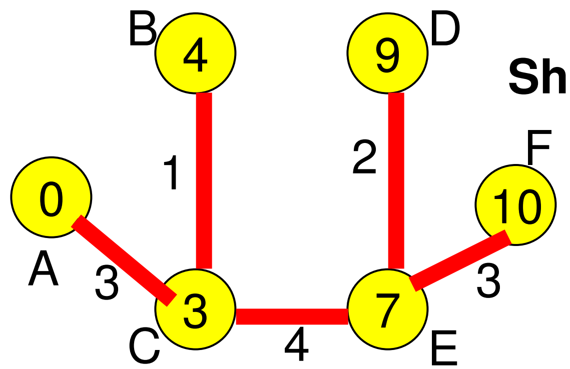
Dijkstra Algorithm Example 1

Shortest Path Tree





Dijkstra Algorithm Example 2



Shortest Path Tree

Dijkstra Algorithm

```
Begin Algorithm Dijkstra (G, s)
    For each vertex v Є V
        d [v] ← ∞ // an estimate of the min-weight path from s to v
    End For
    d [s] ← 0
    S ← Φ  // set of nodes for which we know the min-weight path from s
    Q ←  V // set of nodes for which we know estimate of min-weight path from s
    While Q ≠ Φ
        u ← EXTRACT-MIN(Q)
        S ← S U {u}
        For each vertex v such that (u, v) Є E
            If v ∈ Q and d [v] > d [u] + w (u, v) then
                d [v] ← d [u] + w (u, v)
                Predecessor (v) = u
            End If
        End For
    End While
End Dijkstra
```

Dijkstra Algorithm: Time Complexity

```
Begin Algorithm Dijkstra (G, s)
    For each vertex v Є V
        d [v] ← ∞ // an estimate of the min-weight path from s to v
    End For
    d [s] ← 0
    S ← Φ  // set of nodes for which we know the min-weight path from s
    Q ← V // set of nodes for which we know estimate of min-weight path from s
    While Q ≠ Φ
        u ← EXTRACT-MIN(Q)
        S ← S U {u}
        For each vertex v such that (u, v) Є E
            If v ∈ Q and d [v] > d [u] + w (u, v) then
                d [v] ← d [u] + w (u, v)
                Predecessor (v) = u
            End If
        End For
    End While
End Dijkstra
```

Θ(V) time (lines "For each vertex v Є V" … "End For")

Θ(V) time to Construct a Min-heap (Q ← V)

done |V| times = Θ(V) time (While Q ≠ Φ)

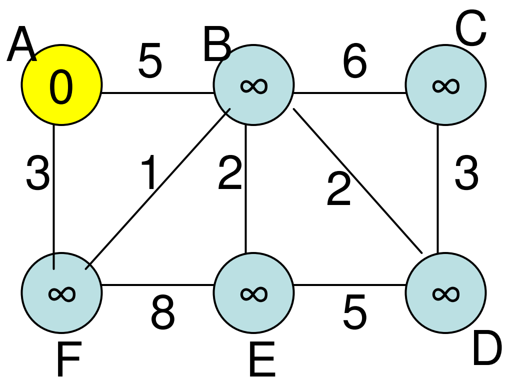
Each extraction takes Θ(logV) time (u ← EXTRACT-MIN(Q))

done Θ(E) times totally (For each vertex v such that (u, v) Є E)

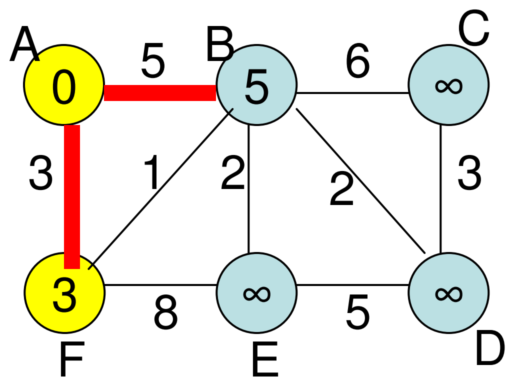
It takes Θ(logV) time when done once (If … Predecessor (v) = u)

Overall Complexity: **Θ(V) + Θ(V) + Θ(VlogV) + Θ(ElogV)**
Since the |E| ≥ |V|-1, the VlogV term is dominated by the ElogV term. Hence, overall complexity = Θ(|E|*log|V|)

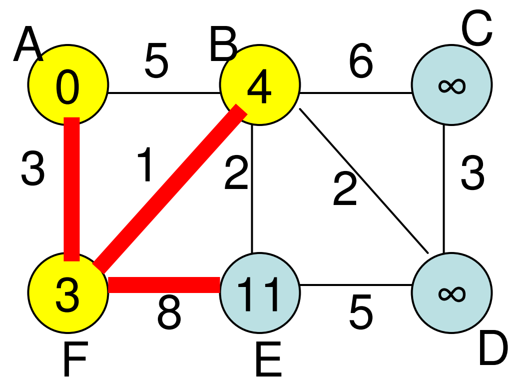
Dijkstra Algorithm Example 3



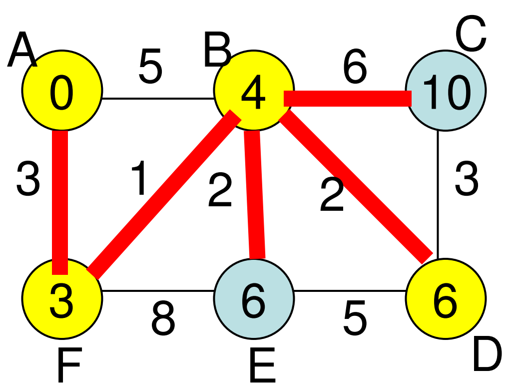
Initial



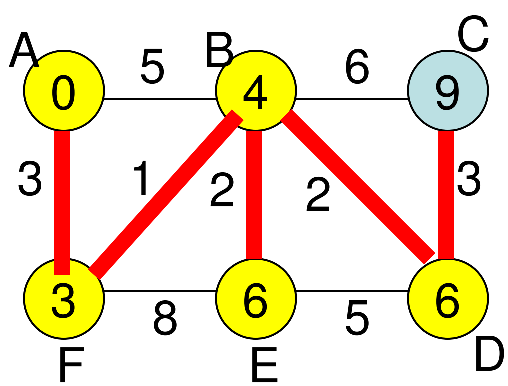
Iteration 1



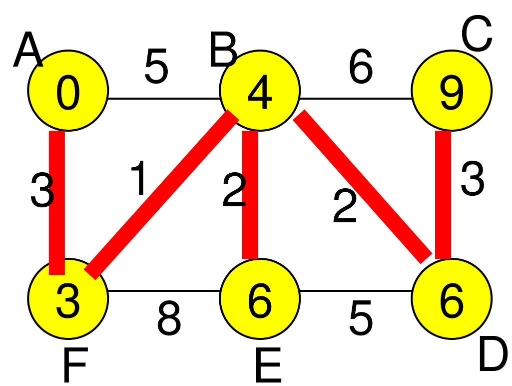
Iteration 2



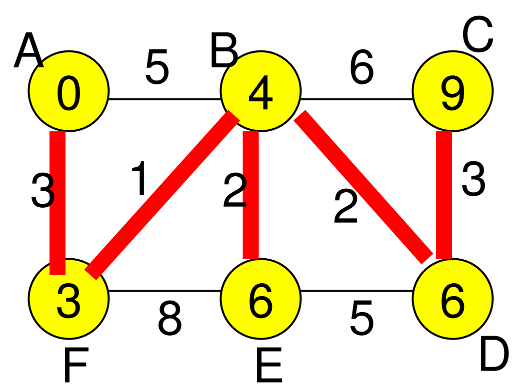
Iteration 3



Iteration 4



Iteration 5



Shortest Path Tree

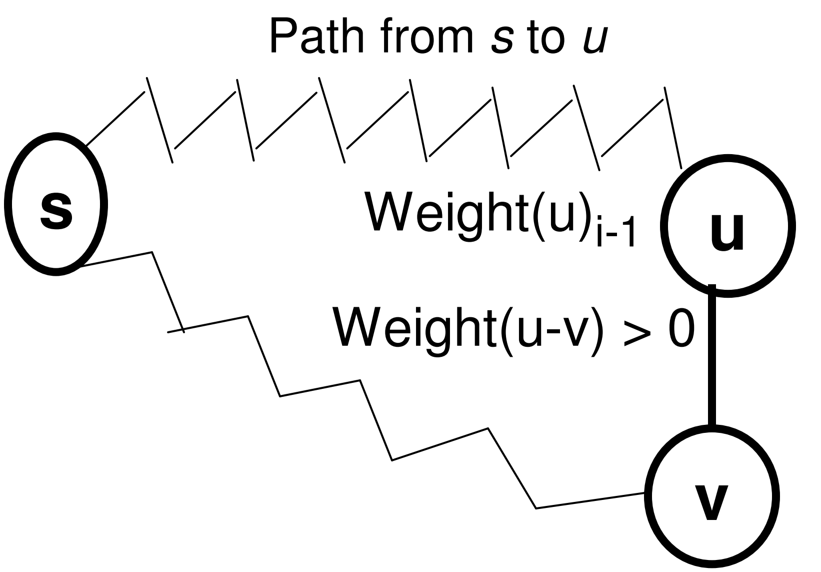
Theorems on Shortest Paths and Dijsktra Algorithm

- **Theorem 1:** Sub path of a shortest path is also shortest.
- **Proof:** Lets say there is a shortest path from s to d through the vertices s – a – b – c – d.
- Then, the shortest path from a to c is also a – b – c.
- If there is a path of lower weight than the weight of the path from a – b – c, then we could have gone from s to d through this alternate path from a to c of lower weight than a – b – c.
- However, if we do that, then the weight of the path s – a – b – c – d is not the lowest and there exists an alternate path of lower weight.
- This contradicts our assumption that s – a – b – c – d is the shortest (lowest weight) path.

Theorems on Shortest Paths and Dijsktra Algorithm

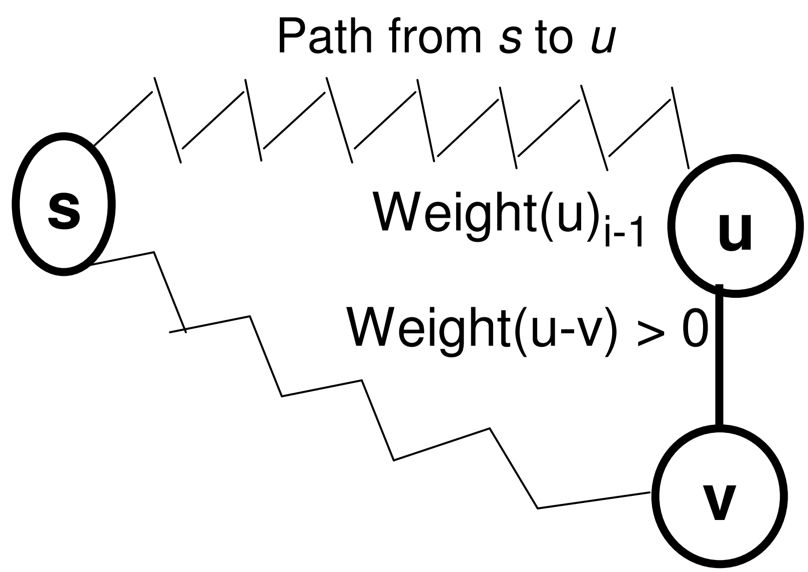
- **Theorem 2:** The weights of the vertices that are optimized are in the non-decreasing (i.e., typically increasing) order.
- **Proof:** We want to prove that if a vertex u is optimized in an earlier iteration (say iteration i), then the weight of the vertex v optimized at a later iteration (say iteration j; $i < j$) is always greater than or equal to that of vertex u.
- Vertex v could be either a neighbor of vertex u or not. In either case, the $weight(v)_{i-1} \geq weight(u)_{i-1}$ during the beginning of iteration i as vertex u was considered to have been optimized instead of vertex v during this iteration.
- During iteration i: we relax the neighbors of vertex u
  - If vertex v is a neighbor of vertex u, $weight(v)_i$ could have become less than $weight(v)_{i-1}$, but $weight(v)_i$ could never become $weight(u)_{i-1}$ as all edge weights are positive (including the weight of the edge u-v). Hence, $weight(v)_i$ could have become $weight(u)_{i-1}$ + weight(u-v), but it will still be only less than $weight(u)_{i-1}$, as weight(u-v) > 0.
- If vertex v is not a neighbor of vertex u, then vertex v should ultimately get optimized through some neighbor x (that is not u). But all such neighbors x should have $weight(x)_{i-1} \geq weight(u)_{i-1}$, as x was not picked for optimization in iteration i. Hence, by going through such neighbors x, the weight(v) during iterations i or later, could never become still less than $weight(u)_{i-1}$, as all the edge weights w(x-v) are greater than 0.

Proof for Theorem 2
Scenario: Vertex v is a neighbor of Vertex u



Before Iteration i (at the end of Iteration i-1)

$\text{Weight}(v)_{i-1} \geq \text{Weight}(u)_{i-1}$

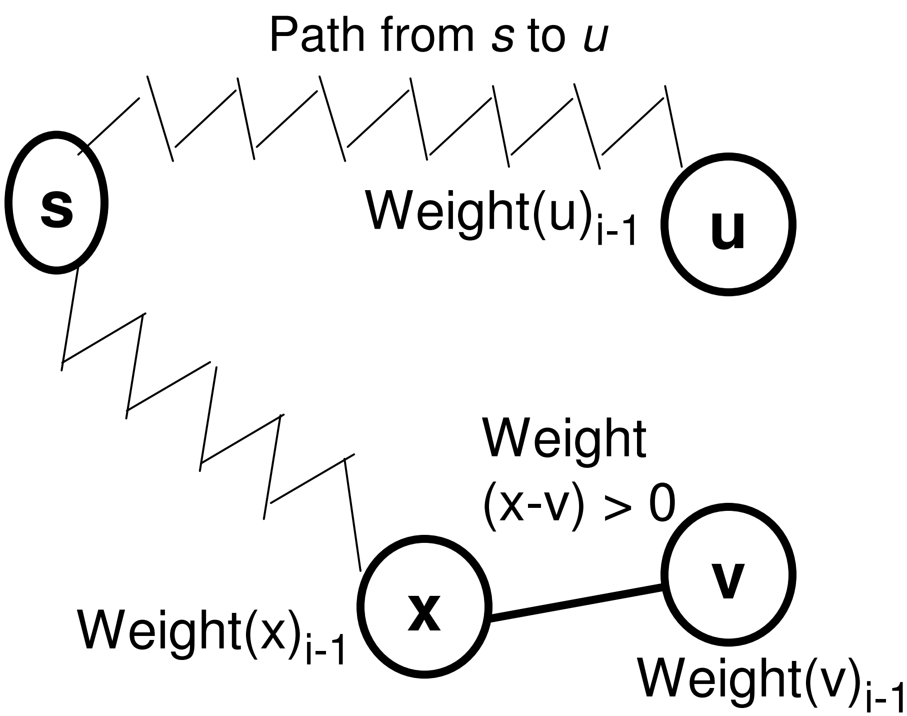


At the end of Iteration i

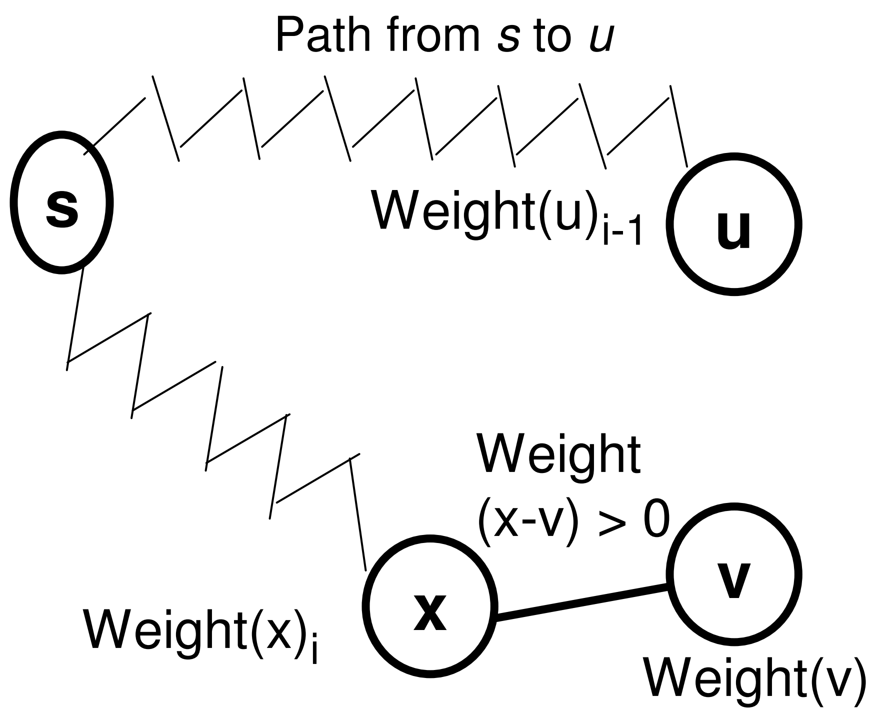
$\text{Weight}(v)_{i} \geq \text{Weight}(u)_{i-1}$

Proof for Theorem 2

Scenario: Vertex v is NOT a neighbor of Vertex u, but a neighbor of some other vertex x



Before Iteration i (at the end of Iteration i-1)



At the end of Iteration i

$\text{Weight}(x)_{i-1} \geq \text{Weight}(u)_{i-1}$

$\text{Weight}(v)_{i-1} \geq \text{Weight}(u)_{i-1}$

$\text{Weight}(x)_{i} \geq \text{Weight}(u)_{i-1}$

$\text{Weight}(v)_{i} \geq \text{Weight}(u)_{i-1}$

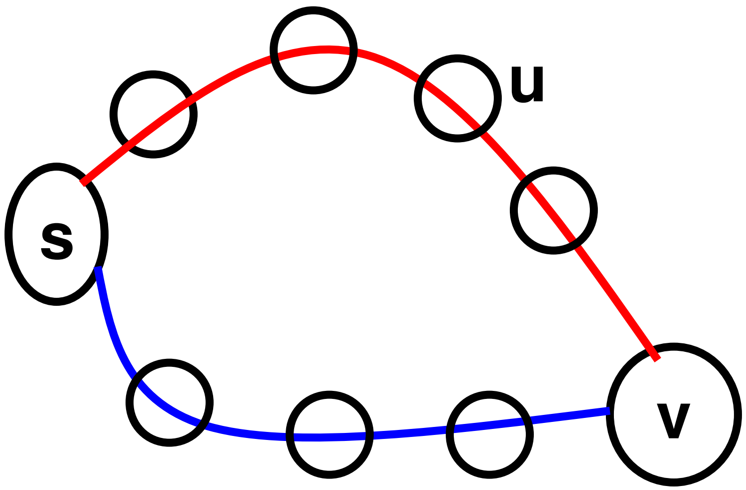
Theorems on Shortest Paths and Dijsktra Algorithm

- **Theorem 3:** When a vertex v is picked for relaxation/optimization, every intermediate vertex on the s…v shortest path is already optimized.
- **Proof:** Let there be a path from s to v that includes a vertex *x* (i.e., s...x...v) for which we have not yet found the shortest path.
- From Theorem 1, shortest path weight(s...x) < shortest path weight(s...v).
- From Theorem 2, vertices are optimized in the non-decreasing order of shortest path weights.
- So, if vertex v is picked for optimization based on the path s…x…v, then the intermediate vertex x should have been already picked (before v) for optimization. A contradiction.

- **Theorem 4:** When a vertex v is picked for relaxation, we have optimized the vertex (i.e., found the shortest path for the vertex from a source vertex s).
- **Proof:** Let P be the path from source s to vertex v based on whose weight we decide to relax the vertex. We want to prove P is the optimal path of minimum weight from s to v. We will prove this by contradiction.
- Let P' be a hypothetical shortest path from s to v such that w(P') < w(P)
- If all the intermediate vertices from s to v on the path P' are already optimized, we would have indeed found the shortest path from s to v of weight w(P').
- If P' is not chosen and P is chosen by Dijkstra algorithm for optimizing vertex v, then there should be at least one intermediate vertex (say vertex 'u') on the path P' from s to v that is not yet optimized (and because of this we were not able to optimize v from s on path P').
- From the earlier Theorems, the weight(s…u in P') ≥ weight(s…v in P) because the algorithm picks vertices for optimization in the non-decreasing (i.e., increasing) order of shortest path weights.
  - So, even if vertex u on path P' is chosen for optimization after vertex v on path P, the weight of the s…u…v path (P') would be only larger than that of the s…v path (P). Hence, a contradiction.
- Thus, the path P found by Dijkstra algorithm is the shortest path from the source s to a vertex v.

Proof for Theorem 4 (by Contradiction)

Hypothetical Path P' that We assume:
$Weight(s...v)_{P'} < Weight(s...v)_{P}$



Path P found by Dijkstra algorithm

<u>From Theorem 3,</u>
If P' is an optimal path from s to v, then all the intermediate vertices on the path should have been already optimized, and as a result of the accompanying relaxations, we would have traced the path P' from s to v as the optimal path instead of the path P. Hence, if the algorithm did not pick P' as the optimal path, there should be some intermediate vertex u on the path P' that is not yet optimized and all the subsequent vertices on the path P' are not optimized either.

<u>From Theorem 2,</u>
$Weight(s...u)_{P'} \geq Weight(s...v)_{P}$

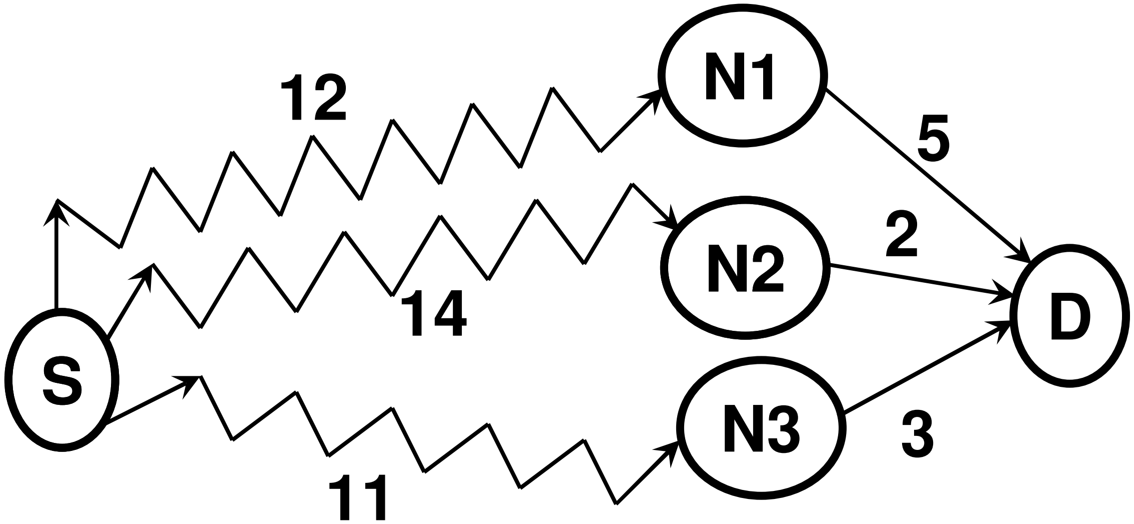
<u>From Theorem 1,</u>
$Weight(s...u...v)_{P'} > Weight(s...u)_{P'}$

<u>Hence:</u>
$Weight(s...u...v)_{P'} > Weight(s...v)_{P}$

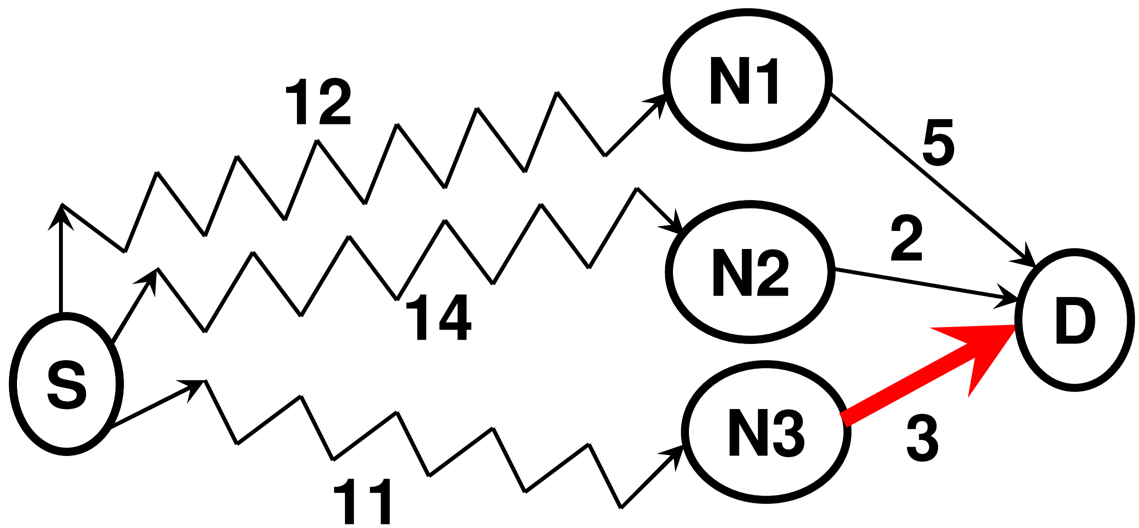
Bellman-Ford Algorithm

- The Bellman-Ford algorithm is a single source shortest path algorithm that can be run for weighted directed graphs with positive and/or negative edge weights.
  - Note that the Dijkstra algorithm will work only for graphs with positive edge weights, and is typically applied for undirected graphs.
- The Bellman-Ford algorithm maintains an estimate of the shortest path distance from the source to every vertex (including itself) and tries to reduce the estimate as much as possible by a going through series of iterations.
  - In each iteration, we try to reduce the estimate of the shortest path distance for a node on the basis of the estimate of the shortest path distance for its INCOMING neighbors (calculated in the previous iteration).
    - The incoming neighbor node that gives the smallest value for the estimate is chosen/updated as the predecessor.
  - We go through a series of V-1 iterations for a graph of V vertices.
  - <u>Optimization:</u> If the estimates for the shortest path distances do not change for any vertex during an iteration, stop the algorithm.

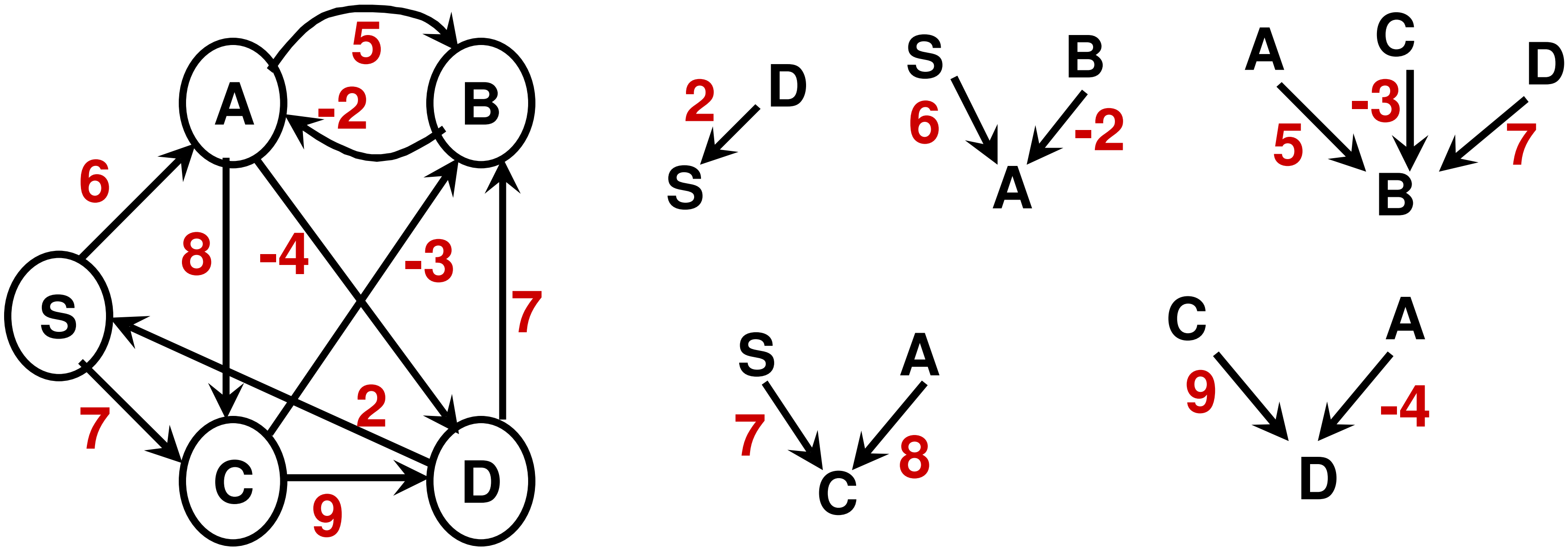
Bellman-Ford Algorithm
Operating Principle



11 + 3 = 14 is lower than 12 + 5 and 14 + 2. So, N3 is chosen as the Predecessor for D

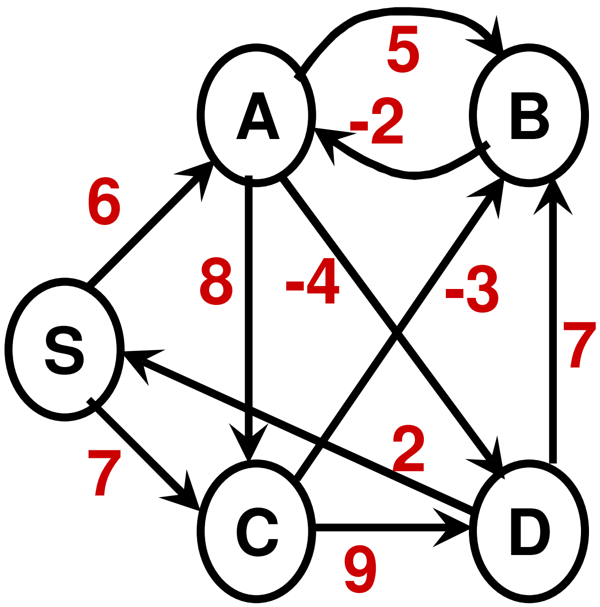


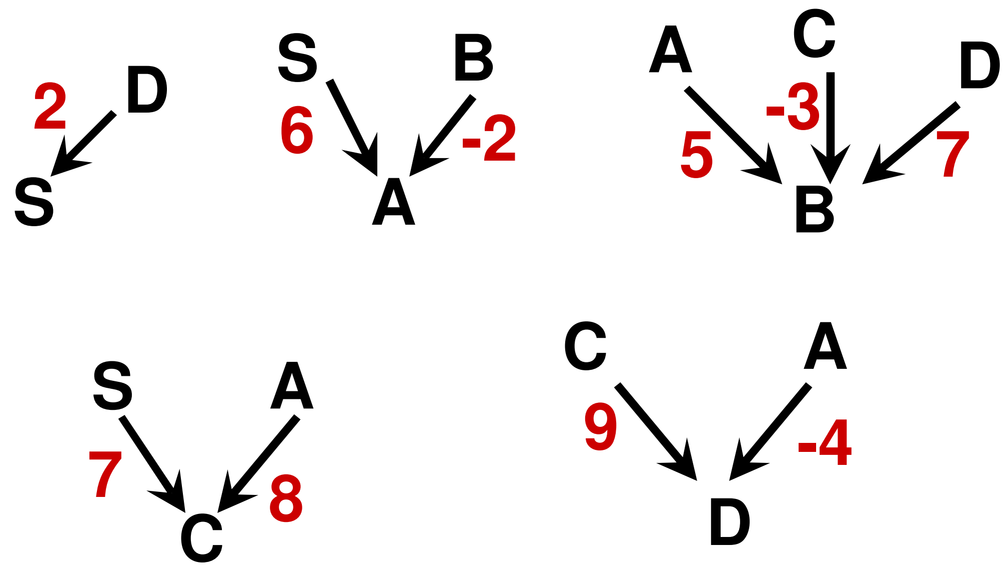
Bellman-Ford Algorithm: Example 1



	Initial	
	Est.	Pred
S	0	-
A	inf	-
B	inf	-
C	inf	-
D	inf	-

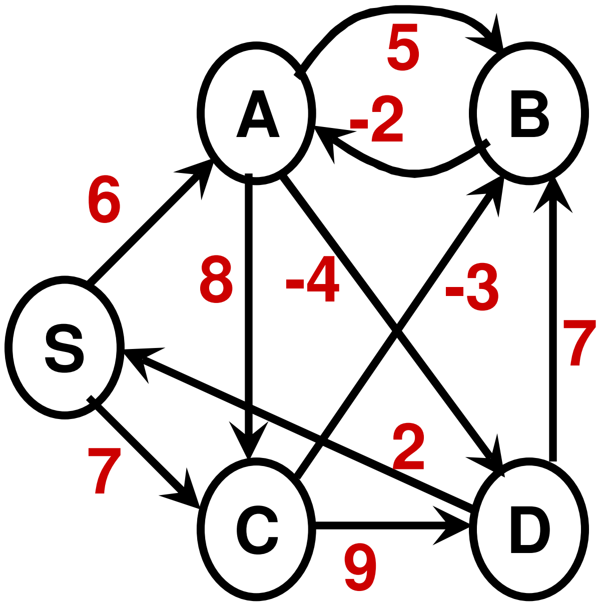
Bellman-Ford Algorithm: Example 1

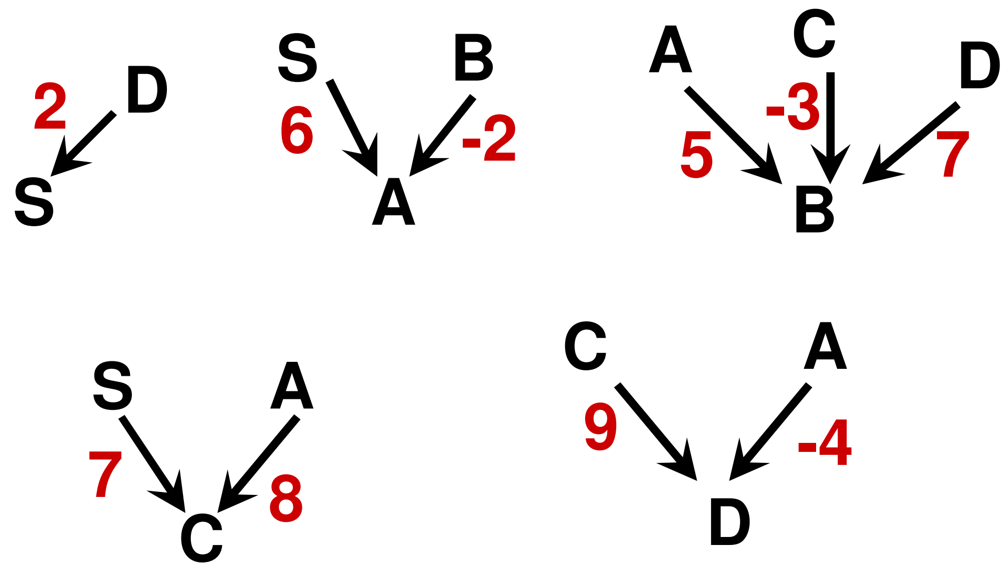




	Initial		Iteration 1	
	Est.	Pred	Est.	Pred
S	0	-	0	-
A	inf	-	6	S
B	inf	-	inf	-
C	inf	-	7	S
D	inf	-	inf	-

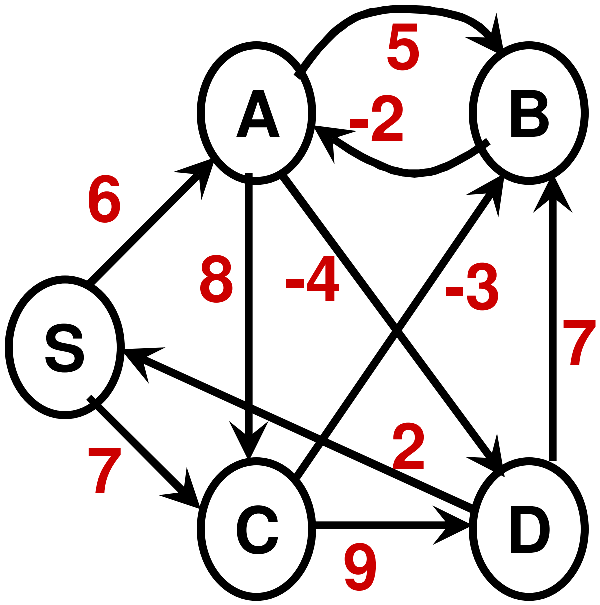
Bellman-Ford Algorithm: Example 1

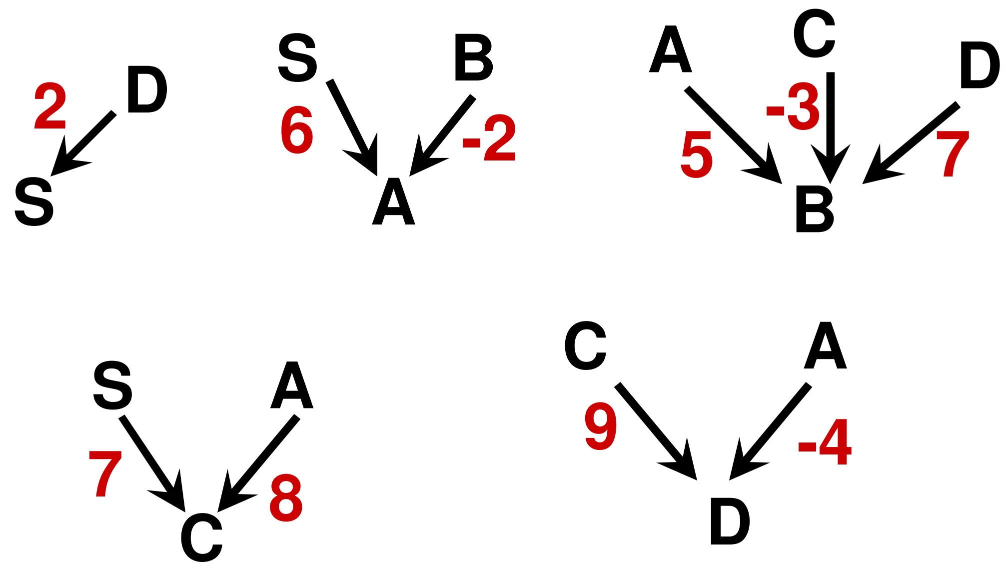




	Iteration 1		Iteration 2	
	Est.	Pred	Est.	Pred
S	0	-	0	-
A	6	S	6	S
B	inf	-	4	C
C	7	S	7	S
D	inf	-	2	A

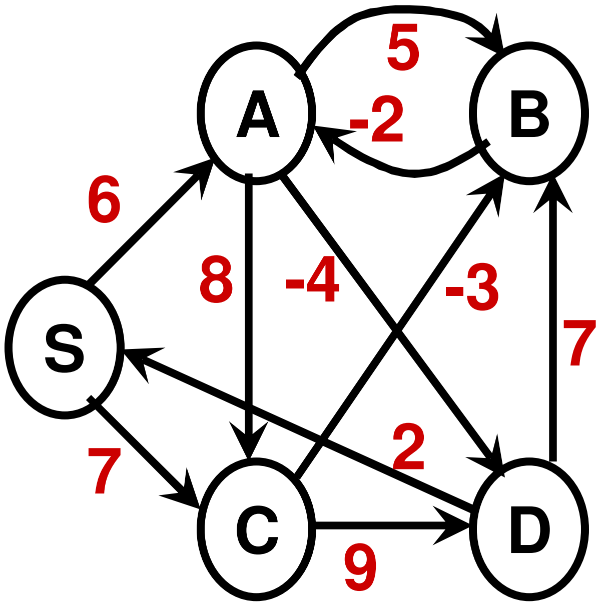
Bellman-Ford Algorithm: Example 1

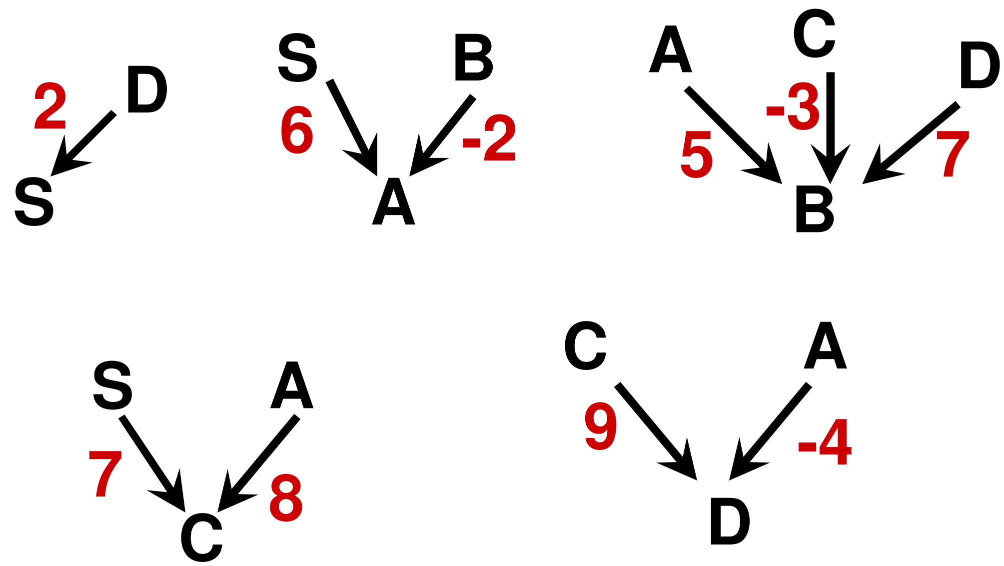




	Iteration 2		Iteration 3	
	Est.	Pred	Est.	Pred
S	0	-	0	-
A	6	S	2	B
B	4	C	4	C
C	7	S	7	S
D	2	A	2	A

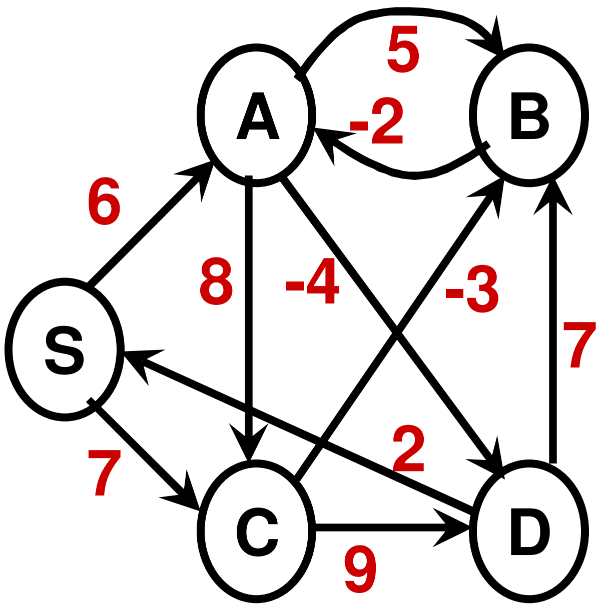
Bellman-Ford Algorithm: Example 1





		Iteration 3		Iteration 4	
		Est.	Pred	Est.	Pred
S		0	-	0	-
A		2	B	2	B
B		4	C	4	C
C		7	S	7	S
D		2	A	-2	A

Bellman-Ford Algorithm: Example 1



Sample Shortest Path (S…D)

S ……………….. A → D

S ……….. B → A → D

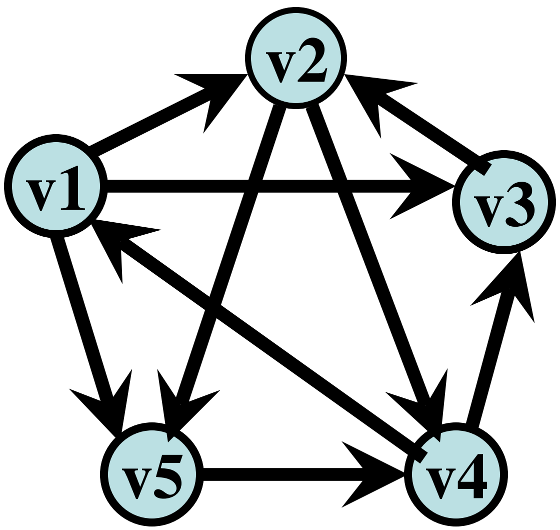
S …. C → B → A → D

S → C → B → A → D

Note that the property "sub path of a shortest path is also a shortest path" is still satisfied.

	Initial		Iteration 1		Iteration 2		Iteration 3		Iteration 4	
	Est.	Pred	Est.	Pred	Est.	Pred	Est.	Pred	Est.	Pred
S	0	-	0	-	0	-	0	-	0	-
A	inf	-	6	S	6	S	2	B	2	B
B	inf	-	inf	-	4	C	4	C	4	C
C	inf	-	7	S	7	S	7	S	7	S
D	inf	-	inf	-	2	A	2	A	-2	A

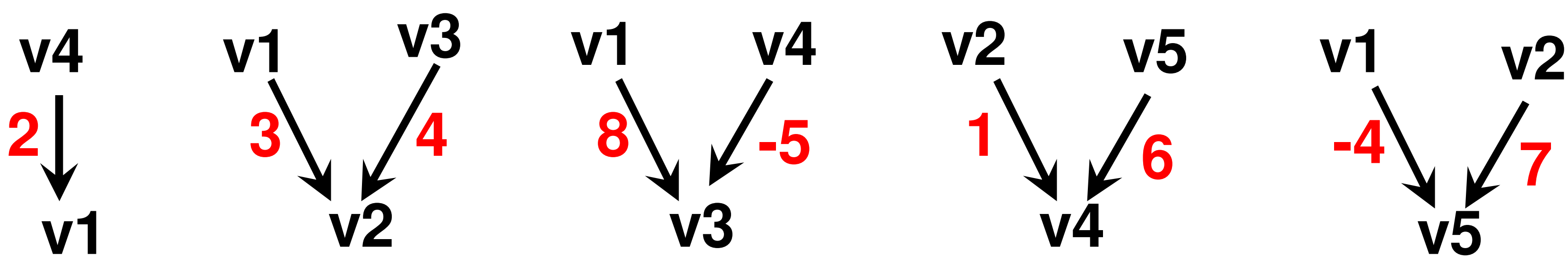
Bellman-Ford Algorithm: Example 2



	Weight Matrix				
	v1	v2	v3	v4	v5
v1	0	3	8	∞	-4
v2	∞	0	∞	1	7
v3	∞	4	0	∞	∞
v4	2	∞	-5	0	∞
v5	∞	∞	∞	6	0

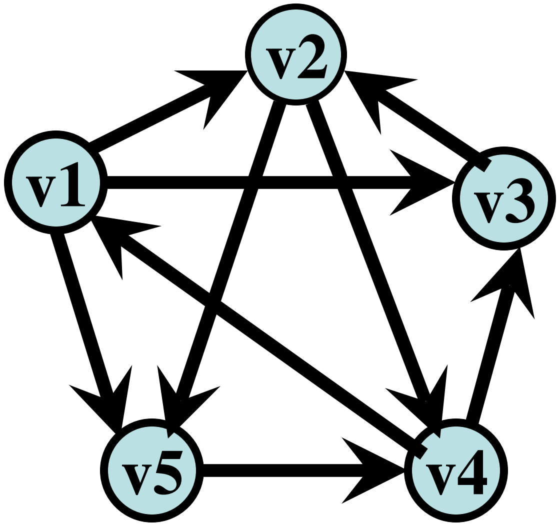
Note: An entry in the cell (i, j) indicates the weight of the edge i → j (i.e., row i, column j).

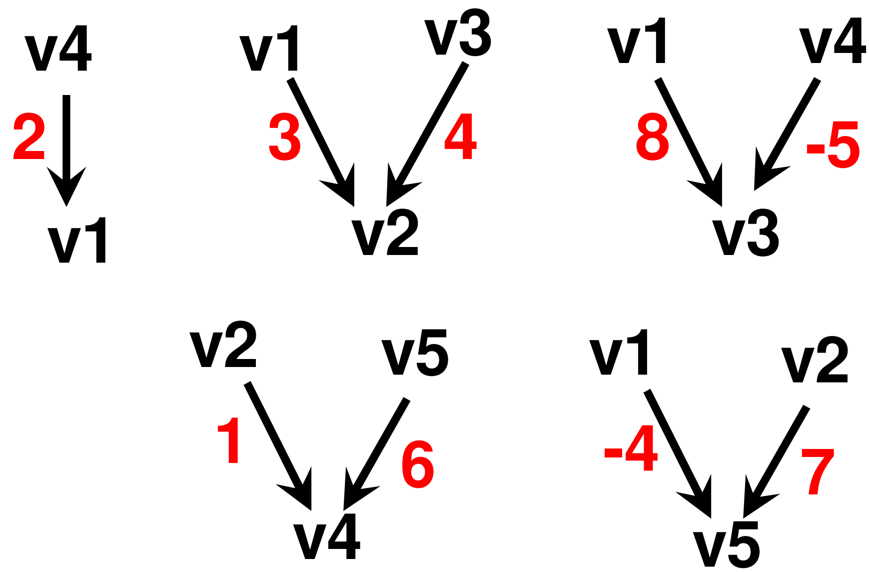
The entries in the column j indicate the weights of the incoming edges to vertex v-j.



Let v1 be the source

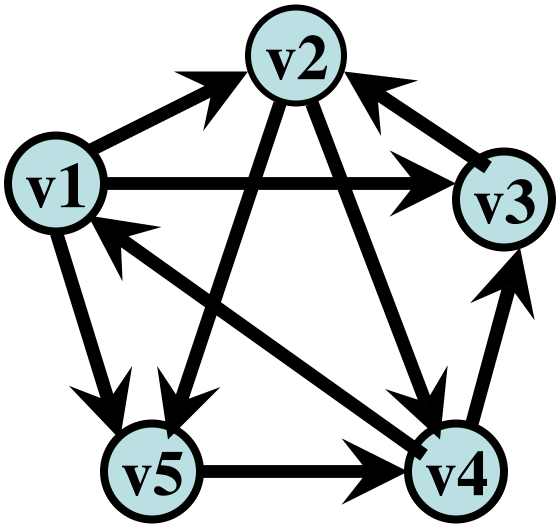
Bellman-Ford Algorithm: Example 2

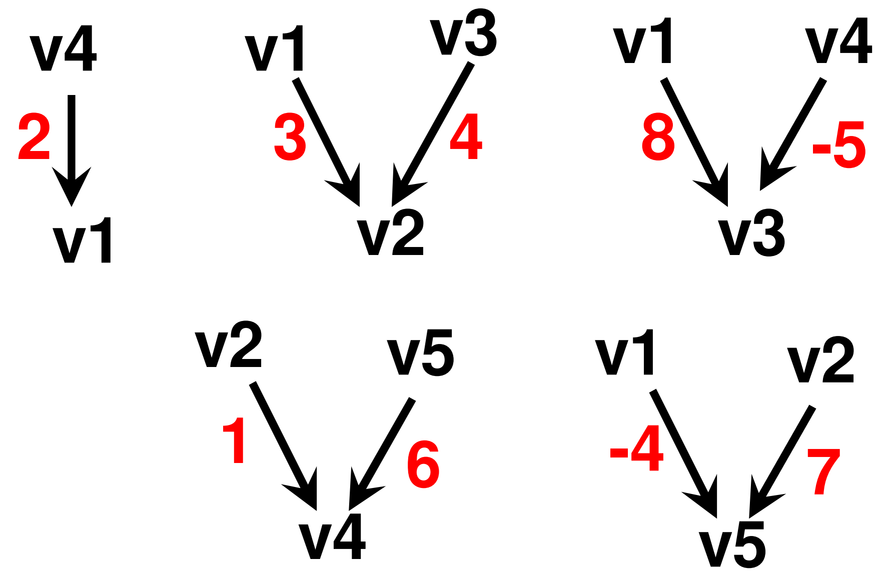




	Initial	
	Est.	Pred
v1	0	-
v2	inf	-
v3	inf	-
v4	inf	-
v5	inf	-

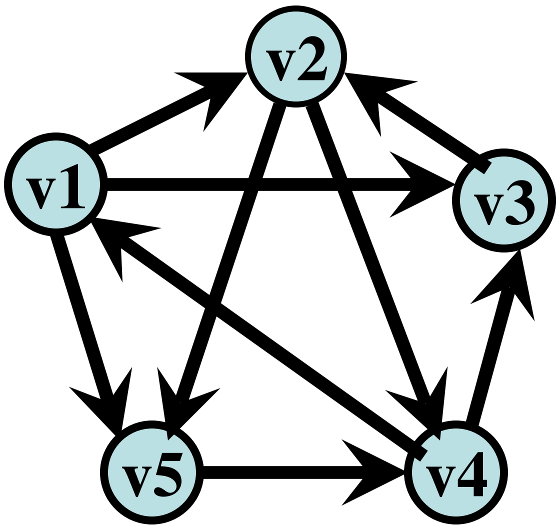
Bellman-Ford Algorithm: Example 2

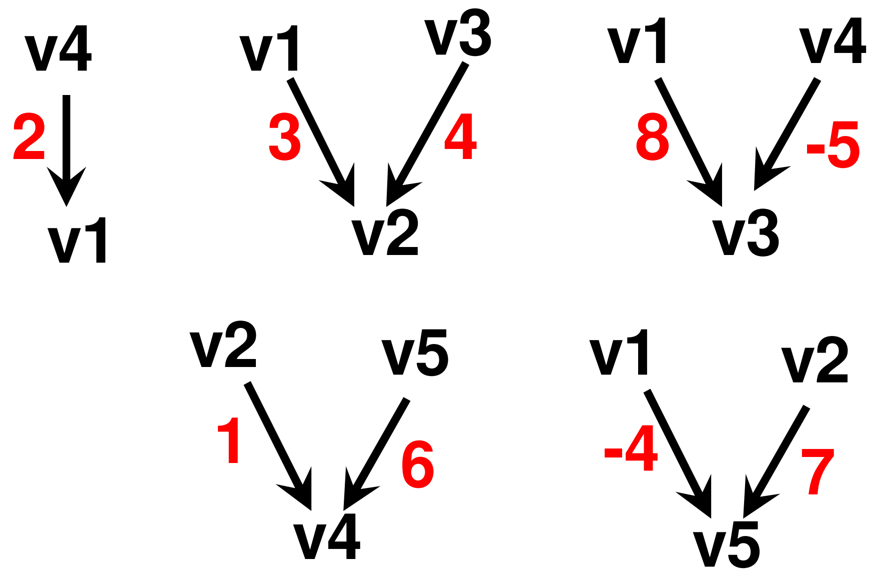




	Initial		Iteration 1	
	Est.	Pred	Est.	Pred
v1	0	-	0	-
v2	inf	-	3	v1
v3	inf	-	8	v1
v4	inf	-	inf	-
v5	inf	-	-4	v1

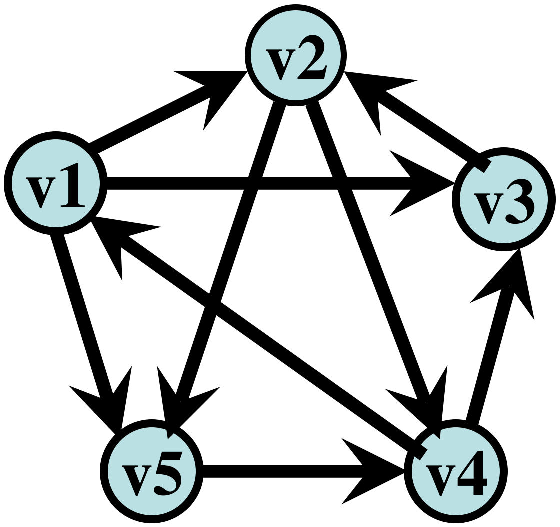
Bellman-Ford Algorithm: Example 2

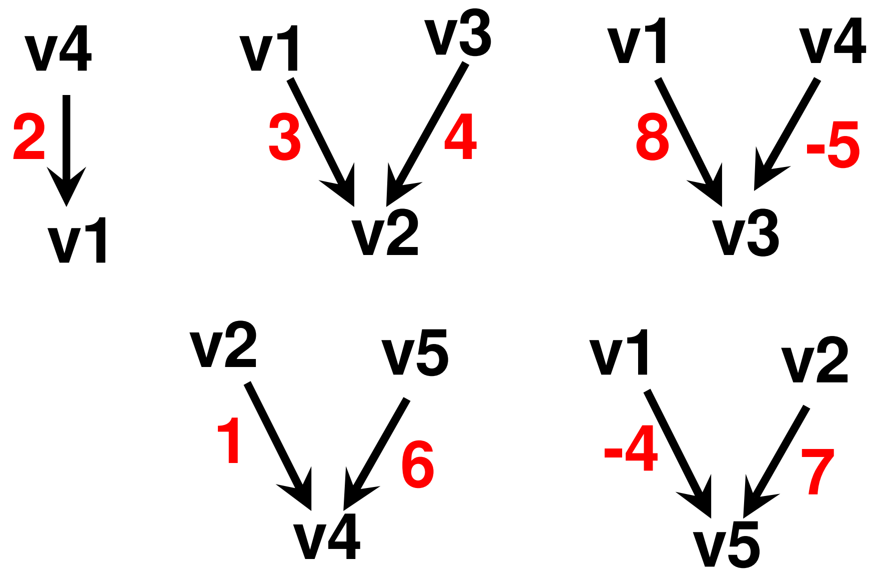




	Iteration 1		Iteration 2	
	Est.	Pred	Est.	Pred
v1	0	-	0	-
v2	3	v1	3	v1
v3	8	v1	8	v1
v4	inf	-	2	v5
v5	-4	v1	-4	v1

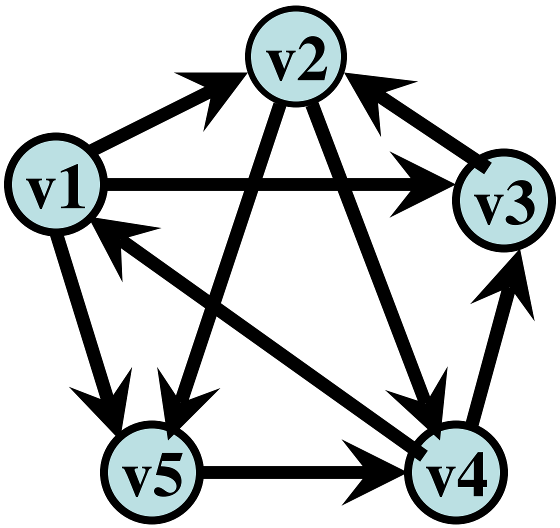
Bellman-Ford Algorithm: Example 2

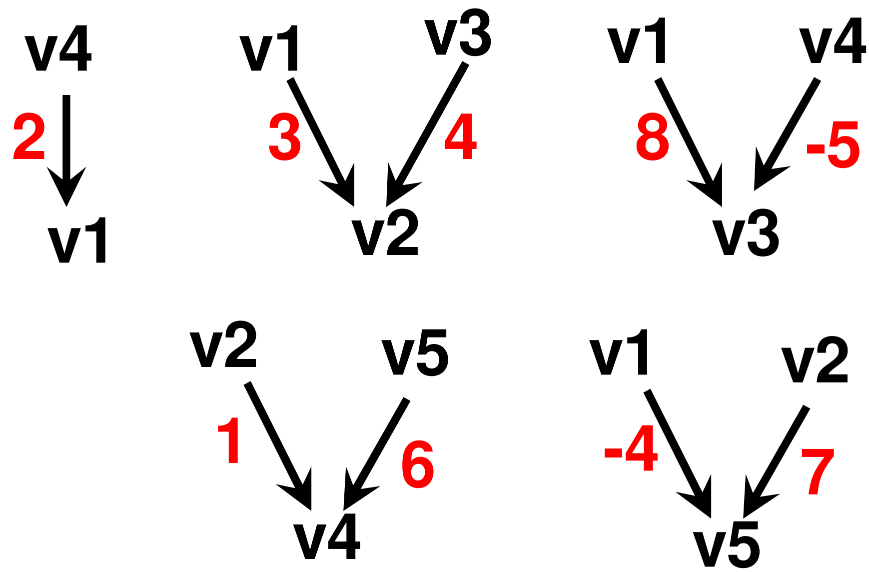




	Iteration 2		Iteration 3	
	Est.	Pred	Est.	Pred
v1	0	-	0	-
v2	3	v1	3	v1
v3	8	v1	-3	v4
v4	2	v5	2	v5
v5	-4	v1	-4	v1

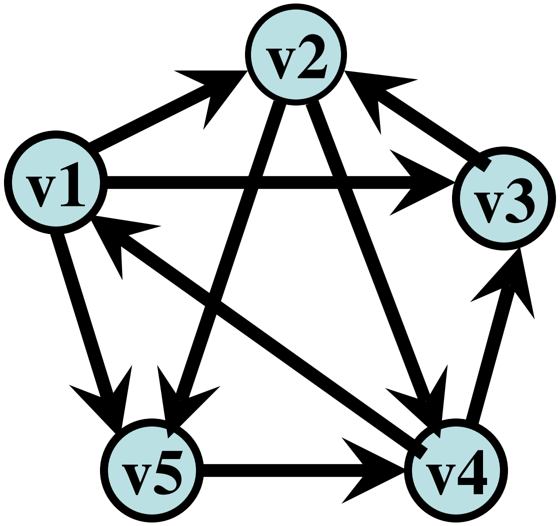
Bellman-Ford Algorithm: Example 2





	Iteration 3		Iteration 4	
	Est.	Pred	Est.	Pred
v1	0	-	0	-
v2	3	v1	1	v3
v3	-3	v4	-3	v4
v4	2	v5	2	v5
v5	-4	v1	-4	v1

Bellman-Ford Algorithm: Example 2



Sample Shortest Path (v1 … v2)

v1 ………………. v3 → v2

v1 ……….. v4 → v3 → v2

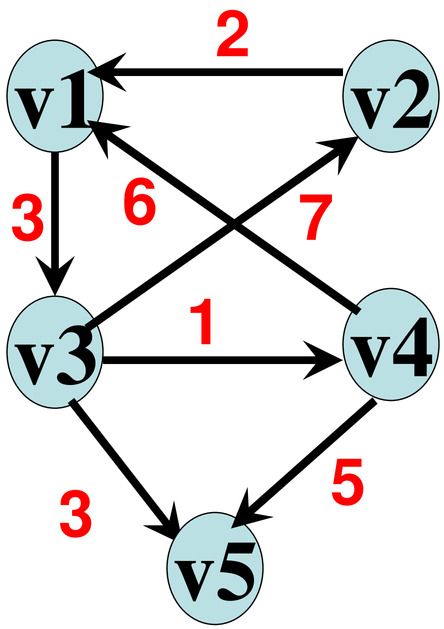
v1 …. v5 → v4 → v3 → v2

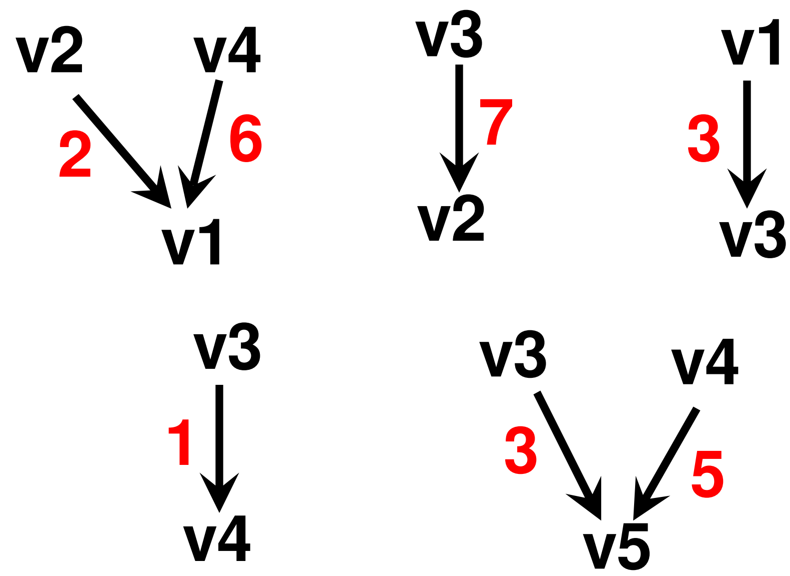
v1 → v5 → v4 → v3 → v2

-4 6 -5 4

	Initial		Iteration 1		Iteration 2		Iteration 3		Iteration 4	
	Est.	Pred	Est.	Pred	Est.	Pred	Est.	Pred	Est.	Pred
v1	0	-	0	-	0	-	0	-	0	-
v2	inf	-	3	v1	3	v1	3	v1	1	v3
v3	inf	-	8	v1	8	v1	-3	v4	-3	v4
v4	inf	-	inf	-	2	v5	2	v5	2	v5
v5	inf	-	-4	v1	-4	v1	-4	v1	-4	v1

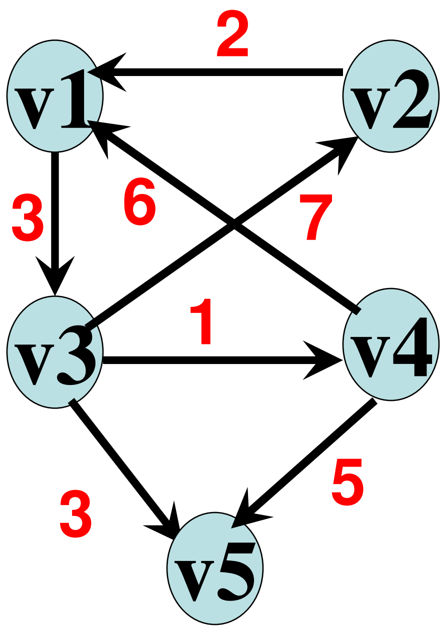
Bellman-Ford Algorithm: Example 3

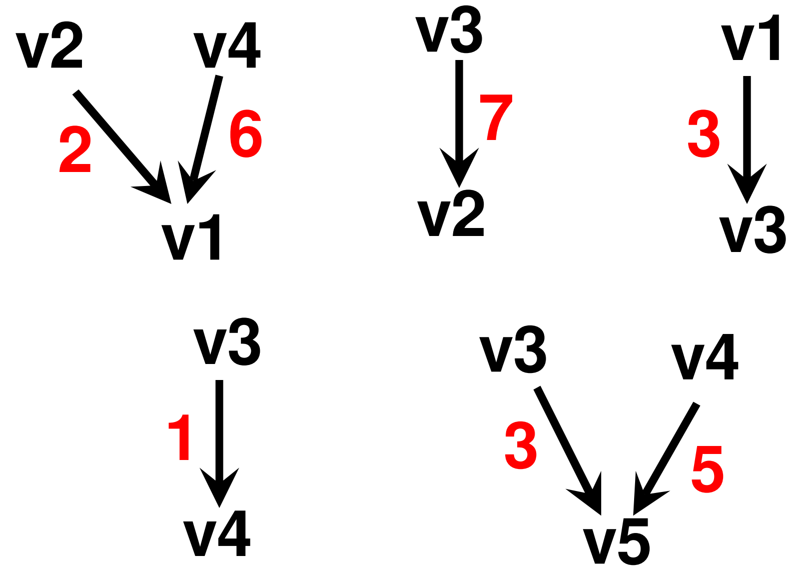




	Initial	
	Est.	Pred
v1	0	-
v2	inf	-
v3	inf	-
v4	inf	-
v5	inf	-

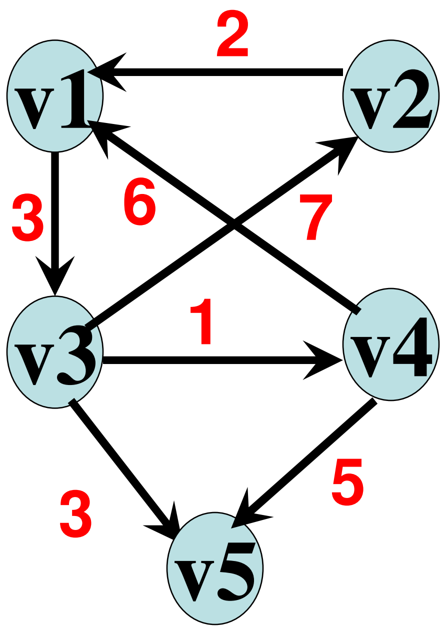
Bellman-Ford Algorithm: Example 3

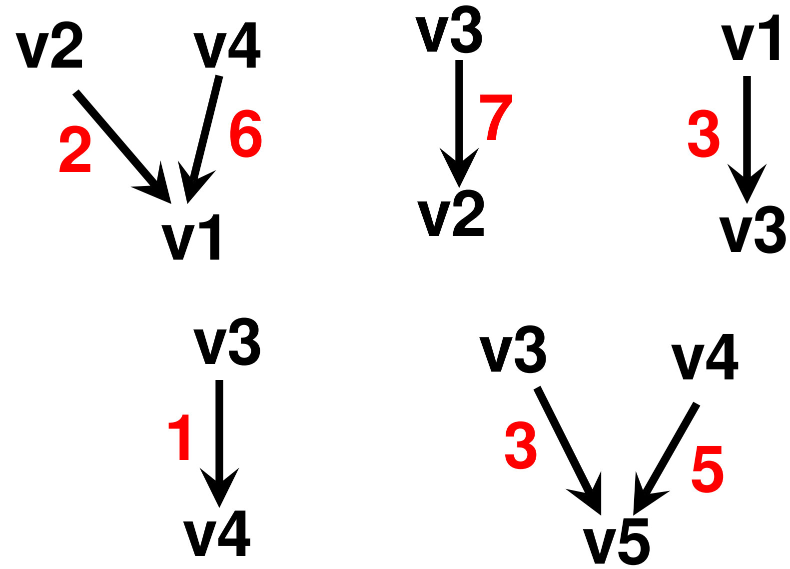




	Initial		Iteration 1	
	Est.	Pred	Est.	Pred
v1	0	-	0	-
v2	inf	-	inf	-
v3	inf	-	3	v1
v4	inf	-	inf	-
v5	inf	-	inf	-

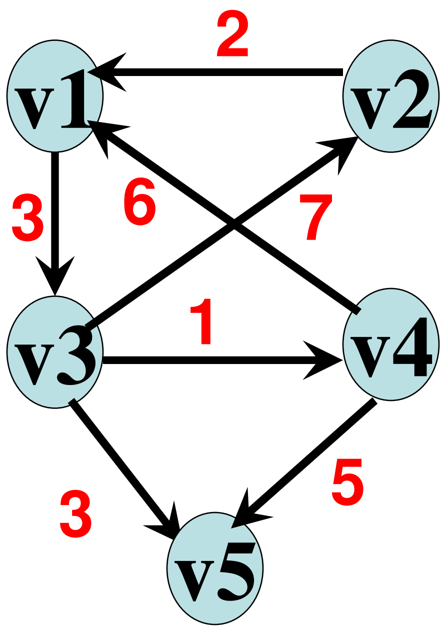
Bellman-Ford Algorithm: Example 3



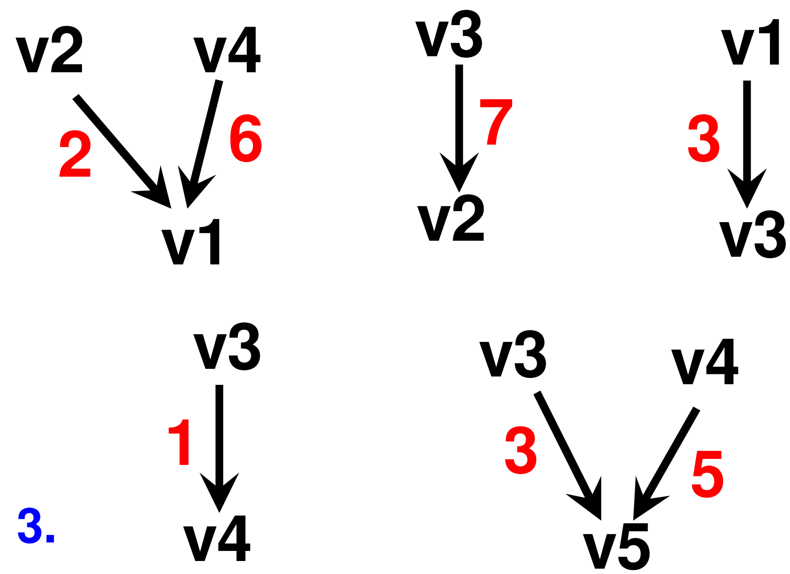


		Iteration 1		Iteration 2		
		Est.	Pred	Est.	Pred	
v1		0	-	0	-	
v2		inf	-	10	v3	
v3		3	v1	3	v1	
v4		inf	-	4	v3	
v5		inf	-	6	v3	

Bellman-Ford Algorithm: Example 3

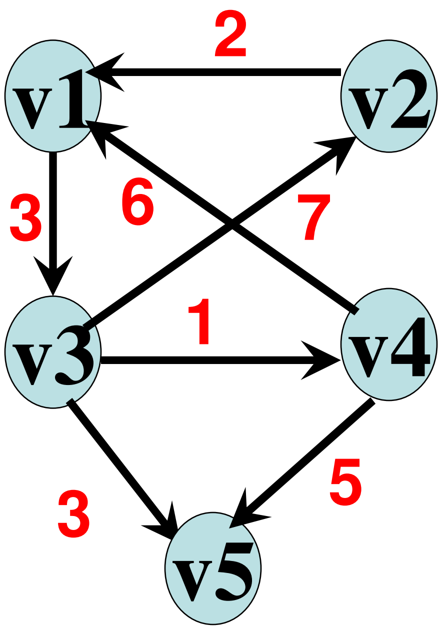


Note that the Estimates did Not change in Iterations 2 and 3. We can STOP!



		Iteration 2		Iteration 3		
		Est.	Pred	Est.	Pred	
v1		0	-	0	-	
v2		10	v3	10	v3	
v3		3	v1	3	v1	
v4		4	v3	4	v3	
v5		6	v3	v6	v3	

Bellman-Ford Algorithm: Example 3



V1 → V3
V1 → V3 → V2
V1 → V3 → V4
V1 → V3 → V5

Optimization Possible!!

	Initial		Iteration 1		Iteration 2		Iteration 3		Iteration 4	
	Est.	Pred	Est.	Pred	Est.	Pred	Est.	Pred	Est.	Pred
v1	0	-	0	-	0	-	0	-		
v2	inf	-	inf	-	10	v3	10	v3		
v3	inf	-	3	v1	3	v1	3	v1		
v4	inf	-	inf	-	4	v3	4	v3		
v5	inf	-	inf	-	6	v3	v6	v3		

NOT NEEDED

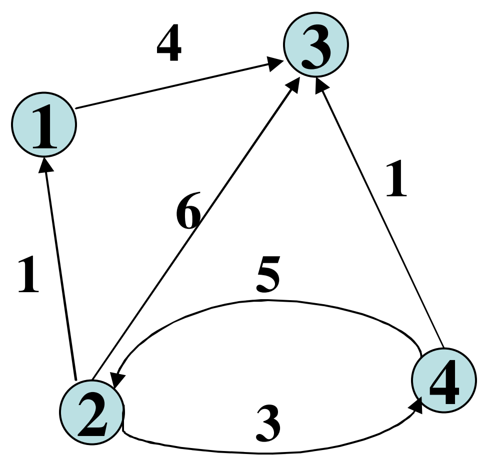
All Pairs Shortest Paths Problem

Dynamic Programming Algorithm for All Pairs Shortest Paths

Problem: **In a weighted (di)graph, find shortest paths between every pair of vertices**

idea: construct solution through series of matrices $D^{(0)}$, …, $D^{(n)}$ using increasing subsets of the vertices allowed as intermediate

Example:



The algorithm we are going to see was developed by two people Floyd and Warshall. We will shortly refer to the algorithm as the FW algorithm

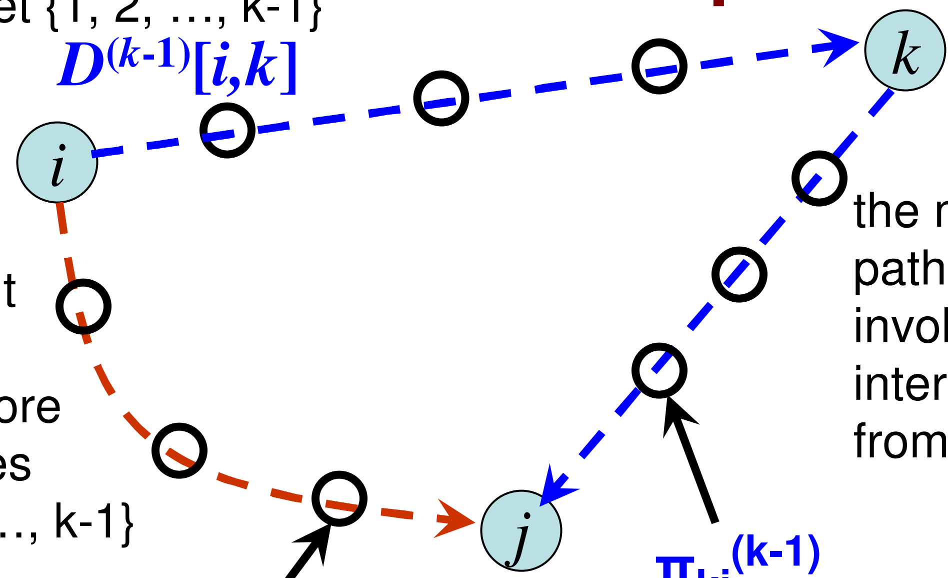
FW Algorithm: Operating Principle

- **Operating Principle:** The vertices are numbered from 1 to n. There are ‘n’ iterations. In the kth iteration, the candidate set of vertices available to choose from as intermediate vertices are {1, 2, 3, …, k}.
- **Initialization:** No vertex is a candidate intermediate vertex. There is a path between two vertices only if there is a direct edge between them (i.e., i → j); otherwise, not.
- **Iteration 1:** Candidate intermediate vertex {1}. Hence, the candidate paths to choose from are (depending on the graph, the following two may be true):
  i → j (or) i → 1 → j
- **Iteration 2:** Candidate intermediate vertices {1, 2}. Hence, the candidate paths to choose from are (depending on the graph; the following in an exhaustive list for a complete graph in case of a brute force approach):
  i –> j (or) i → 1 → j (or) i → 2 → j (or) i → 1 → 2 → j (or) i → 2 → 1 → j
- **Iteration 3:** Candidate intermediate vertices {1, 2, 3}. Hence, the candidate paths to choose from are (depending on the graph; the following in an exhaustive list for a complete graph in case of a brute force approach):
  i –> j (or) i → 1 → j (or) i → 2 → j (or) i → 3 → j (or) i → 1 → 2 → j (or) i → 2 → 1 → j (or) i → 1 → 3 → j (or) i → 3 → 1 → j (or) i → 2 → 3 → j (or) i → 3 → 2 → j (or) i → 1 → 2 → 3 → j (or) i → 3 → 2 → 1 → j (or) i → 1 → 3 → 2 → j (or) i → 3 → 1 → 2 → j (or) i → 2 → 3 → 1 → j (or) i → 2 → 1 → 3 → j

FW Algorithm: Operating Principle

the minimum weight path from i to k involving zero or more intermediate vertices from the set {1, 2, …, k-1}

$D^{(k-1)}[i,k]$



the minimum weight path from k to j involving zero or more intermediate vertices from the set {1, 2, …, k-1}

$D^{(k-1)}[k,j]$

the minimum weight path from i to j involving zero or more intermediate vertices from the set {1, 2, …, k-1}

$D^{(k-1)}[i,j]$

$\pi_{ij}^{(k-1)}$

$\pi_{kj}^{(k-1)}$

$$D^{(k)}[i,j] = \min \{ D^{(k-1)}[i,j],\ D^{(k-1)}[i,k] + D^{(k-1)}[k,j] \}$$

$$D^{(0)}[i,j] = w_{ij} \quad \text{if } i \rightarrow j \in E$$

$$D^{(0)}[i,j] = \infty \quad \text{if } i \rightarrow j \notin E$$

$$\pi_{ij}^{(0)} = i \quad \text{if } i \rightarrow j \in E$$

$$\pi_{ij}^{(0)} = N/A \quad \text{if } i \rightarrow j \notin E$$

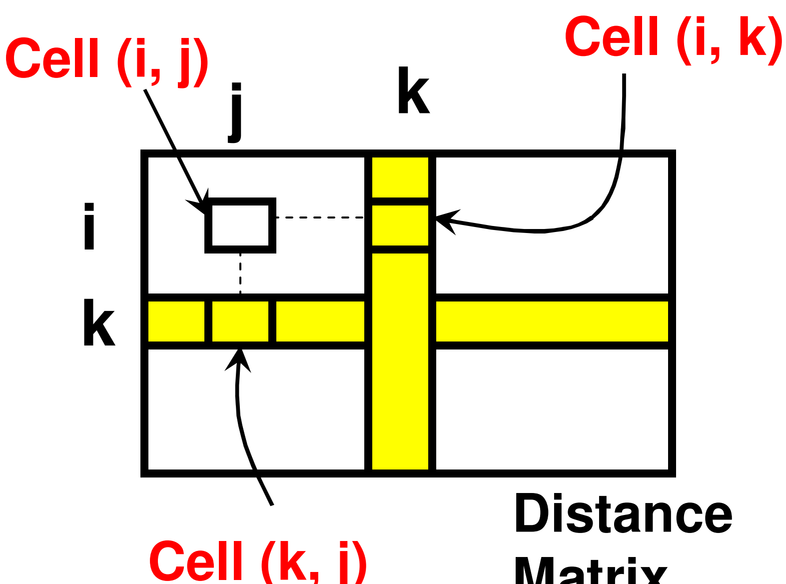
$$\pi_{ij}^{(k)} = \begin{cases} \pi_{ij}^{(k-1)} & \text{if } D_{ij}^{(k-1)} \le D_{ik}^{(k-1)} + D_{kj}^{(k-1)} \\ \pi_{kj}^{(k-1)} & \text{if } D_{ij}^{(k-1)} > D_{ik}^{(k-1)} + D_{kj}^{(k-1)} \end{cases}$$

FW Algorithm: Working Principle

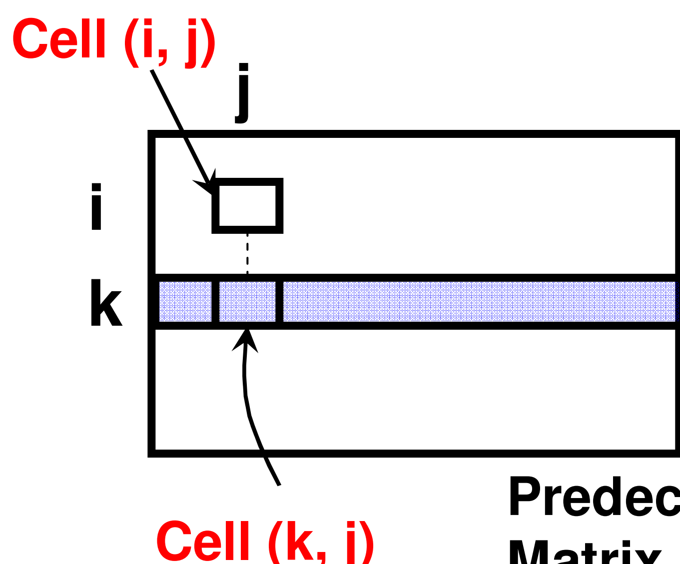
- **In iteration k, we highlight the row and column corresponding to vertex k, and check whether the values for each of the other cells could be reduced from what they were prior to that iteration. We do not change the values for the cells in the row and column corresponding to vertex k.**

We update a cell (i, j) if the value in the cell is greater than the sum of the Values of the cells (i, k) and (k, j)

If we update cell (i, j), we also update the predecessor for (i, j) to be the value corresponding to the predecessor for (k, j) in row k.

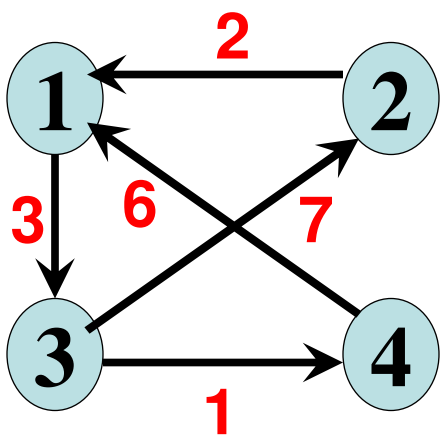


Distance Matrix



Predecessor Matrix

FW Algorithm: Example 1 (1)



Iteration 1

$D^{(0)}$

	v1	v2	v3	v4
v1	0	∞	3	∞
v2	2	0	∞	∞
v3	∞	7	0	1
v4	6	∞	∞	0

$\Pi^{(0)}$

	v1	v2	v3	v4
v1	N/A	N/A	v1	N/A
v2	v2	N/A	N/A	N/A
v3	N/A	v3	N/A	v3
v4	v4	N/A	N/A	N/A

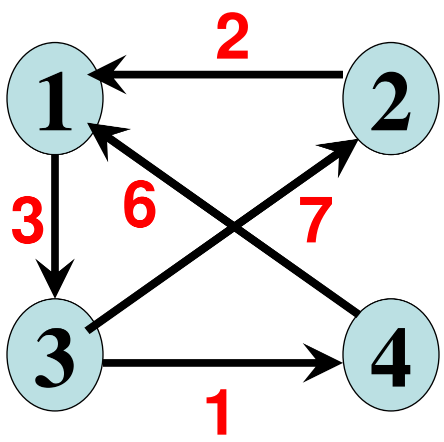
$D^{(1)}$

	v1	v2	v3	v4
v1	0	∞	3	∞
v2	2			
v3	∞			
v4	6			

$\Pi^{(1)}$

	v1	v2	v3	v4
v1	N/A	N/A	v1	N/A
v2				
v3				
v4				

FW Algorithm: Example 1 (1)



Iteration 1

$D^{(0)}$

	v1	v2	v3	v4
v1	0	∞	3	∞
v2	2	0	∞	∞
v3	∞	7	0	1
v4	6	∞	∞	0

$\Pi^{(0)}$

	v1	v2	v3	v4
v1	N/A	N/A	v1	N/A
v2	v2	N/A	N/A	N/A
v3	N/A	v3	N/A	v3
v4	v4	N/A	N/A	N/A

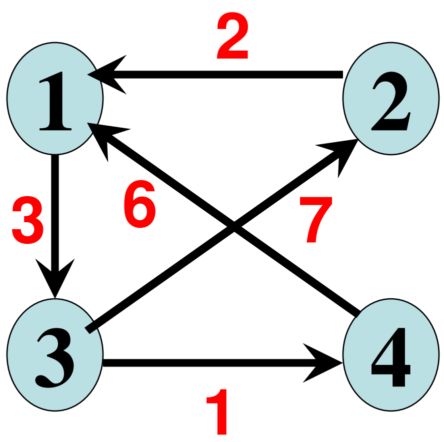
$D^{(1)}$

	v1	v2	v3	v4
v1	0	∞	3	∞
v2	2	0	5	∞
v3	∞	7	0	1
v4	6	∞	9	0

$\Pi^{(1)}$

	v1	v2	v3	v4
v1	N/A	N/A	v1	N/A
v2	v2	N/A	v1	N/A
v3	N/A	v3	N/A	v3
v4	v4	N/A	v1	N/A

FW Algorithm: Example 1 (2)



Iteration 2

$D^{(1)}$

	v1	v2	v3	v4
v1	0	∞	3	∞
v2	2	0	5	∞
v3	∞	7	0	1
v4	6	∞	9	0

$\Pi^{(1)}$

	v1	v2	v3	v4
v1	N/A	N/A	v1	N/A
v2	v2	N/A	v1	N/A
v3	N/A	v3	N/A	v3
v4	v4	N/A	v1	N/A

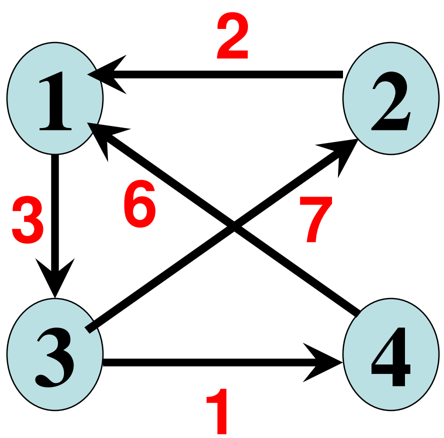
$D^{(2)}$

	v1	v2	v3	v4
v1		∞		
v2	2	0	5	∞
v3		7		
v4		∞		

$\Pi^{(2)}$

	v1	v2	v3	v4
v1				
v2	v2	N/A	v1	N/A
v3				
v4				

FW Algorithm: Example 1 (2)



Iteration 2

$D^{(1)}$

	v1	v2	v3	v4
v1	0	∞	3	∞
v2	2	0	5	∞
v3	∞	7	0	1
v4	6	∞	9	0

$\Pi^{(1)}$

	v1	v2	v3	v4
v1	N/A	N/A	v1	N/A
v2	v2	N/A	v1	N/A
v3	N/A	v3	N/A	v3
v4	v4	N/A	v1	N/A

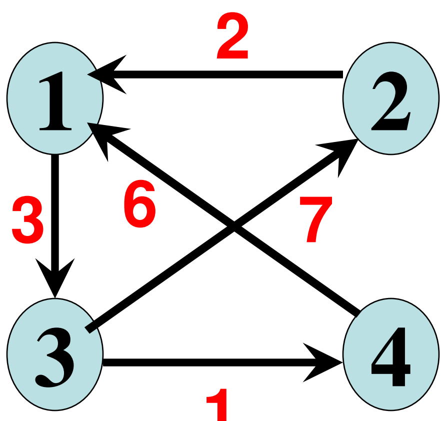
$D^{(2)}$

	v1	v2	v3	v4
v1	0	∞	3	∞
v2	2	0	5	∞
v3	9	7	0	1
v4	6	∞	9	0

$\Pi^{(2)}$

	v1	v2	v3	v4
v1	N/A	N/A	v1	N/A
v2	v2	N/A	v1	N/A
v3	v2	v3	N/A	v3
v4	v4	N/A	v1	N/A

FW Algorithm: Example 1 (3)



Iteration 3

$D^{(2)}$

	v1	v2	v3	v4
v1	0	∞	3	∞
v2	2	0	5	∞
v3	9	7	0	1
v4	6	∞	9	0

$\Pi^{(2)}$

	v1	v2	v3	v4
v1	N/A	N/A	v1	N/A
v2	v2	N/A	v1	N/A
v3	v2	v3	N/A	v3
v4	v4	N/A	v1	N/A

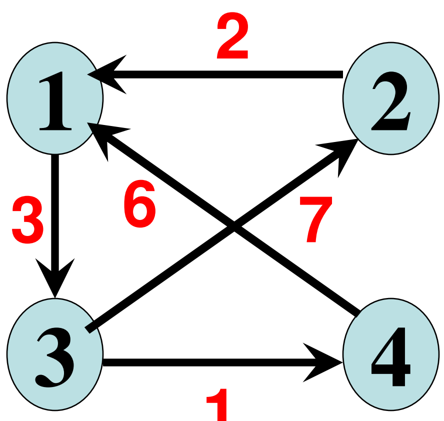
$D^{(3)}$

	v1	v2	v3	v4
v1			3	
v2			5	
v3	9	7	0	1
v4			9	

$\Pi^{(3)}$

	v1	v2	v3	v4
v1				
v2				
v3	v2	v3	N/A	v3
v4				

FW Algorithm: Example 1 (3)



Iteration 3

$D^{(2)}$

	v1	v2	v3	v4
v1	0	∞	3	∞
v2	2	0	5	∞
v3	9	7	0	1
v4	6	∞	9	0

$\Pi^{(2)}$

	v1	v2	v3	v4
v1	N/A	N/A	v1	N/A
v2	v2	N/A	v1	N/A
v3	v2	v3	N/A	v3
v4	v4	N/A	v1	N/A

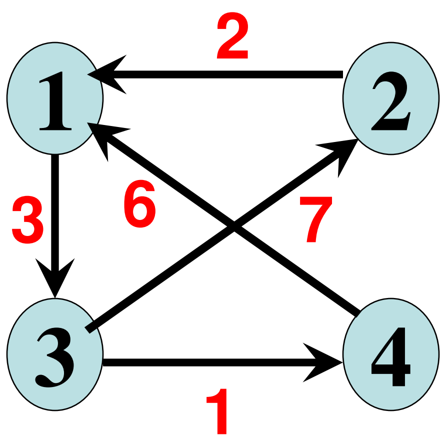
$D^{(3)}$

	v1	v2	v3	v4
v1	0	10	3	4
v2	2	0	5	6
v3	9	7	0	1
v4	6	16	9	0

$\Pi^{(3)}$

	v1	v2	v3	v4
v1	N/A	v3	v1	v3
v2	v2	N/A	v1	v3
v3	v2	v3	N/A	v3
v4	v4	v3	v1	N/A

FW Algorithm: Example 1 (4)



Iteration 4

$D^{(3)}$

	v1	v2	v3	v4
v1	0	10	3	4
v2	2	0	5	6
v3	9	7	0	1
v4	6	16	9	0

$\Pi^{(3)}$

	v1	v2	v3	v4
v1	N/A	v3	v1	v3
v2	v2	N/A	v1	v3
v3	v2	v3	N/A	v3
v4	v4	v3	v1	N/A

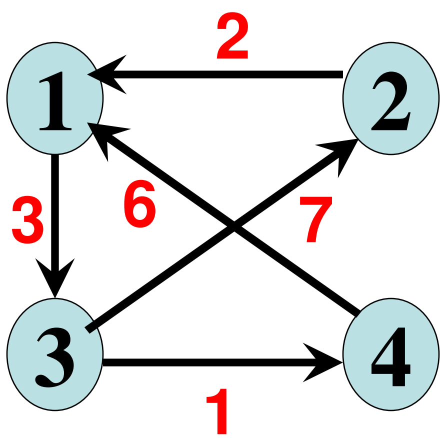
$D^{(4)}$

	v1	v2	v3	v4
v1				4
v2				6
v3				1
v4	6	16	9	0

$\Pi^{(4)}$

	v1	v2	v3	v4
v1				
v2				
v3				
v4	v4	v3	v1	N/A

FW Algorithm: Example 1 (4)



Iteration 4

$D^{(3)}$

	v1	v2	v3	v4
v1	0	10	3	4
v2	2	0	5	6
v3	9	7	0	1
v4	6	16	9	0

$\Pi^{(3)}$

	v1	v2	v3	v4
v1	N/A	v3	v1	v3
v2	v2	N/A	v1	v3
v3	v2	v3	N/A	v3
v4	v4	v3	v1	N/A

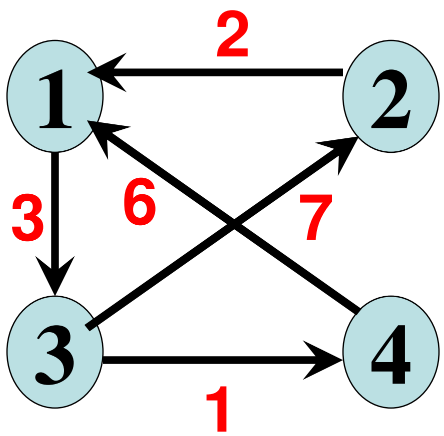
$D^{(4)}$

	v1	v2	v3	v4
v1	0	10	3	4
v2	2	0	5	6
v3	7	7	0	1
v4	6	16	9	0

$\Pi^{(4)}$

	v1	v2	v3	v4
v1	N/A	v3	v1	v3
v2	v2	N/A	v1	v3
v3	v4	v3	N/A	v3
v4	v4	v3	v1	N/A

FW Algorithm: Example 1 (5)



$D^{(4)}$

	v1	v2	v3	v4
v1	0	10	3	4
v2	2	0	5	6
v3	7	7	0	1
v4	6	16	9	0

$\Pi^{(4)}$

	v1	v2	v3	v4
v1	N/A	v3	v1	v3
v2	v2	N/A	v1	v3
v3	v4	v3	N/A	v3
v4	v4	v3	v1	N/A

Path from v2 to v4
π (v2 … v4)
= π (v2 … v3) → v3 → v4
= π (v2 … v1) → v1 → v3 → v4
= v2 → v1 → v3 → v4

Path from v4 to v2
π (v4 … v2)
= π (v4 … v3) → v3 → v2
= π (v4 … v1) → v1 → v3 → v2
= v4 → v1 → v3 → v2

FW Algorithm (pseudocode and analysis)

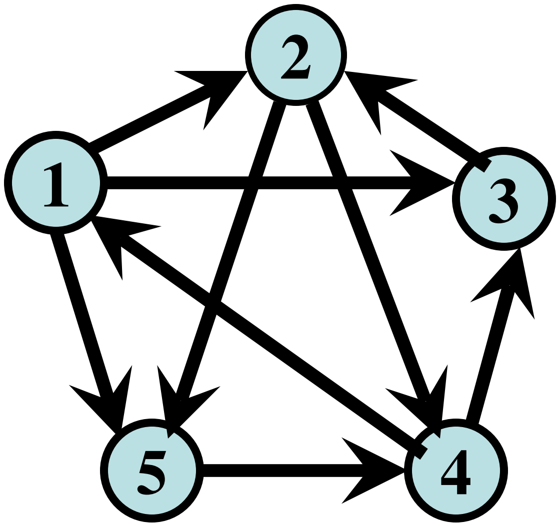
ALGORITHM *Floyd*($W[1..n, 1..n]$)

```
//Implements Floyd's algorithm for the all-pairs shortest-paths problem
//Input: The weight matrix W of a graph with no negative-length cycle
//Output: The distance matrix of the shortest paths' lengths
D ← W //is not necessary if W can be overwritten
for k ← 1 to n do
    for i ← 1 to n do
        for j ← 1 to n do
            D[i, j] ← min{D[i, j], D[i, k] + D[k, j]}
return D
```

Time efficiency: $\Theta(n^3)$

Space efficiency: $\Theta(n^2)$

FW Algorithm: Example 2(1)



	Weight Matrix				
	v1	v2	v3	v4	v5
v1	0	3	8	∞	-4
v2	∞	0	∞	1	7
v3	∞	4	0	∞	∞
v4	2	∞	-5	0	∞
v5	∞	∞	∞	6	0

$D^{(0)}$

	v1	v2	v3	v4	v5
v1	0	3	8	∞	-4
v2	∞	0	∞	1	7
v3	∞	4	0	∞	∞
v4	2	∞	-5	0	∞
v5	∞	∞	∞	6	0

$\Pi^{(0)}$

	v1	v2	v3	v4	v5
v1	N/A	v1	v1	N/A	v1
v2	N/A	N/A	N/A	v2	v2
v3	N/A	v3	N/A	N/A	N/A
v4	v4	N/A	v4	N/A	N/A
v5	N/A	N/A	N/A	v5	N/A

$D^{(1)}$

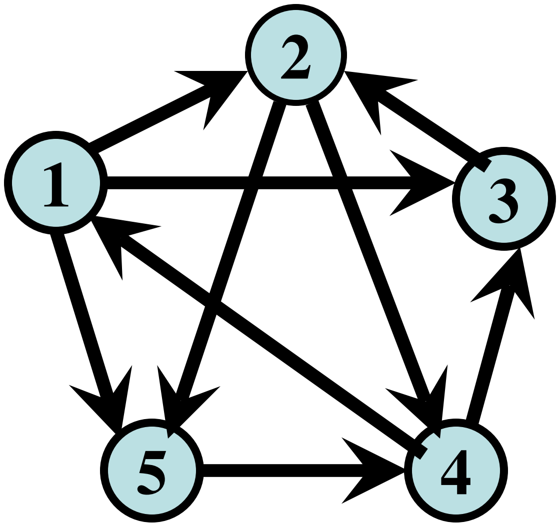
	v1	v2	v3	v4	v5
v1	0	3	8	∞	-4
v2	∞				
v3	∞				
v4	2				
v5	∞				

$\Pi^{(1)}$

	v1	v2	v3	v4	v5
v1	N/A	v1	v1	N/A	v1
v2					
v3					
v4					
v5					

Iteration 1

FW Algorithm: Example 2(1)



$D^{(0)}$

	v1	v2	v3	v4	v5
v1	0	3	8	∞	-4
v2	∞	0	∞	1	7
v3	∞	4	0	∞	∞
v4	2	∞	-5	0	∞
v5	∞	∞	∞	6	0

$\Pi^{(0)}$

	v1	v2	v3	v4	v5
v1	N/A	v1	v1	N/A	v1
v2	N/A	N/A	N/A	v2	v2
v3	N/A	v3	N/A	N/A	N/A
v4	v4	N/A	v4	N/A	N/A
v5	N/A	N/A	N/A	v5	N/A

	Weight Matrix				
	v1	v2	v3	v4	v5
v1	0	3	8	∞	-4
v2	∞	0	∞	1	7
v3	∞	4	0	∞	∞
v4	2	∞	-5	0	∞
v5	∞	∞	∞	6	0

$D^{(1)}$

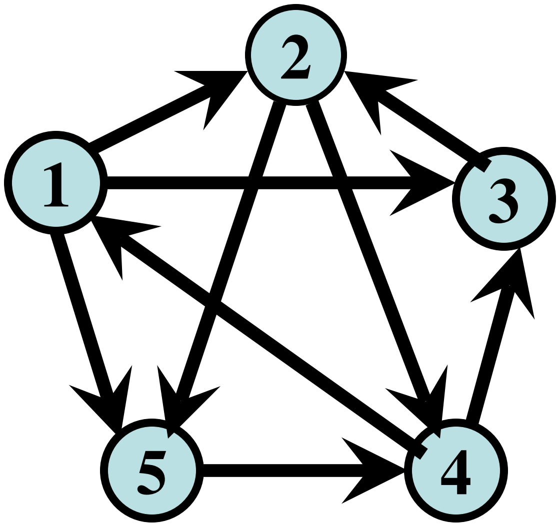
	v1	v2	v3	v4	v5
v1	0	3	8	∞	-4
v2	∞	0	∞	1	7
v3	∞	4	0	∞	∞
v4	2	5	-5	0	-2
v5	∞	∞	∞	6	0

$\Pi^{(1)}$

	v1	v2	v3	v4	v5
v1	N/A	v1	v1	N/A	v1
v2	N/A	N/A	N/A	v2	v2
v3	N/A	v3	N/A	N/A	N/A
v4	v4	v1	v4	N/A	v1
v5	N/A	N/A	N/A	v5	N/A

Iteration 1

FW Algorithm: Example 2(2)



	Weight Matrix				
	v1	v2	v3	v4	v5
v1	0	3	8	∞	-4
v2	∞	0	∞	1	7
v3	∞	4	0	∞	∞
v4	2	∞	-5	0	∞
v5	∞	∞	∞	6	0

$D^{(1)}$

	v1	v2	v3	v4	v5
v1	0	3	8	∞	-4
v2	∞	0	∞	1	7
v3	∞	4	0	∞	∞
v4	2	5	-5	0	-2
v5	∞	∞	∞	6	0

$\Pi^{(1)}$

	v1	v2	v3	v4	v5
v1	N/A	v1	v1	N/A	v1
v2	N/A	N/A	N/A	v2	v2
v3	N/A	v3	N/A	N/A	N/A
v4	v4	v1	v4	N/A	v1
v5	N/A	N/A	N/A	v5	N/A

$D^{(2)}$

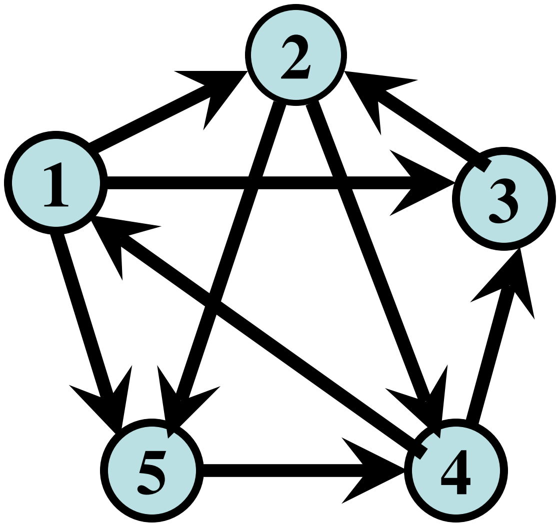
	v1	v2	v3	v4	v5
v1		3			
v2	∞	0	∞	1	7
v3		4			
v4		5			
v5		∞			

$\Pi^{(2)}$

	v1	v2	v3	v4	v5
v1					
v2	N/A	N/A	N/A	v2	v2
v3					
v4					
v5					

Iteration 2

FW Algorithm: Example 2(2)



	Weight Matrix				
	v1	v2	v3	v4	v5
v1	0	3	8	∞	-4
v2	∞	0	∞	1	7
v3	∞	4	0	∞	∞
v4	2	∞	-5	0	∞
v5	∞	∞	∞	6	0

$D^{(1)}$

	v1	v2	v3	v4	v5
v1	0	3	8	∞	-4
v2	∞	0	∞	1	7
v3	∞	4	0	∞	∞
v4	2	5	-5	0	-2
v5	∞	∞	∞	6	0

$\Pi^{(1)}$

	v1	v2	v3	v4	v5
v1	N/A	v1	v1	N/A	v1
v2	N/A	N/A	N/A	v2	v2
v3	N/A	v3	N/A	N/A	N/A
v4	v4	v1	v4	N/A	v1
v5	N/A	N/A	N/A	v5	N/A

$D^{(2)}$

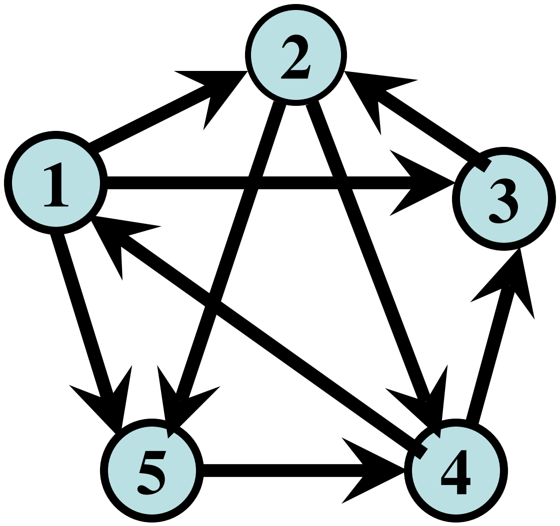
	v1	v2	v3	v4	v5
v1	0	3	8	4	-4
v2	∞	0	∞	1	7
v3	∞	4	0	5	11
v4	2	5	-5	0	-2
v5	∞	∞	∞	6	0

$\Pi^{(2)}$

	v1	v2	v3	v4	v5
v1	N/A	v1	v1	v2	v1
v2	N/A	N/A	N/A	v2	v2
v3	N/A	v3	N/A	v2	v2
v4	v4	v1	v4	N/A	v1
v5	N/A	N/A	N/A	v5	N/A

Iteration 2

FW Algorithm: Example 2(3)



	Weight Matrix				
	v1	v2	v3	v4	v5
v1	0	3	8	∞	-4
v2	∞	0	∞	1	7
v3	∞	4	0	∞	∞
v4	2	∞	-5	0	∞
v5	∞	∞	∞	6	0

$D^{(2)}$

	v1	v2	v3	v4	v5
v1	0	3	8	4	-4
v2	∞	0	∞	1	7
v3	∞	4	0	5	11
v4	2	5	-5	0	-2
v5	∞	∞	∞	6	0

$\Pi^{(2)}$

	v1	v2	v3	v4	v5
v1	N/A	v1	v1	v2	v1
v2	N/A	N/A	N/A	v2	v2
v3	N/A	v3	N/A	v2	v2
v4	v4	v1	v4	N/A	v1
v5	N/A	N/A	N/A	v5	N/A

$D^{(3)}$

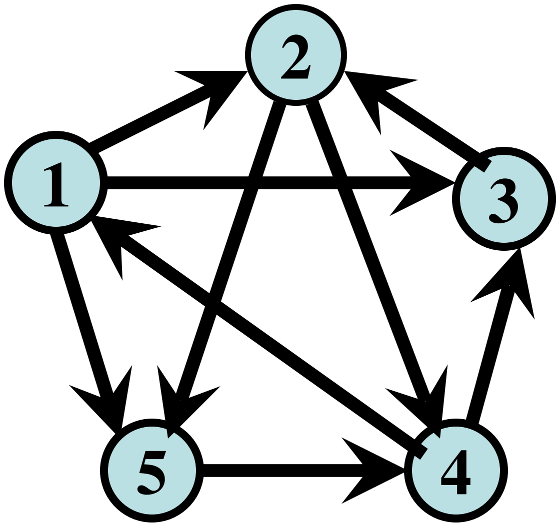
	v1	v2	v3	v4	v5
v1	0	3	8	4	-4
v2	∞	0	∞	1	7
v3	∞	4	0	5	11
v4	2	-1	-5	0	-2
v5	∞	∞	∞	6	0

$\Pi^{(3)}$

	v1	v2	v3	v4	v5
v1	N/A	v1	v1	v2	v1
v2	N/A	N/A	N/A	v2	v2
v3	N/A	v3	N/A	v2	v2
v4	v4	v3	v4	N/A	v1
v5	N/A	N/A	N/A	v5	N/A

Iteration 3

FW Algorithm: Example 2(4)



	Weight Matrix				
	v1	v2	v3	v4	v5
v1	0	3	8	∞	-4
v2	∞	0	∞	1	7
v3	∞	4	0	∞	∞
v4	2	∞	-5	0	∞
v5	∞	∞	∞	6	0

$D^{(3)}$

	v1	v2	v3	v4	v5
v1	0	3	8	4	-4
v2	∞	0	∞	1	7
v3	∞	4	0	5	11
v4	2	-1	-5	0	-2
v5	∞	∞	∞	6	0

$\Pi^{(3)}$

	v1	v2	v3	v4	v5
v1	N/A	v1	v1	v2	v1
v2	N/A	N/A	N/A	v2	v2
v3	N/A	v3	N/A	v2	v2
v4	v4	v3	v4	N/A	v1
v5	N/A	N/A	N/A	v5	N/A

$D^{(4)}$

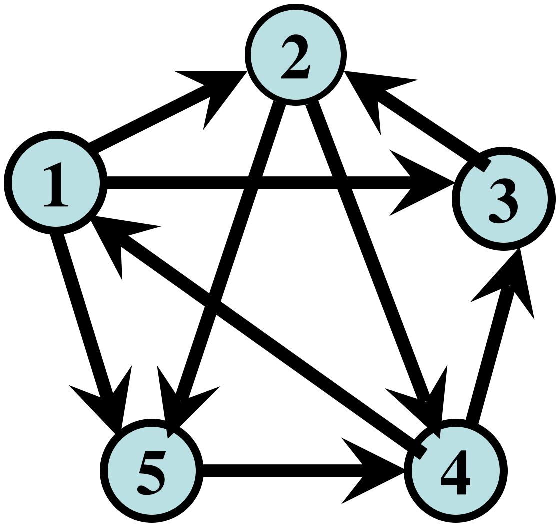
	v1	v2	v3	v4	v5
v1	0	3	-1	4	-4
v2	3	0	-4	1	-1
v3	7	4	0	5	3
v4	2	-1	-5	0	-2
v5	8	5	1	6	0

$\Pi^{(4)}$

	v1	v2	v3	v4	v5
v1	N/A	v1	v4	v2	v1
v2	v4	N/A	v4	v2	v1
v3	v4	v3	N/A	v2	v1
v4	v4	v3	v4	N/A	v1
v5	v4	v3	v4	v5	N/A

Iteration 4

FW Algorithm: Example 2(5)



	Weight Matrix				
	v1	v2	v3	v4	v5
v1	0	3	8	∞	-4
v2	∞	0	∞	1	7
v3	∞	4	0	∞	∞
v4	2	∞	-5	0	∞
v5	∞	∞	∞	6	0

$D^{(4)}$

	v1	v2	v3	v4	v5
v1	0	3	-1	4	-4
v2	3	0	-4	1	-1
v3	7	4	0	5	3
v4	2	-1	-5	0	-2
v5	8	5	1	6	0

$\Pi^{(4)}$

	v1	v2	v3	v4	v5
v1	N/A	v1	v4	v2	v1
v2	v4	N/A	v4	v2	v1
v3	v4	v3	N/A	v2	v1
v4	v4	v3	v4	N/A	v1
v5	v4	v3	v4	v5	N/A

$D^{(5)}$

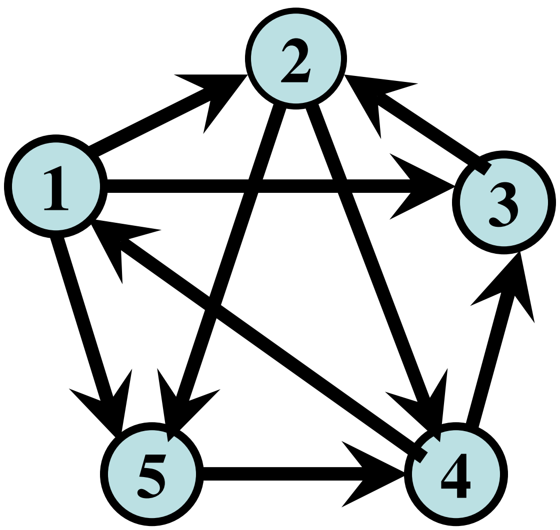
	v1	v2	v3	v4	v5
v1	0	1	-3	2	-4
v2	3	0	-4	1	-1
v3	7	4	0	5	3
v4	2	-1	-5	0	-2
v5	8	5	1	6	0

$\Pi^{(5)}$

	v1	v2	v3	v4	v5
v1	N/A	v3	v4	v5	v1
v2	v4	N/A	v4	v2	v1
v3	v4	v3	N/A	v2	v1
v4	v4	v3	v4	N/A	v1
v5	v4	v3	v4	v5	N/A

Iteration 5

FW Algorithm: Example 2(6)



$D^{(5)}$

	v1	v2	v3	v4	v5
v1	0	1	-3	2	-4
v2	3	0	-4	1	-1
v3	7	4	0	5	3
v4	2	-1	-5	0	-2
v5	8	5	1	6	0

$\Pi^{(5)}$

	v1	v2	v3	v4	v5
v1	N/A	v3	v4	v5	v1
v2	v4	N/A	v4	v2	v1
v3	v4	v3	N/A	v2	v1
v4	v4	v3	v4	N/A	v1
v5	v4	v3	v4	v5	N/A

Path from v3 to v1
π (v3 … v1)
= π (v3 … v4) → v4 → v1
= π (v3 … v2) → v2 → v4 → v1
= v3 → v2 → v4 → v1

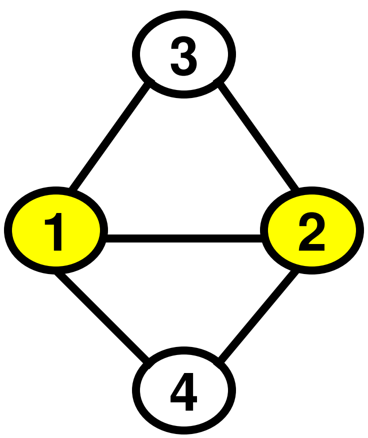
Path from v1 to v3
π (v1 … v3)
= π (v1 … v4) → v4 → v3
= π (v1 … v5) → v5 → v4 → v3
= v1 → v5 → v4 → v3

Comparison of the Shortest Path Algorithms

	Dijkstra	**Bellman-Ford**	**Floyd-Warshall**
Type	Single source shortest path	Single source shortest path	All pairs shortest path
Typical Graphs	Undirected	Directed	Undirected and Directed
Edge Weights	Positive only	Positive and/ or Negative	Positive and/ or Negative
Time Complexity	$\Theta(E*\log V)$	$\Theta(E*V)$	$\Theta(V^3)$

Number of Walks in a Graph

- An u-v walk between two vertices u and v is a sequence of zero or more intermediate vertices (that could be even repeated).
- The length of a walk is one plus the number of intermediate vertices
  - Example: **2 – 3 – 1 – 4 – 1 is a walk of length 4**.
- A walk is a path if the intermediate vertices, if any, are not repeated.
  - Example: **2 – 3 – 1 is a walk as well as a path, but the walk 2 – 3 – 1 – 4 – 1 is not a path.**
- The number of walks of length *k* between any two vertices in a graph could be determined by finding A^k where A is the binary adjacency matrix of the graph.



Adjacency Matrix (A)

	1	2	3	4
1	0	1	1	1
2	1	0	1	1
3	1	1	0	0
4	1	1	0	0

X

Adjacency Matrix (A)

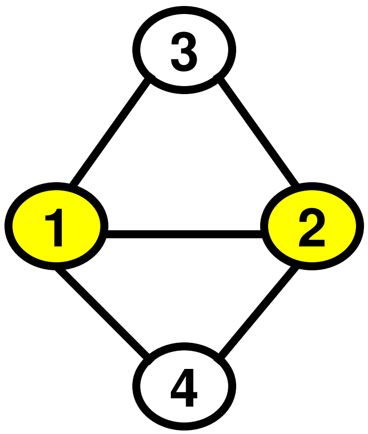
	1	2	3	4
1	0	1	1	1
2	1	0	1	1
3	1	1	0	0
4	1	1	0	0

=

 # Walks of Length 2 (A^2)

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

Number of Walks in a Graph



 # Walks of Length 2 (A^2)

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

X

Adjacency Matrix (A)

	1	2	3	4
1	0	1	1	1
2	1	0	1	1
3	1	1	0	0
4	1	1	0	0

=

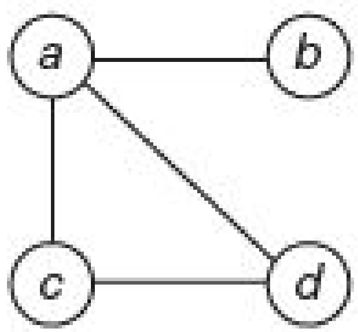
 # Walks of Length 3 (A^3)

	1	2	3	4
1	4	5	5	5
2	5	4	5	5
3	5	5	2	2
4	5	5	2	2

- Basic Rules for Matrix Multiplication
- To multiply two matrices A and B and get a product matrix P = A * B:
- (1) The number of columns in the first matrix A should be equal to the number of rows in the second matrix B
- (2) To get the value of a cell (i, j) in the product matrix P, do a pair-wise multiplication of the elements in row i of the first matrix with the elements in column j of the second matrix.

To find # Walks of Length 'n'

 # Walks of Length 4: Find A^4.



$$A^2 = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 3 & 0 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 1 & 1 & 2 & 1 \\ 1 & 1 & 1 & 2 \end{bmatrix} \end{array} \quad \mathbf{x} \quad A^2 = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 3 & 0 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 1 & 1 & 2 & 1 \\ 1 & 1 & 1 & 2 \end{bmatrix} \end{array}$$

A^4 =

	a	**b**	**c**	**d**
a	11	2	6	6
b	2	3	4	4
c	6	4	5	6
d	6	4	6	7

To find the number of walks length 4 between vertices b and c, just simply do a pair-wise multiplication and addition of the elements corresponding to the row for vertex 'b' in A^2 with the elements corresponding to the column for vertex 'c' in A^2.

Note: Rule for Matrix Multiplication
To find the value of an entry in cell (i, j) in the product matrix P = A * B,
Do a pair-wise multiplication and addition of the elements in row 'i' of the first matrix A and the elements in column 'j' of the second matrix B.