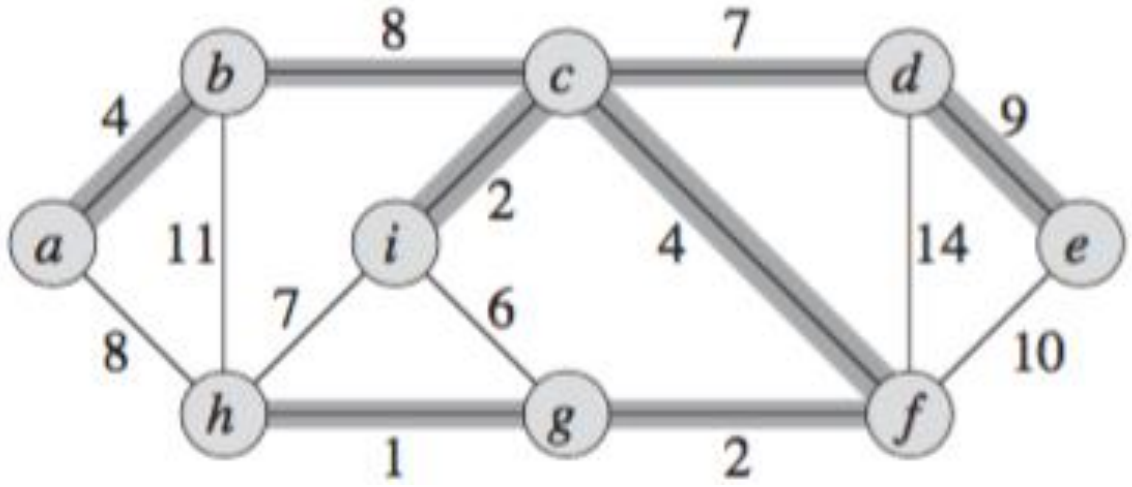
CSCE 3110
Data Structures & Algorithms

- Minimum Spanning Tree
- Single-Source Shortest Paths
- All-Pair Shortest Paths

Minimum Spanning Tree

- MST (G)
  - tree: acyclic subgraph of G
  - spanning: spans (connects) **all the vertices** of G
  - has the minimum total weight = sum of edge weight
  - is not unique

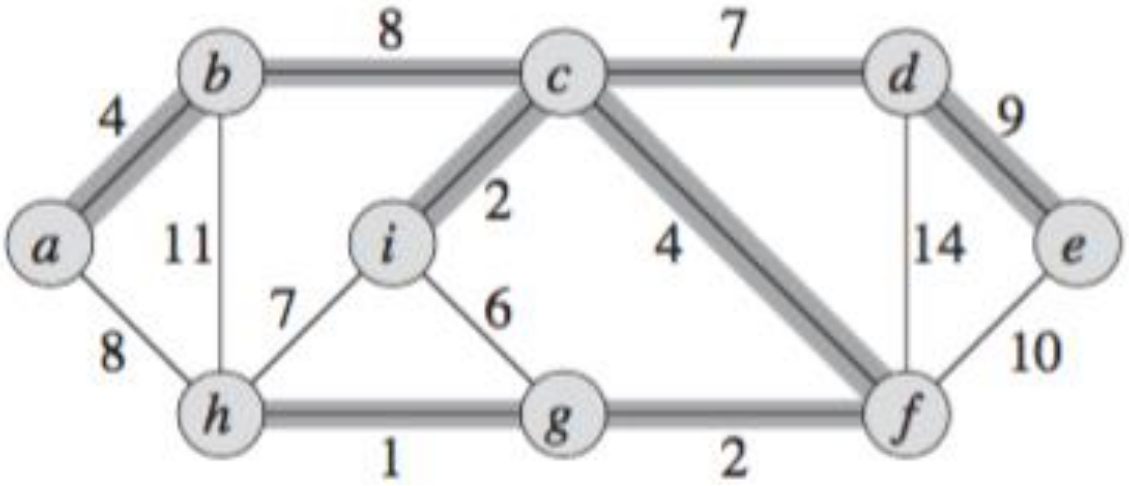


Construction of MST

- Key Idea:

  At each round, select an edge in a greedy manner and terminate when all vertices are spanned.

- Greedy Algorithms:
  - Prim's MST Algorithm
  - Kruskal's MST Algorithm



Prim's MST Algorithm

- V: set of vertices in graph G
- V_T: set of vertices in MST
- Process:

  Initially, let $V_T = \Phi$

  Select an arbitrary vertex, e.g. s, to start a tree

  While $V_T \neq V$

  Select an edge e(u, v) such that **u in V_T, v in V-V_T**, and w(e) is **minimum among all such edges**

  //choose an edge with connected vertex in a greedy manner

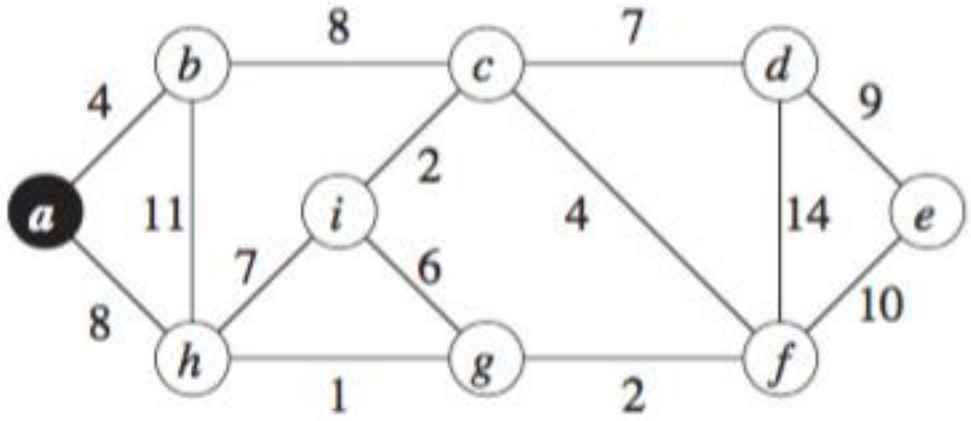
  Endwhile

Example

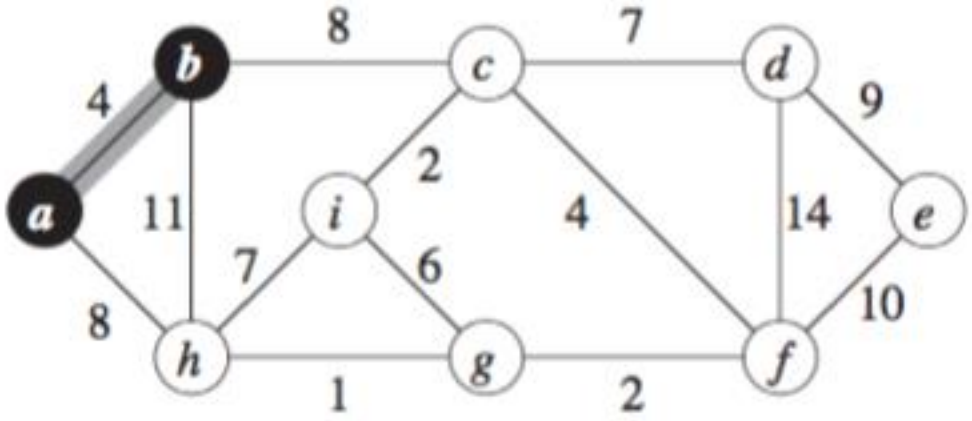
While $V_T \neq V$

Select an edge e(u, v) such that **u in V_T, v in V-V_T**, and w(e) is **minimum among all such edges**

Starting with node a



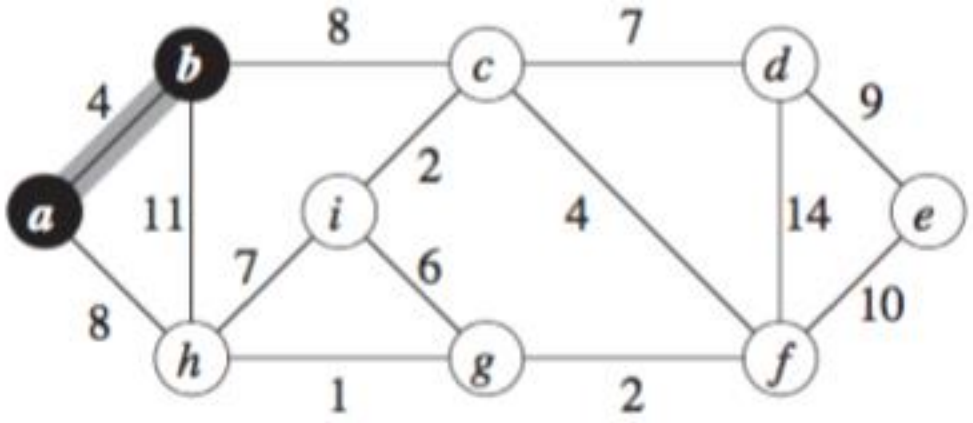
Choose edge e(a, b)



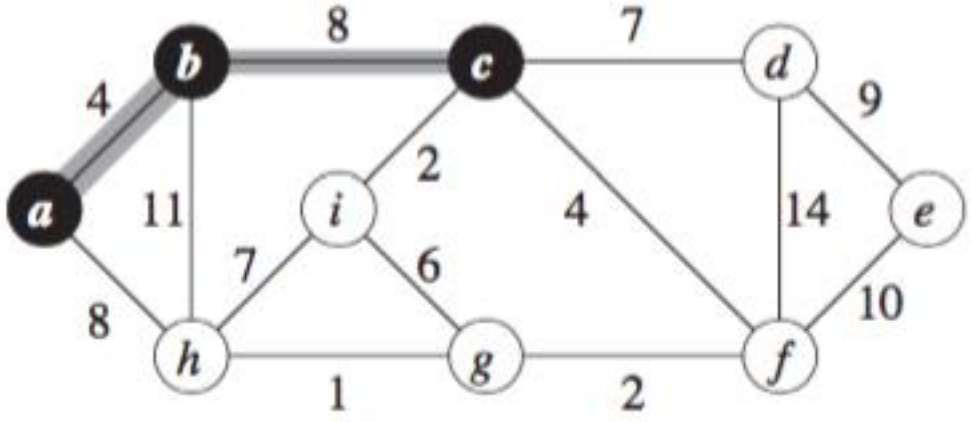
Example (cont.)

While $V_T \neq V$

Select an edge e(u, v) such that **u in V_T, v in V-V_T**, and w(e) is **minimum among all such edges**



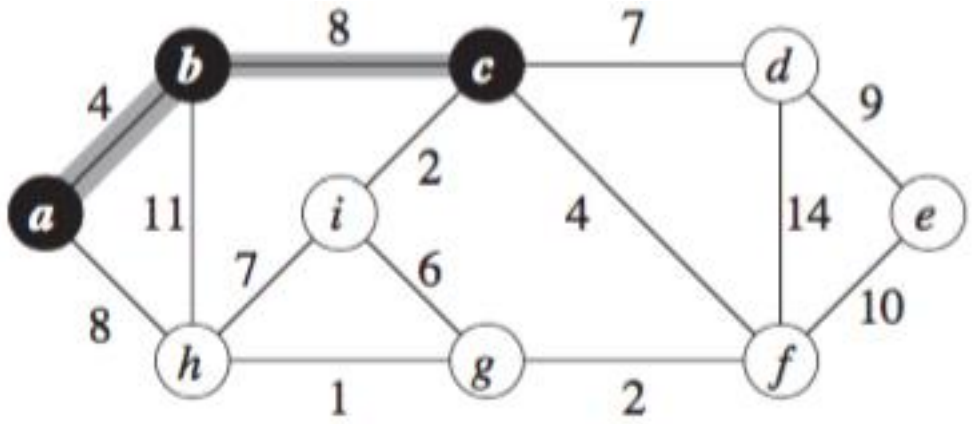
Choose edge e(b, c)



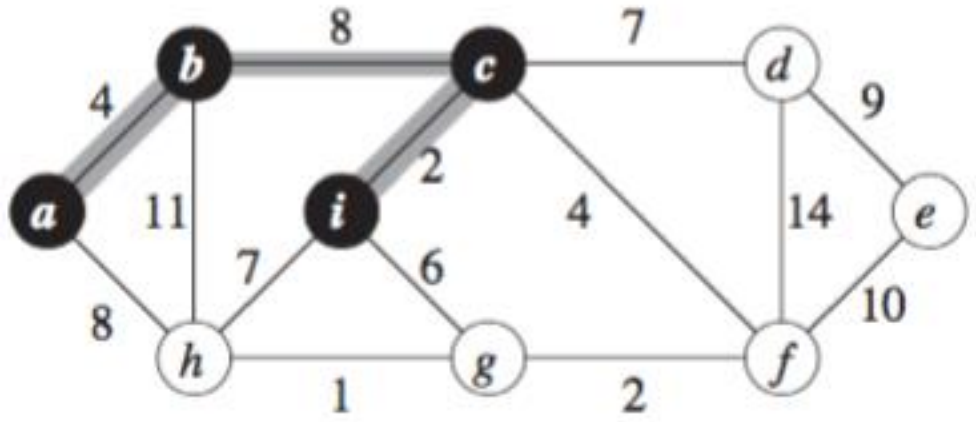
Example (cont.)

While $V_T \neq V$

Select an edge e(u, v) such that **u in V_T, v in V-V_T**, and w(e) is **minimum among all such edges**



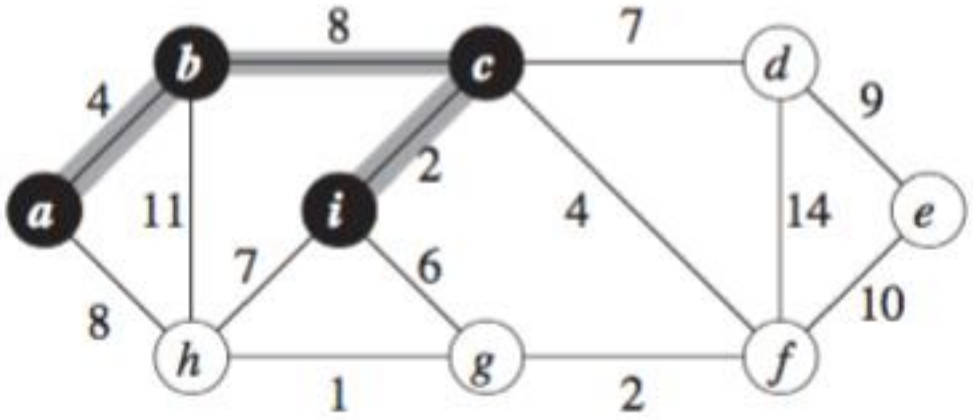
Choose edge e(c, i)



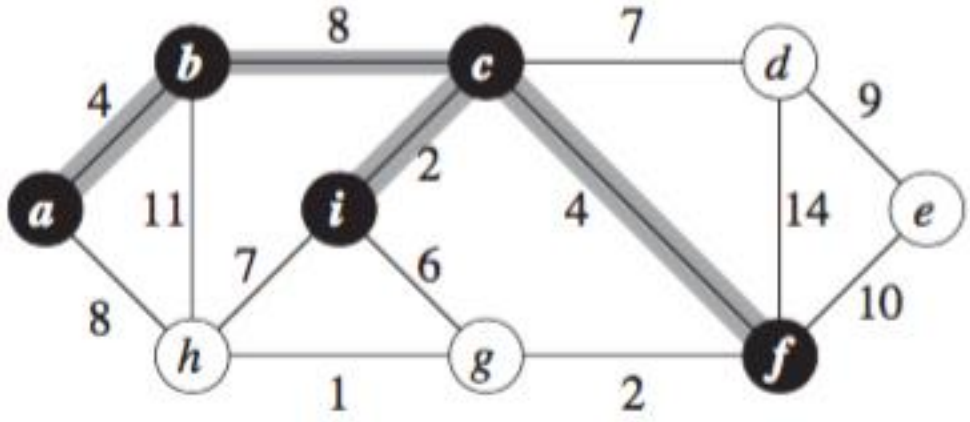
Example (cont.)

While $V_T \neq V$

Select an edge e(u, v) such that **u in V_T, v in V-V_T**, and w(e) is **minimum among all such edges**



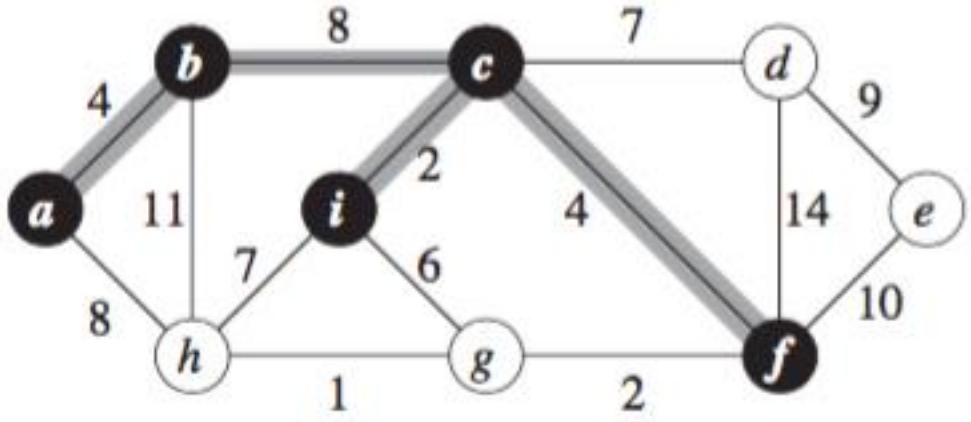
Choose edge e(c, f)



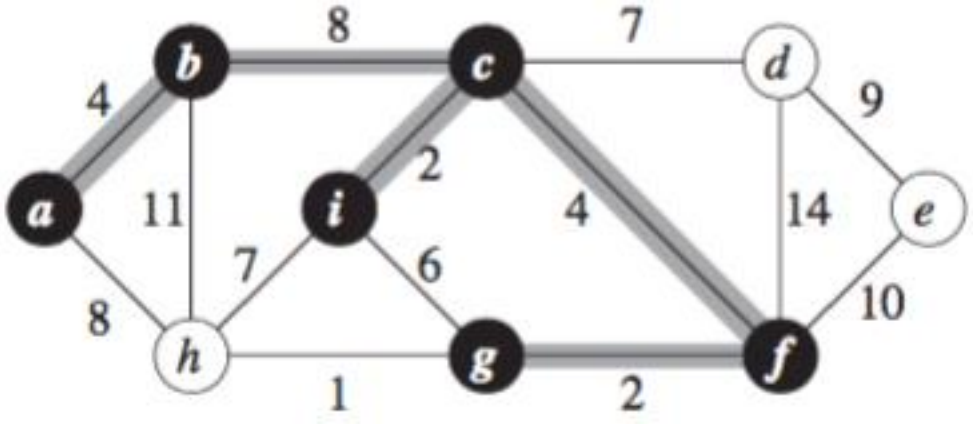
Example (cont.)

While $V_T \neq V$

Select an edge e(u, v) such that **u in V_T, v in $V-V_T$**, and w(e) is **minimum among all such edges**



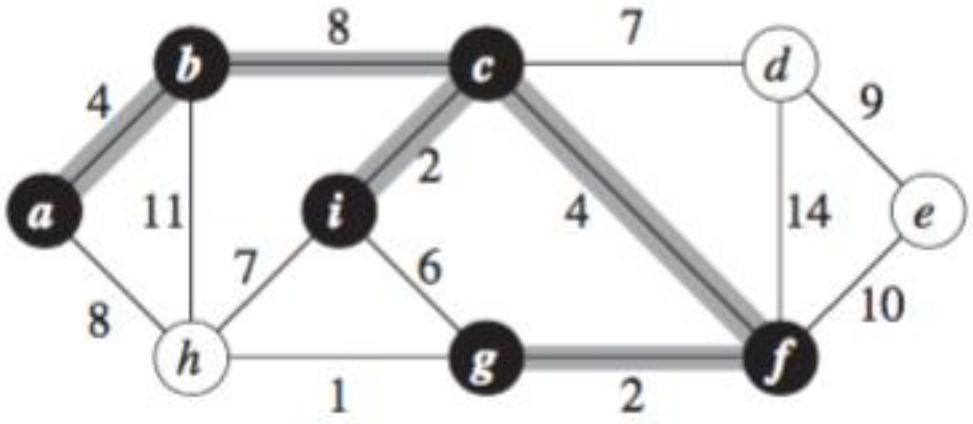
Choose edge e(f, g)



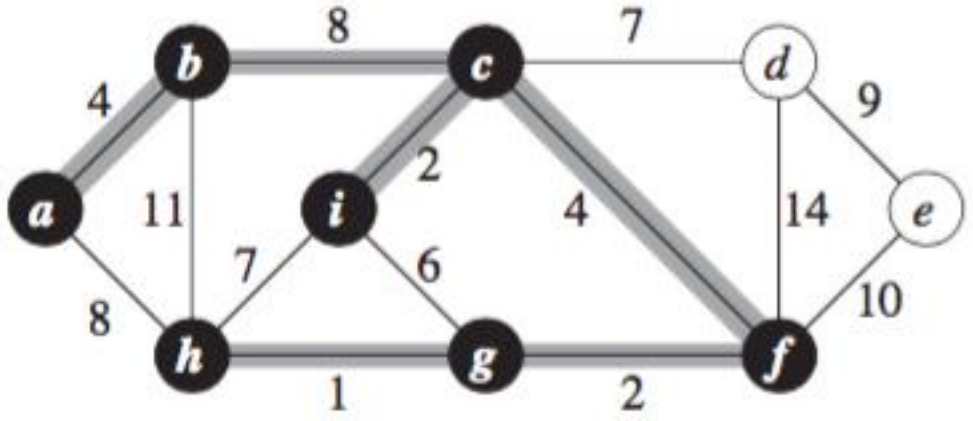
Example (cont.)

While $V_T \neq V$

Select an edge e(u, v) such that **u in V_T, v in $V-V_T$**, and w(e) is **minimum among all such edges**



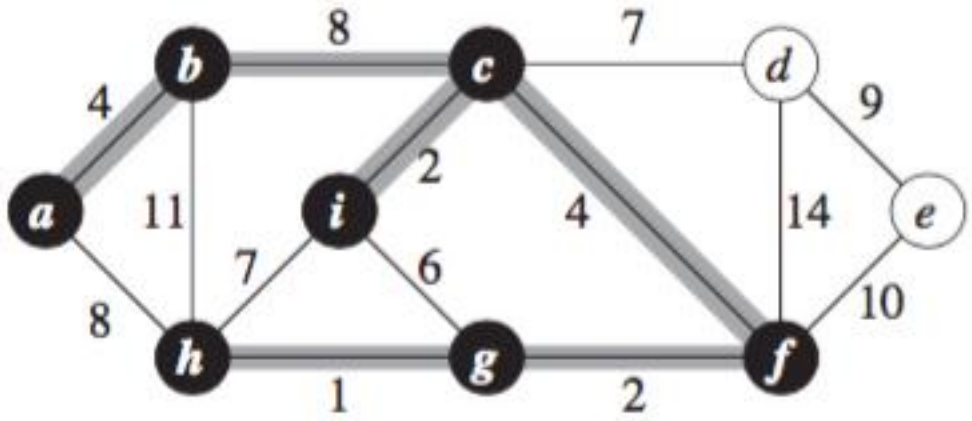
Choose edge e(g, h)



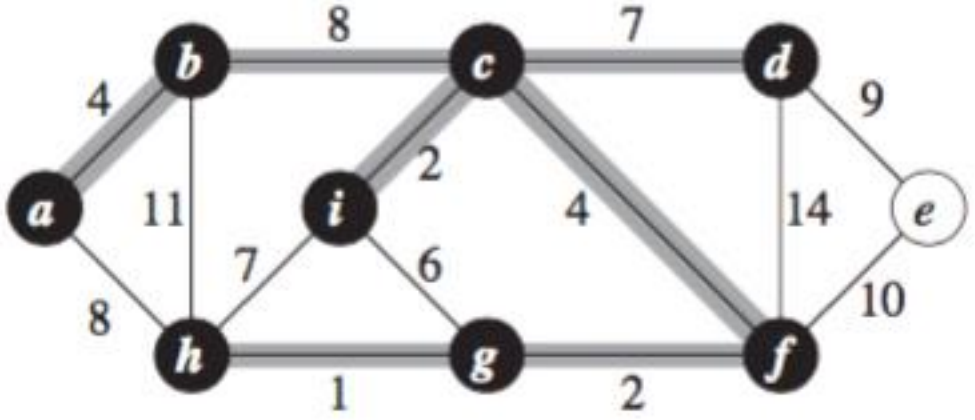
Example (cont.)

While $V_T \neq V$

Select an edge e(u, v) such that **u in V_T, v in $V-V_T$**, and w(e) is **minimum among all such edges**



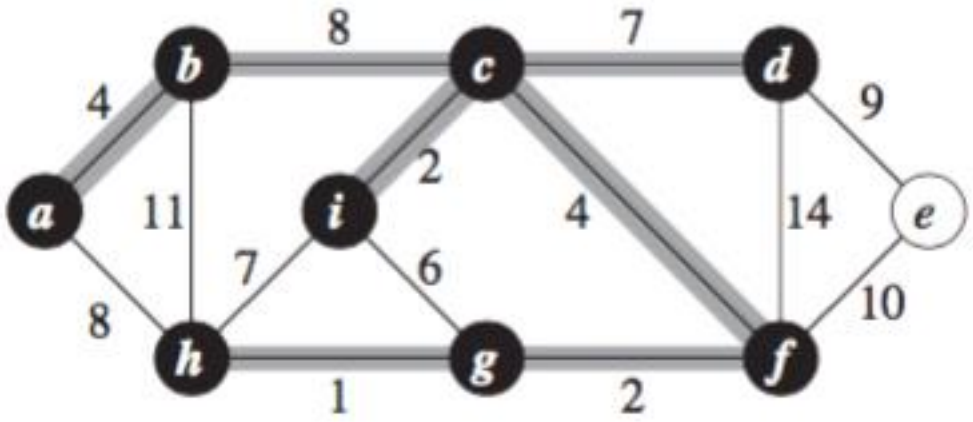
Choose edge e(c, d)



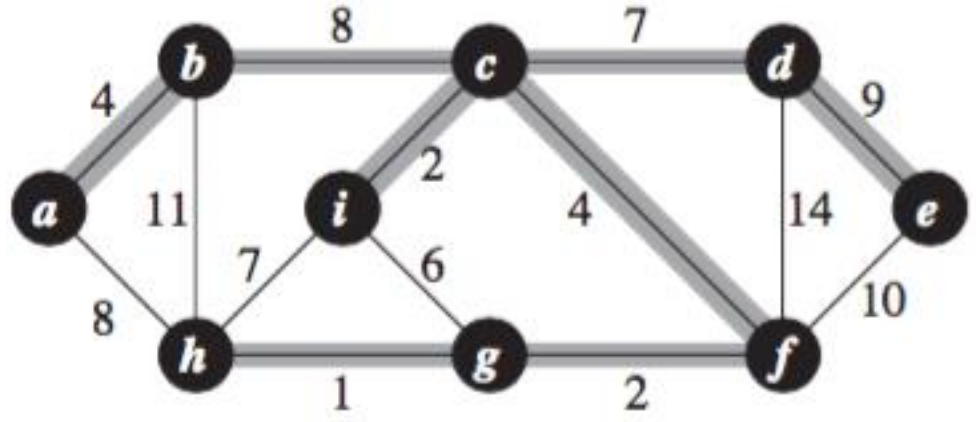
Example (cont.)

While $V_T \neq V$

Select an edge e(u, v) such that **u in V_T, v in V-V_T**, and w(e) is **minimum among all such edges**



Choose edge e(d, e)



Kruskal's MST Algorithm

- $E = \{e_1, e_2, \ldots, e_m\}$: set of edges in graph G
- E_T: set of edges in MST
- V_T: set of vertices in MST
- Process:

Initially, let $V_T = \{v_1, v_2, \ldots, v_n\}$ and $E_T = \Phi$

Sort edges in E such that $e_1 \leq e_2 \leq e_3 \ldots \leq e_m$

For i = 1 to m //

if $E_T \cup \{e_i\}$ **does not creat a cycle**

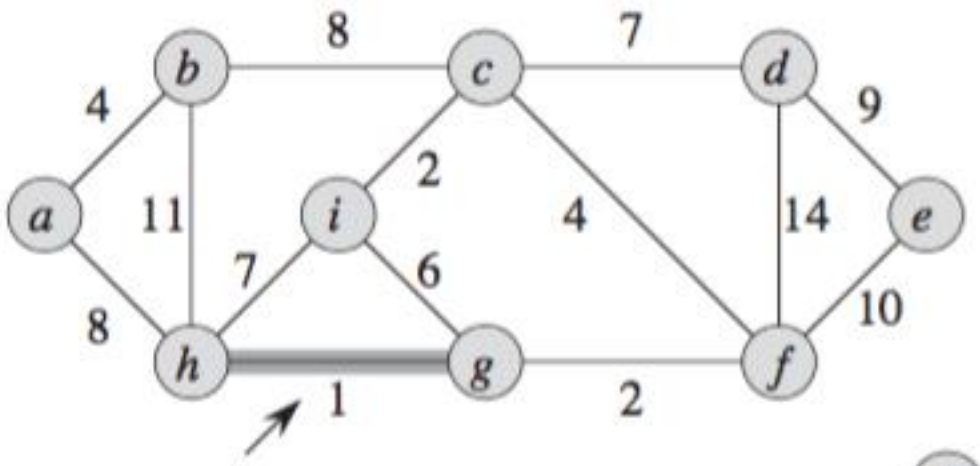
//greedily choose an edge

$E_T = E_T \cup \{e_i\}$

End for

Example

Edge e(h, g) has smallest weight



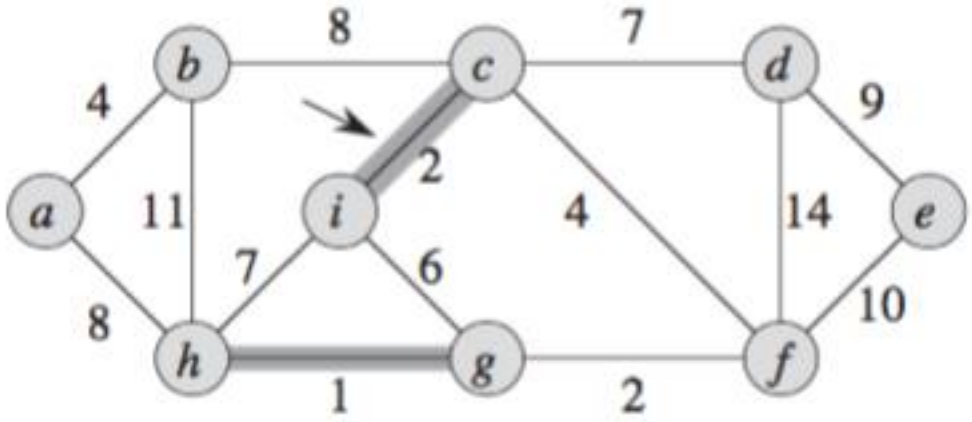
if E_T U $\{e_i\}$ **does not creat a cycle**

//greedily choose an edge

$E_T = E_T$ U $\{e_i\}$

1,2,2,4,4,6,7,7,8,8,9,10,11,14

Choose edge e(c, i)

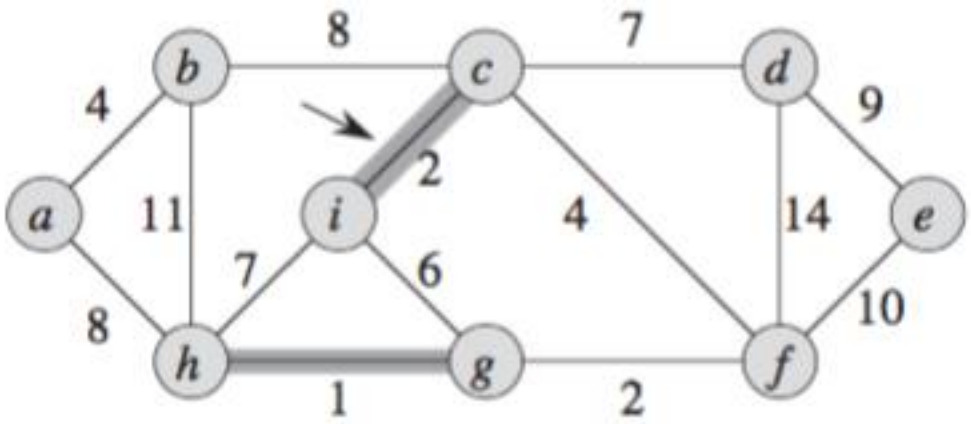


Example (cont.)

if E_T U $\{e_i\}$ **does not creat a cycle**

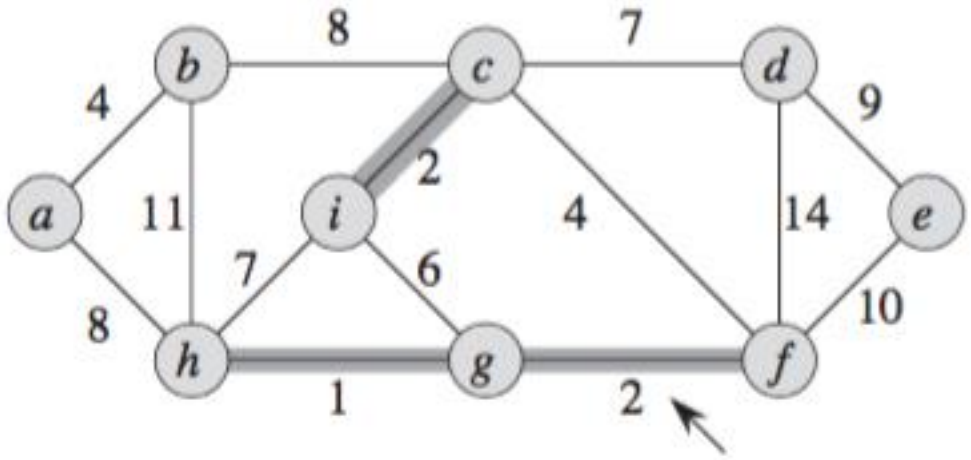
//greedily choose an edge

$E_T = E_T$ U $\{e_i\}$



1,2,2,4,4,6,7,7,8,8,9,10,11,14

Choose edge e(g, f)



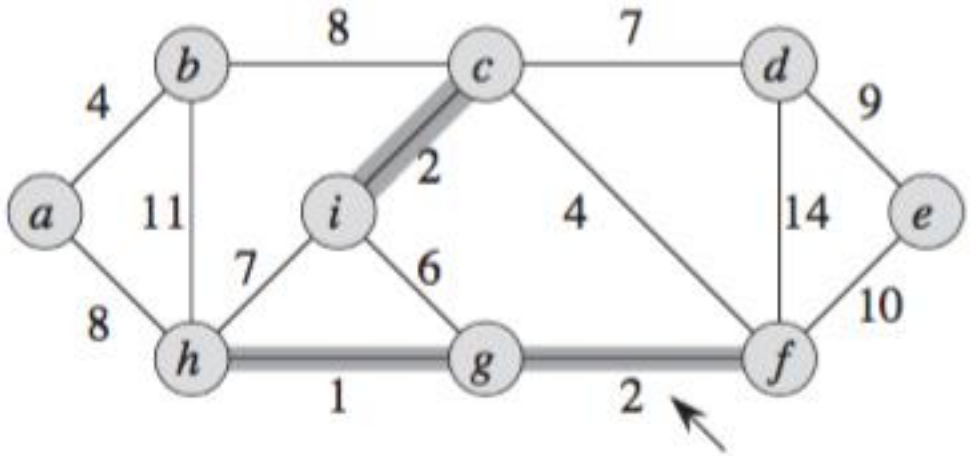
Example (cont.)

if E_T U $\{e_i\}$ **does not creat a cycle**

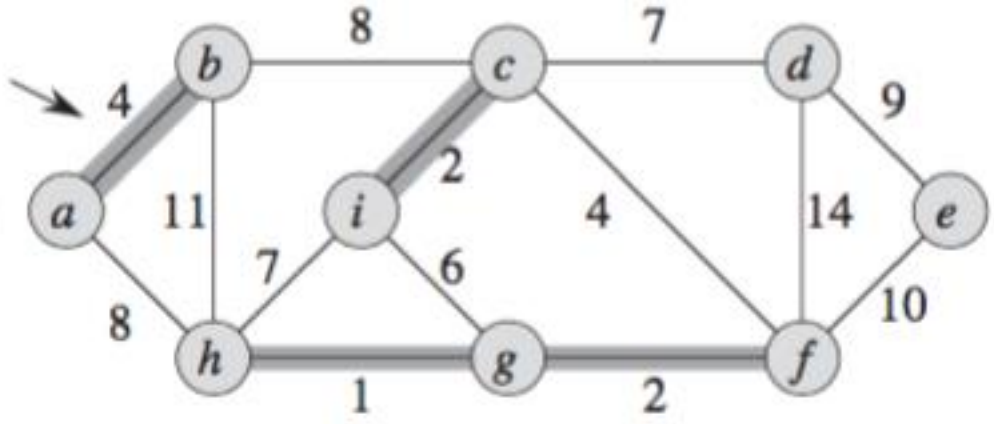
//greedily choose an edge

$E_T = E_T$ U $\{e_i\}$

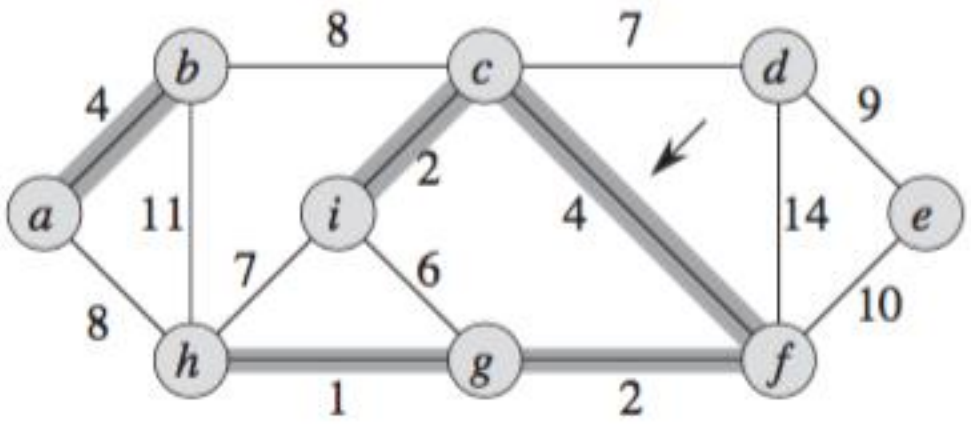
1,2,2,4,4,6,7,7,8,8,9,10,11,14



Choose edge e(a, b)



Example (cont.)



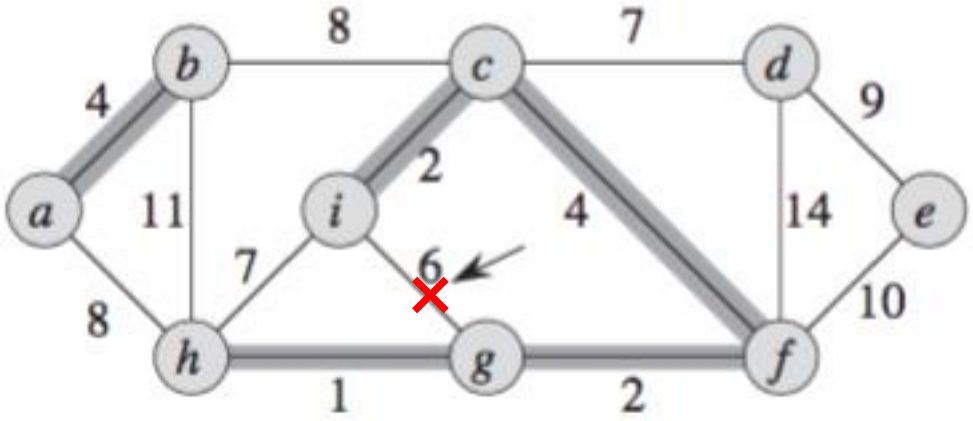
if E_T U $\{e_i\}$ **does not creat a cycle**

//greedily choose an edge

$E_T = E_T$ U $\{e_i\}$

1,2,2,4,4,6,7,7,8,8,9,10,11,14

Adding edge e(g, i) will create a cycle



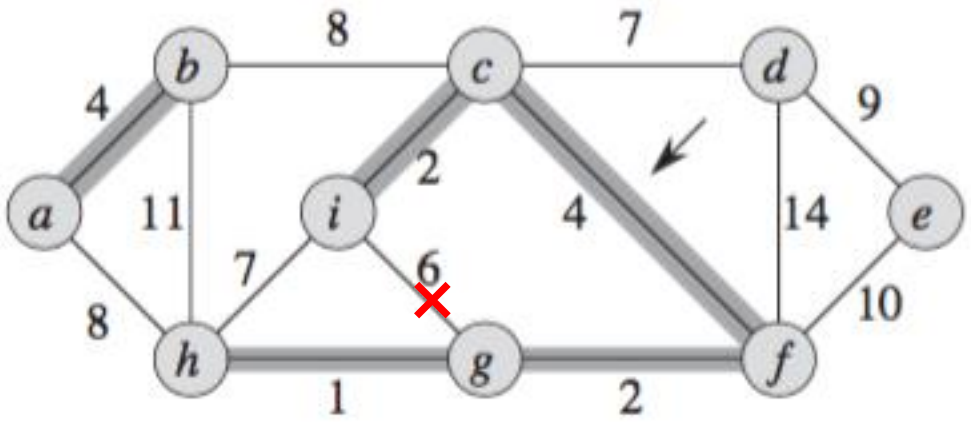
Example (cont.)

if E_T U $\{e_i\}$ **does not creat a cycle**

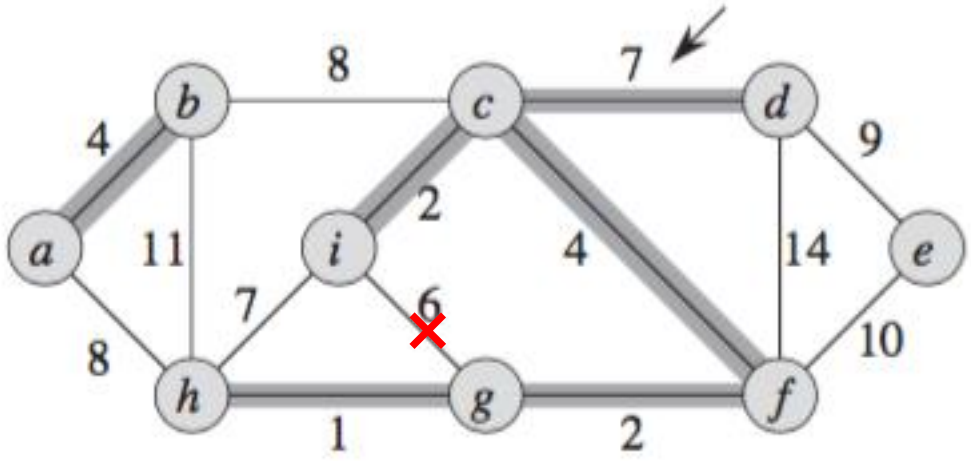
//greedily choose an edge

$E_T = E_T$ U $\{e_i\}$

1,2,2,4,4,6,7,7,8,8,9,10,11,14



Choose edge e(c, d)

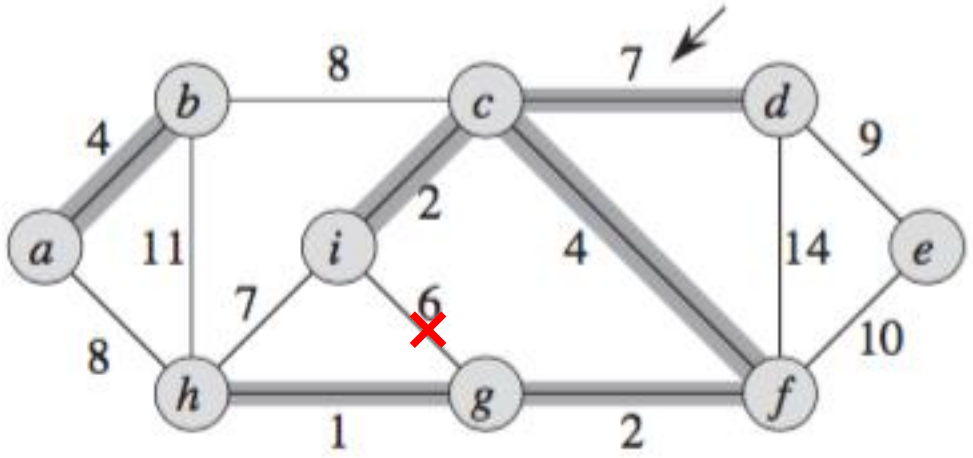


Example (cont.)

if E_T U $\{e_i\}$ **does not creat a cycle**

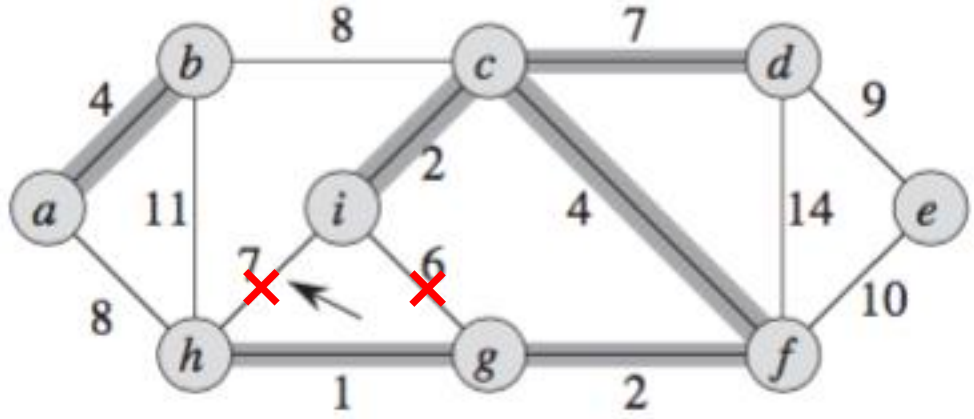
//greedily choose an edge

$E_T = E_T$ U $\{e_i\}$



1,2,2,4,4,6,7,7,8,8,9,10,11,14

Adding edge e(h, i) will create a cycle

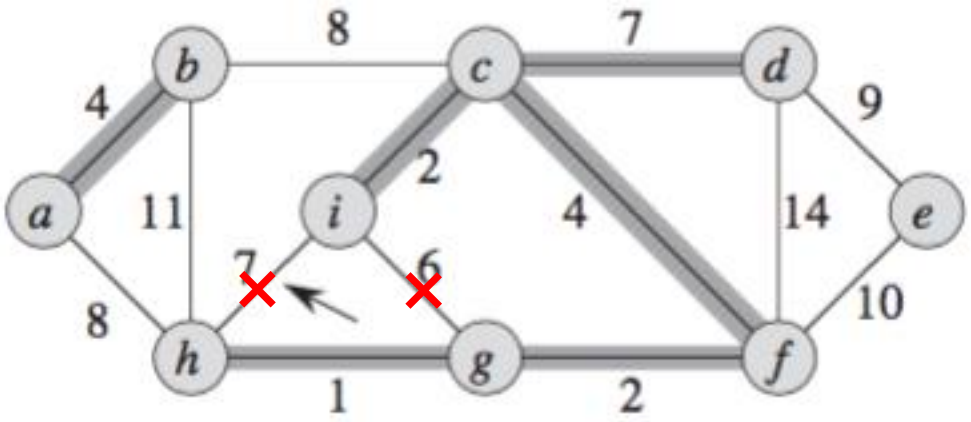


Example (cont.)

if E_T U $\{e_i\}$ **does not creat a cycle**

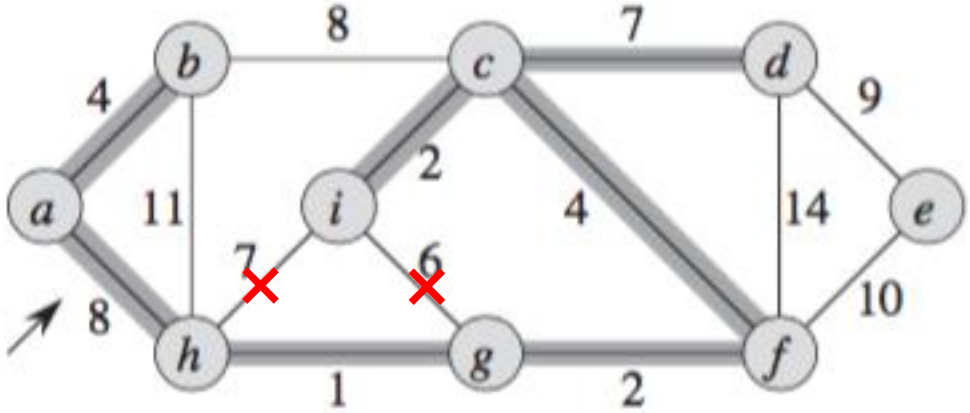
//greedily choose an edge

$E_T = E_T$ U $\{e_i\}$



1,2,2,4,4,6,7,7,8,8,9,10,11,14

Choose edge e(a, h)

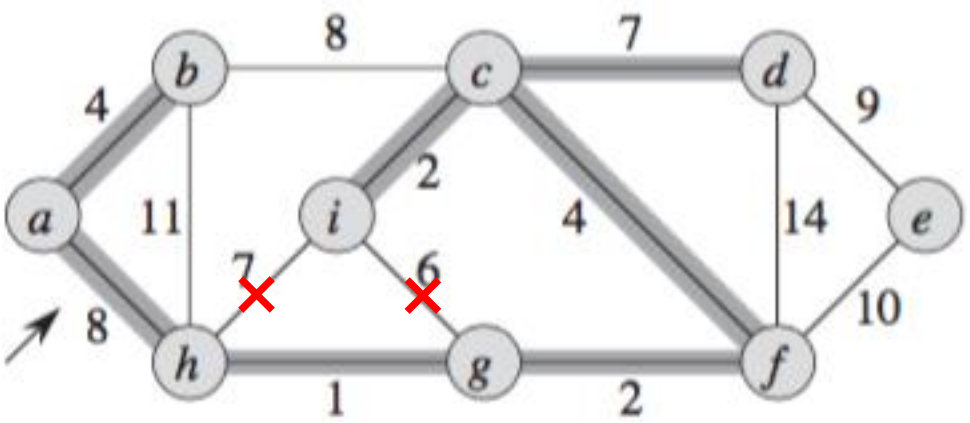


Example (cont.)

if E_T U $\{e_i\}$ **does not creat a cycle**

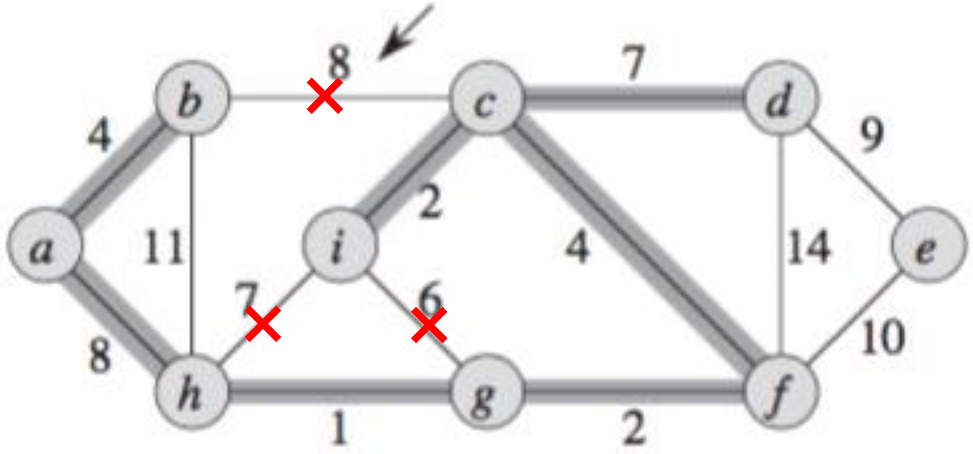
//greedily choose an edge

$E_T = E_T$ U $\{e_i\}$

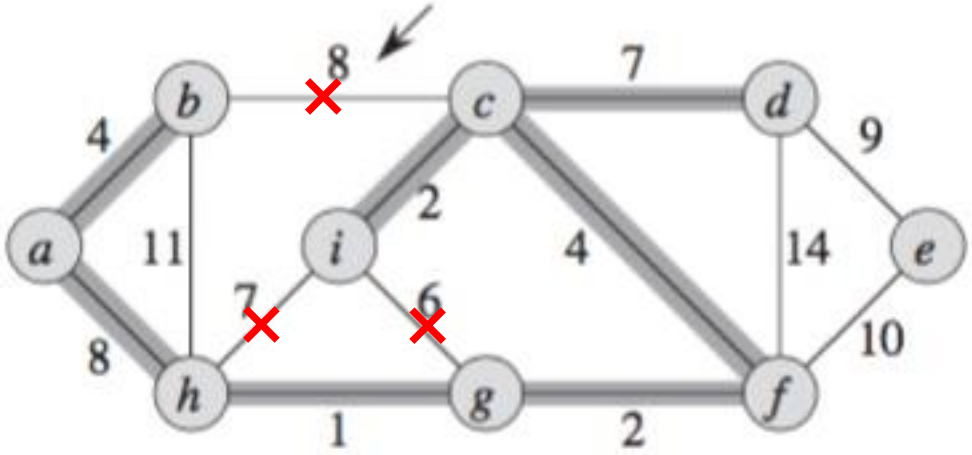


1,2,2,4,4,6,7,7,8,8,9,10,11,14

Adding edge e(b, c) will create a cycle



Example (cont.)



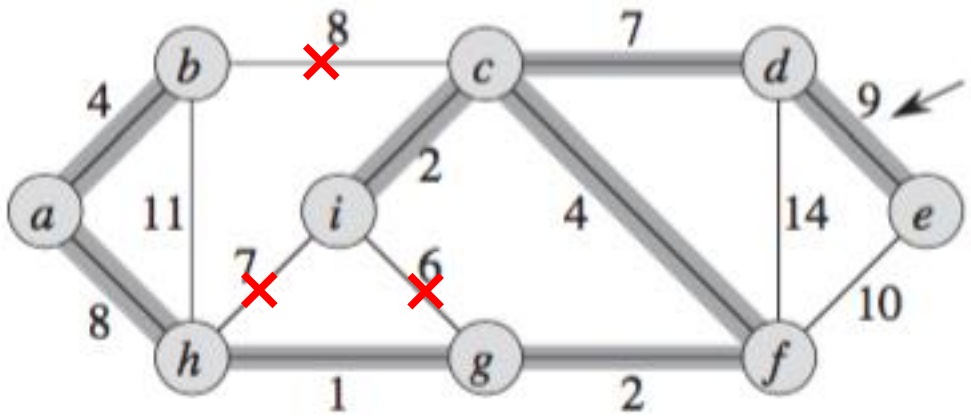
if E_T U $\{e_i\}$ **does not creat a cycle**

//greedily choose an edge

$E_T = E_T$ U $\{e_i\}$

1,2,2,4,4,6,7,7,8,8,9,10,11,14

Choose edge e(d, e)

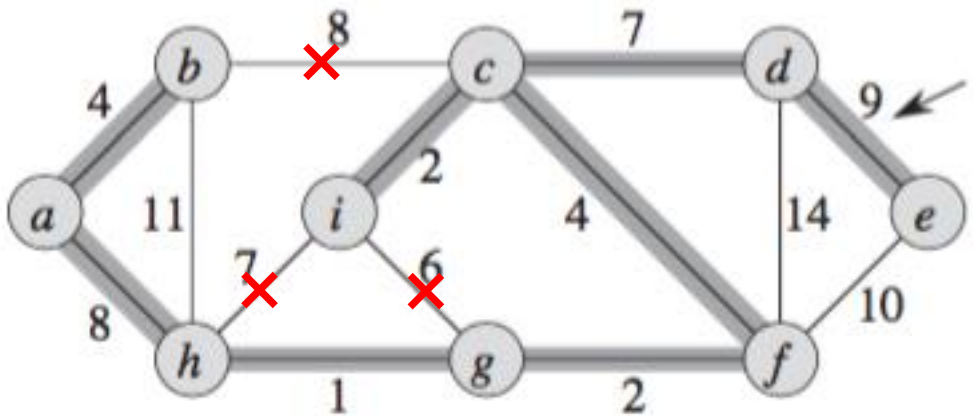


Example (cont.)

if E_T U $\{e_i\}$ **does not creat a cycle**

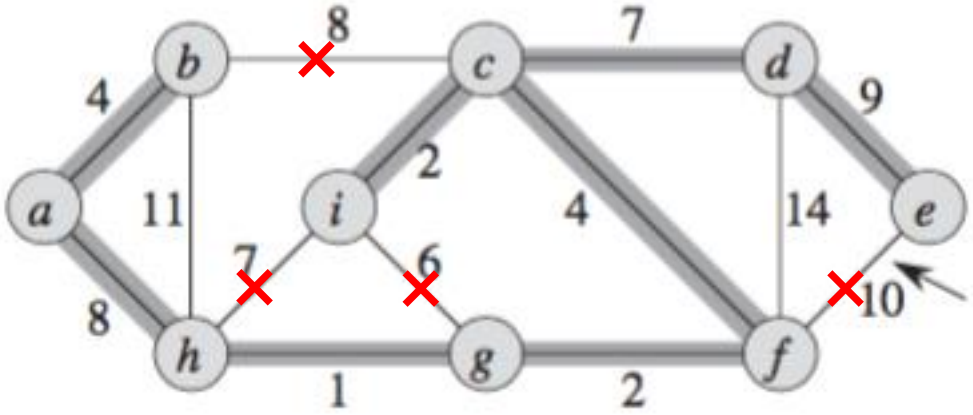
//greedily choose an edge

$E_T = E_T$ U $\{e_i\}$

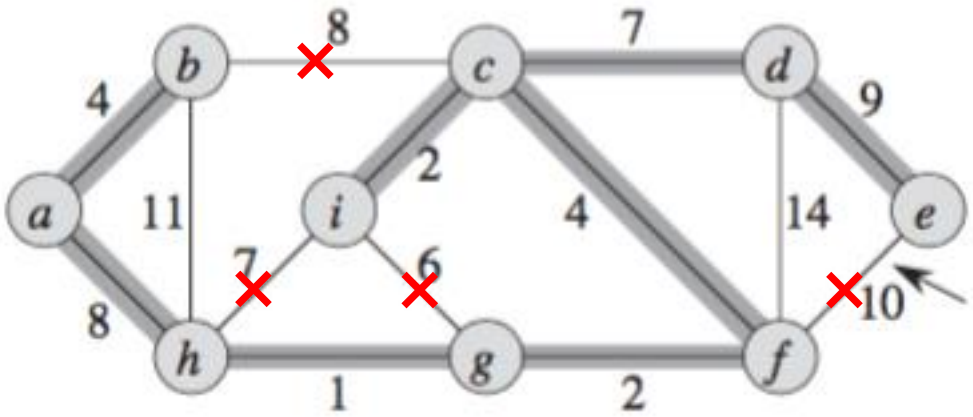


1,2,2,4,4,6,7,7,8,8,9,10,11,14

Adding edge e(e, f) will create a cycle



Example (cont.)



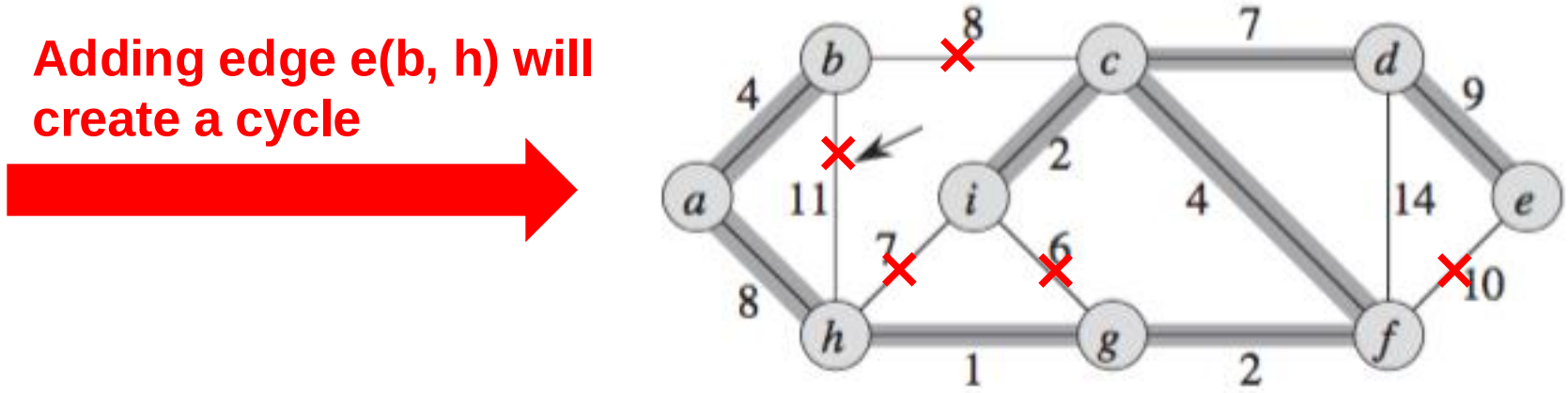
if E_T U $\{e_i\}$ **does not creat a cycle**

//greedily choose an edge

$E_T = E_T$ U $\{e_i\}$

1,2,2,4,4,6,7,7,8,8,9,10,11,14

Adding edge e(b, h) will create a cycle

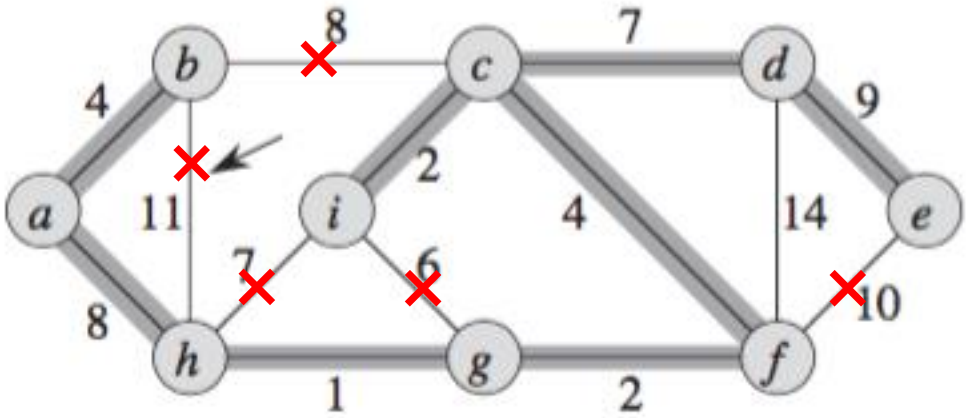


Example (cont.)

if E_T U $\{e_i\}$ **does not creat a cycle**

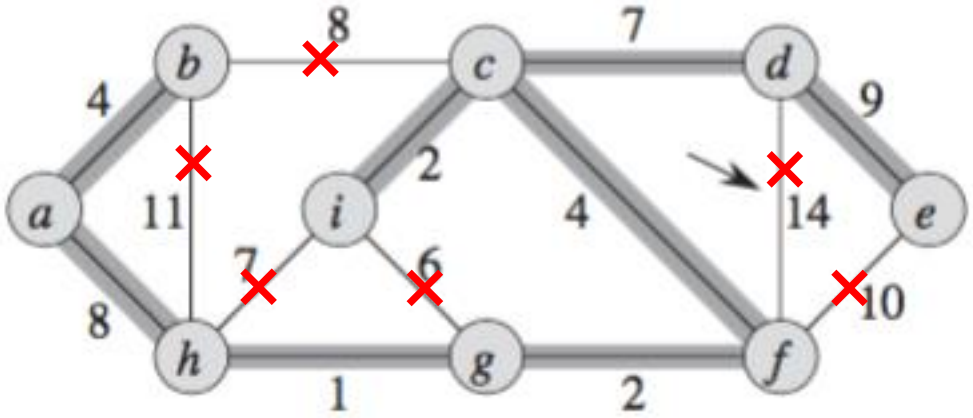
//greedily choose an edge

$E_T = E_T$ U $\{e_i\}$



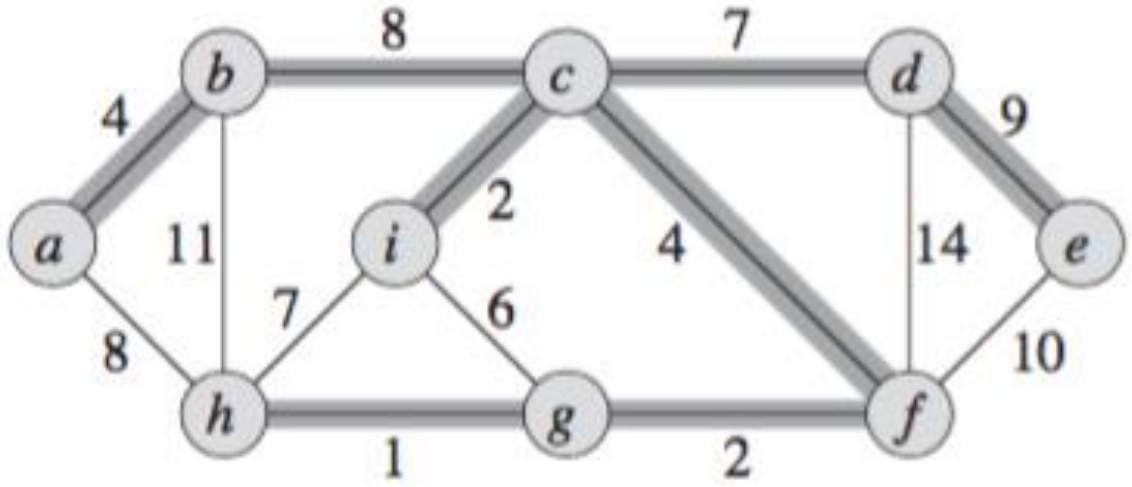
1,2,2,4,4,6,7,7,8,8,9,10,11,14

Adding edge e(d, f) will create a cycle



Prim's vs. Kruskal's

- Prim's: edge selection from connected component to new vertex set **(vertices of new added edge are not in the same vertex set)**
- Kruskal's: edge selection between two connected components **(check cycle after each selection)**

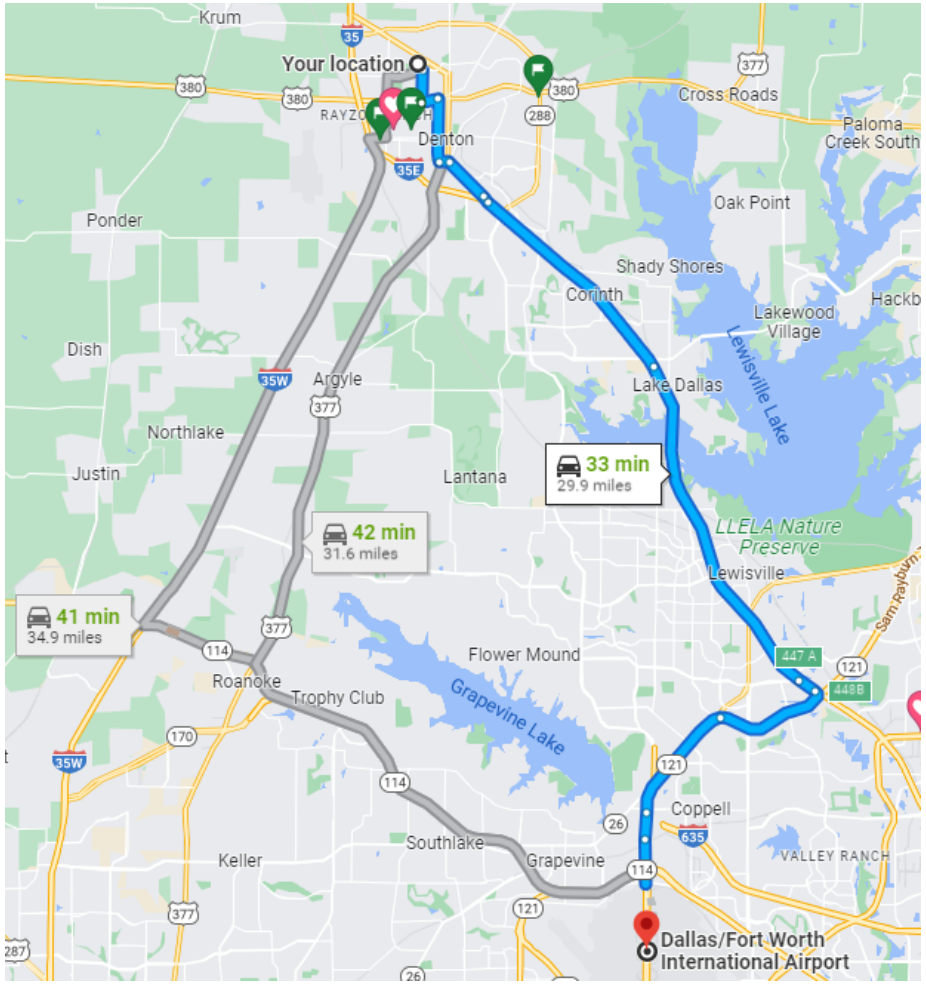


Prims vs Kruskal

Prim's Algorithm	Kruskal's Algorithm
Starts building the MST from any node.	Starts building the MST from the minimum weighted edge in the graph.
Uses adjacency matrix, binary heap, or Fibonacci heap.	Uses a disjoint set (Union-Find) data structure.
Runs faster on dense graphs.	Runs faster on sparse graphs.
Time complexity is $O(E \log V)$ with a binary heap and $O(E + V \log V)$ with a Fibonacci heap.	Time complexity is $O(E \log V)$.
The next node included must be connected to the currently traversed node.	The next edge included may or may not connect directly to the current set of nodes but should not form a cycle.
Traverses each node multiple times to get the minimum distance.	Traverses each edge once, deciding to accept or reject based on cycle formation.
Greedy algorithm.	Greedy algorithm.

Single-Source Shortest Paths

Which path do you prefer?



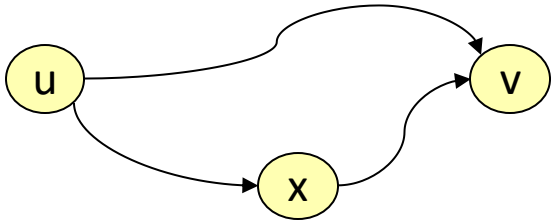
Shortest-Paths Problems

- Graph G(V, E)
  - V = $\{v_0, v_1, v_2, \ldots, v_n\}$: set of vertices
  - $w(v_i, v_j)$: weight of edge $e(v_i, v_j)$
  - $p(v_0, v_1, v_2, \ldots, v_k)$: a path from v_0 to v_k
  - $w(p) = w(v_0, v_1) + w(v_1, v_2) + \ldots + w(v_{k-1}, v_k)$: weight of path p
- Problems:
  - Single-Source Shortest-Paths: find a shortest path **from a source s to every vertex v in V**.
  - Single-Destination Shortest-Path: find a shortest path **to a destination d from every vertex v in V**.
  - Single-Pair Shortest-Path: find a shortest path **from source s to destination d**.

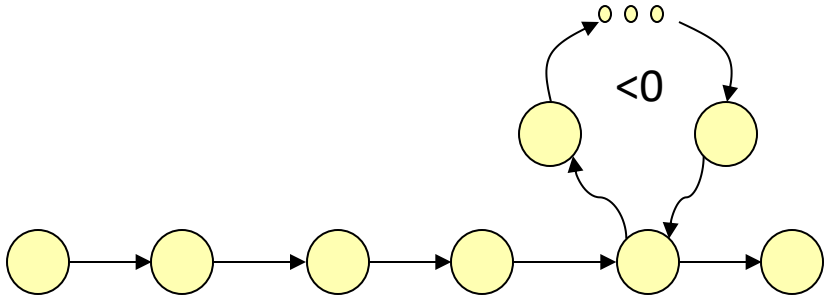
Shortest Paths

- Shortest-paths tree rooted at source s
- Subpath of a shortest path is a shortest path
- Triangle Inequality:

$$\delta(u,v) \leq \delta(u,x) + \delta(x,v)$$



- Well defined: some paths may not exist if there is a negative-weight cycle in graph



Bellman-Ford Algorithm

- BFS-based algorithm, shortest paths (tree) easy to reconstruct.
- find the shortest path from a single source node s to all other nodes v in graph
- Bellman-Ford(G, w, s)

  For each $v \in V$, $d[v] \leftarrow \infty$ and $d[s] \leftarrow 0$ **//initiation**

  For i =1,…,|V|-1 do

    For each edge $e(u,v) \in E$ do **//from source to v, if node u is added**

      $d[v] \leftarrow \min\{d[v], d[u]+w(u,v)\}$

      //update local shortest paths for each vertex

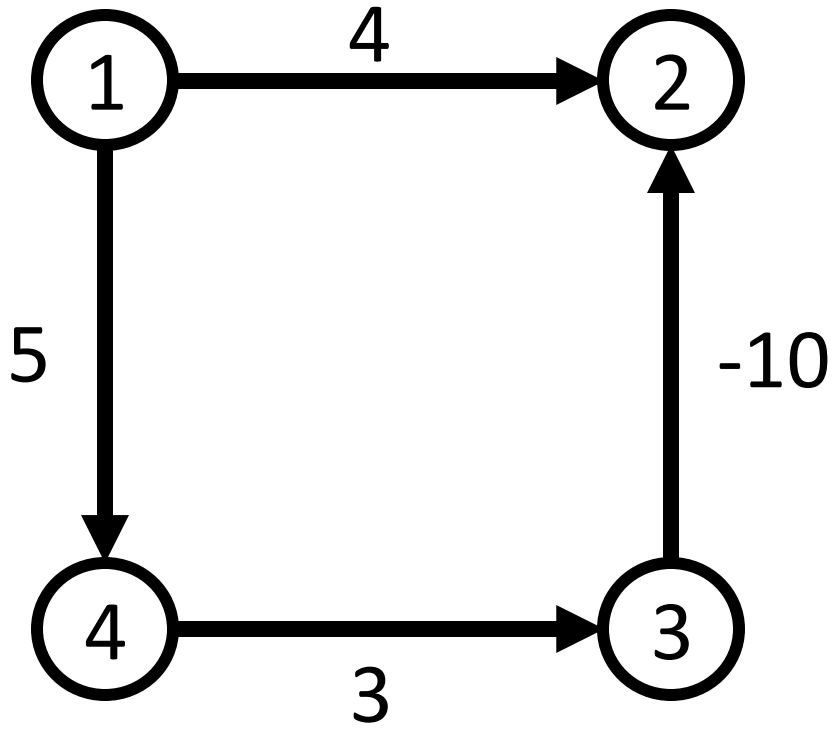
  For each $v \in V$ do

    if $d[v] > d[u] + w(u,v)$ then no solution **//negative weight cycle**

  Return d[v] = actual shortest-path weight $\delta(s,v)$

- Running Time: O(|V||E|)

Example: Bellman-Ford



Relaxation:

If (d[u]+w(u, v)<d[v])

d[v] = d[u]+w(u, v)

|V|=n=vertices

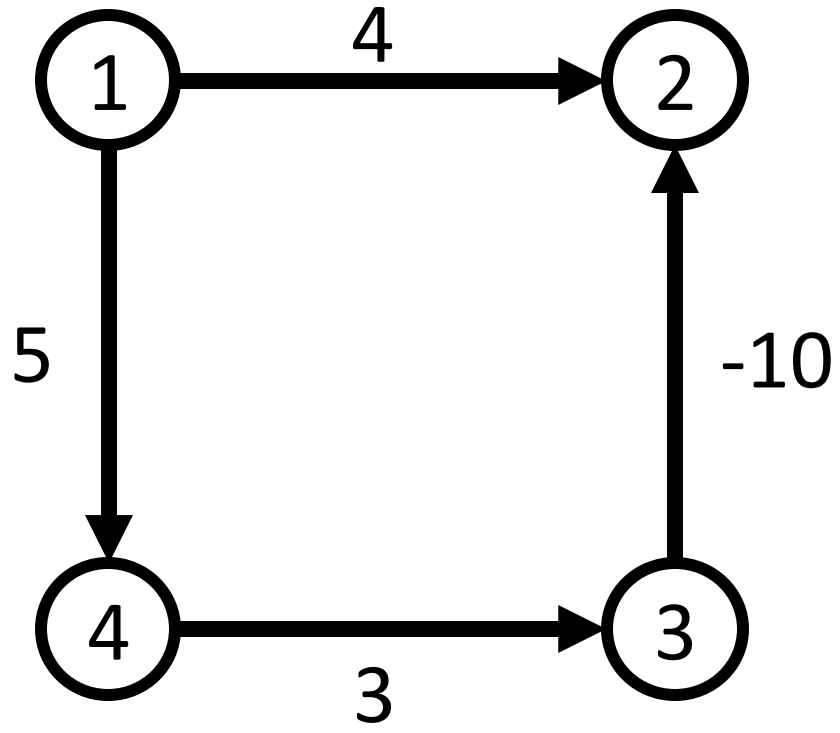
|v|-1

take Node 1 as the source:

•We would initialize d[1]=0 (distance from the source to itself is zero).

	1	2	3	4
Initial	0	∞	∞	∞
1				
2				
3				

Example: Bellman-Ford



Edges list - - > (3,2),(4,3),(1,4),(1,2)

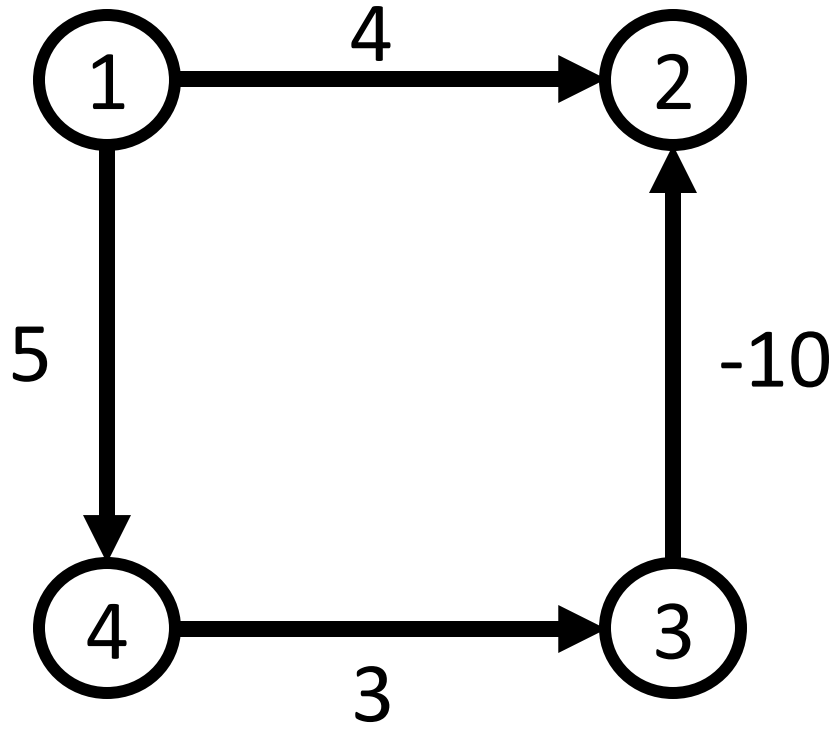
Relaxation:
If (d[u]+w(u, v)<d[v])
d[v] = d[u]+w(u, v)

Edge relaxation (in order of edge list):

- **Edge (3 → 2)**: Skipped since $d[3] = \infty$ (Node 3 is not reachable yet).
- **Edge (4 → 3)**: Skipped since $d[4] = \infty$.
- **Edge (1 → 4)**: Updates $d[4]$ to 5 (0 + 5).
- **Edge (1 → 2)**: Updates $d[2]$ to 4 (0 + 4).

Iteration	d(1)	d(2)	d(3)	d(4)
Initial	0	∞	∞	∞
1	0	4	∞	5
2				
3				

Example: Bellman-Ford



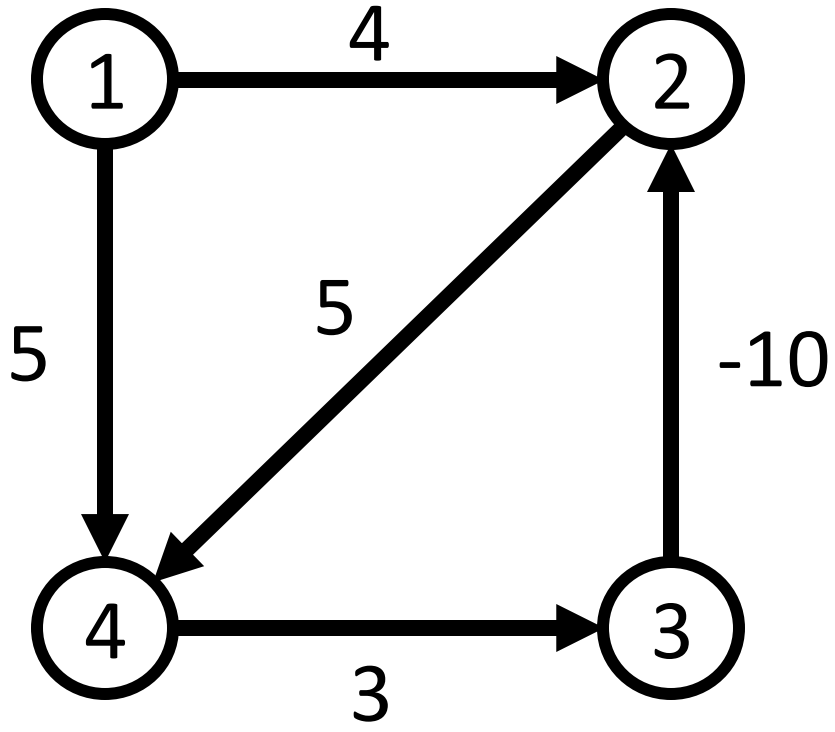
Edges list - - > (3,2),(4,3),(1,4),(1,2)

Relaxation:
If (d[u]+w(u, v)<d[v])
d[v] = d[u]+w(u, v)

- **Initial**: $d[1] = 0, d[2] = \infty, d[3] = \infty, d[4] = \infty$
- **1st Iteration**: After relaxing each edge once, $d[2] = 4$, $d[3] = \infty$, and $d[4] = 5$.
- **2nd Iteration**: Updates $d[3]$ to 8 and maintains other values.
- **3rd Iteration**: Updates $d[2]$ to -2 due to the negative weight edge $(3 \rightarrow 2)$.

Iteration	1	2	3	4
Initial	0	∞	∞	∞
1	0	4	∞	5
2	0	4	8	5
3	0	-2	8	5

Example: Bellman-Ford



Relaxation:

If (d[u]+w(u, v)<d[v])

d[v] = d[u]+w(u, v)

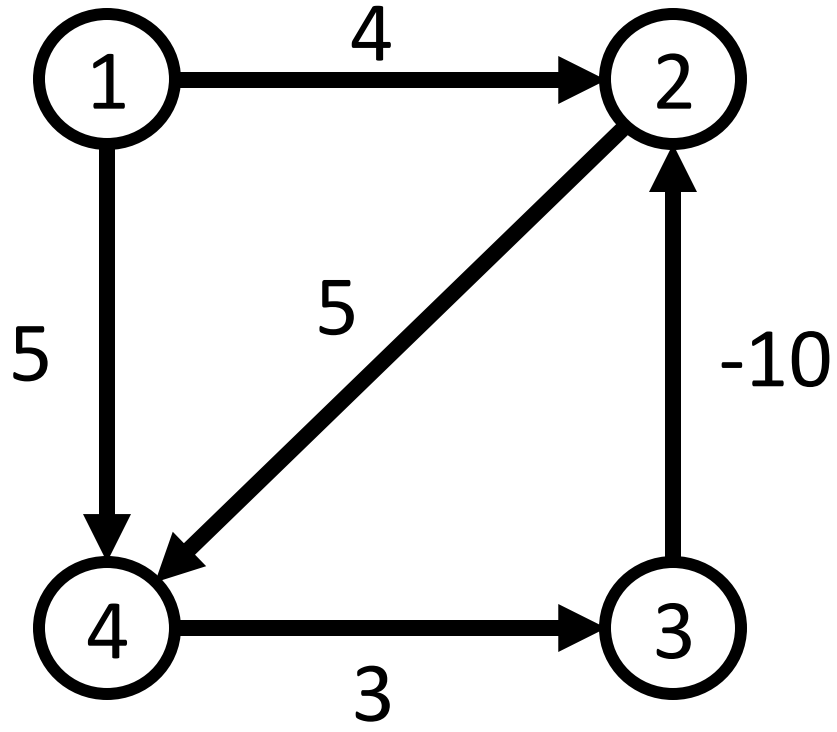
|V|=n=vertices

|v|-1

	1	2	3	4
Initial	0	∞	∞	∞
1	0	4	∞	5
2	0	4	8	5
3	0	-2	8	3
4	0	-2	6	3

Edges list - - > (3,2),(4,3),(1,4),(1,2),(2,4)

Example: Bellman-Ford



Relaxation:

If (d[u]+w(u, v)<d[v])

d[v] = d[u]+w(u, v)

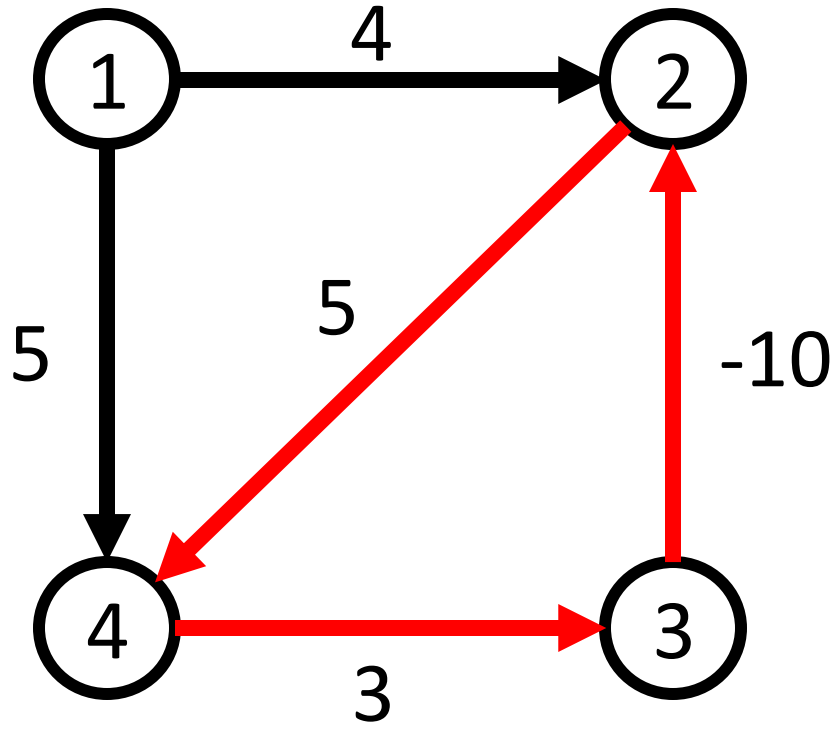
|V|=n=vertices

|v|-1

	1	2	3	4
Initial	0	∞	∞	∞
1	0	4	∞	5
2	0	4	8	5
3	0	-2	8	5
4	0	-2	8	5

Edges list - - > (3,2),(4,3),(1,4),(1,2),(2,4)

Example: Bellman-Ford



Relaxation:

If (d[u]+w(u, v)<d[v])

d[v] = d[u]+w(u, v)

|V|=n=vertices

|v|-1

	1	2	3	4
Initial	0	∞	∞	∞
1	0	4	∞	5
2	0	4	8	5
3	0	-2	8	5
4	0	-2	8	5

Edges list - - > (3,2),(4,3),(1,4),(1,2)

Dijkstra's Shortest Paths

- greedy algorithm used to find the shortest paths from a single source node to all other nodes in a weighted graph
- all edge weights are non-negative
- Combines BFS and Prim's algorithm
- Better than BFS since non-negative weights

Dijkstra’s Shortest Paths

Ideas:

Initialization:

1. Create a set of unvisited nodes.
2. Assign an initial tentative distance of infinity (∞\infty∞) to every node except the source node, which is set to 0.
3. Keep a record of the previous node for each node (used to reconstruct the shortest path later).

Processing Nodes:

1. Select the unvisited node with the smallest tentative distance (let's call it the **current node**).
2. For each neighbor of the current node:
    1. Calculate the tentative distance via the current node.
    2. If this distance is smaller than the previously recorded distance, update it.
3. Mark the current node as visited once all its neighbors have been processed.

Repeat:

1. Continue selecting the node with the smallest tentative distance until all nodes are visited or the target node is reached.

Result:

1. The shortest distance to each node is stored in the distance array.
2. The path to a node can be reconstructed using the record of previous nodes.

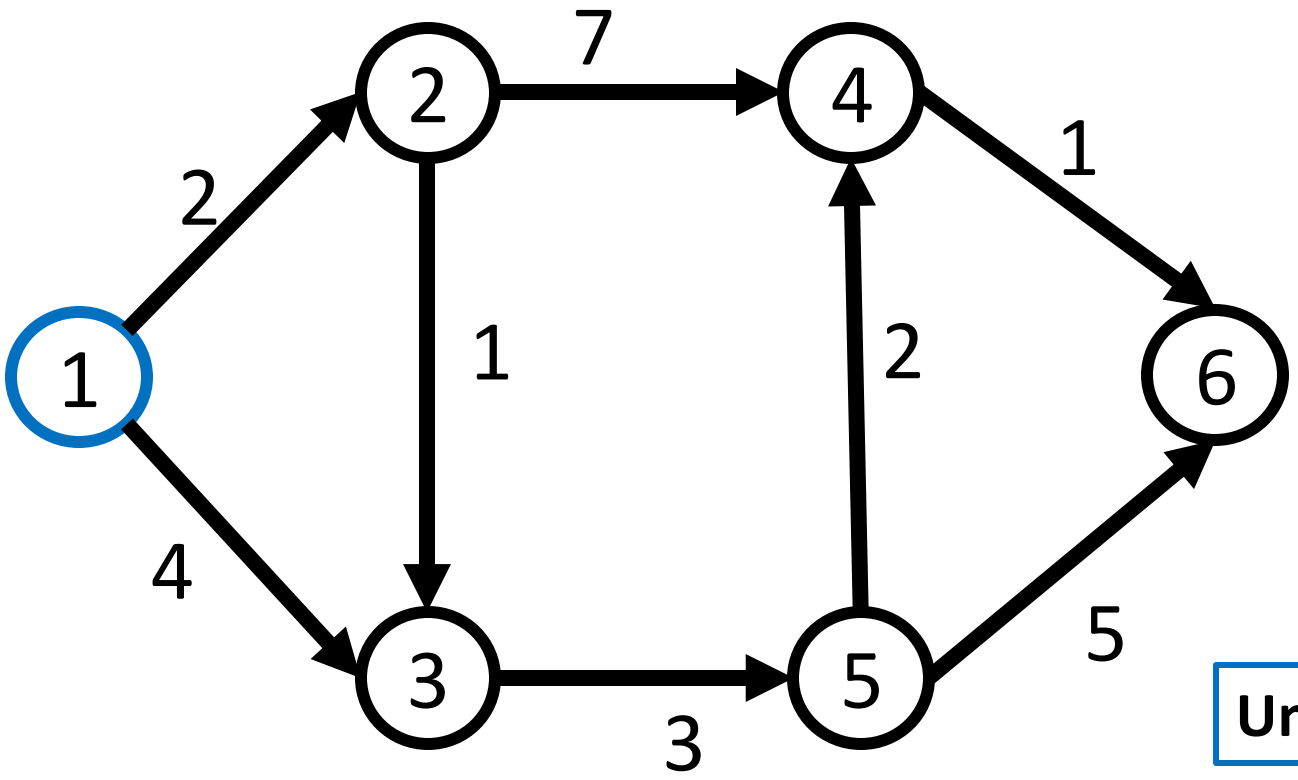
Dijkstra's Shortest Paths

- Combines BFS and Prim's algorithm

```
For each v ∈ V
        do d[v] ← ∞ and d[s] ← 0                    // initiation
V' ← ∅
For each u ∈ V
        Let u in V-V' and d[u] is minimum    //select a local shortest path
V' ← V' U {u}
For each v adjacent to u do
        d[v] ← min{d[v], d[u]+w(u,v)}
        //update shortest paths for each neighboring vertex
```

- Running Time: $O(|V|^2)$

Example: Dijkstra



Unvisited Nodes: {1, 2, 3, 4, 5, 6,}

	1	2	3	4	5	6
Initial	**0**	**∞**	**∞**	**∞**	**∞**	**∞**

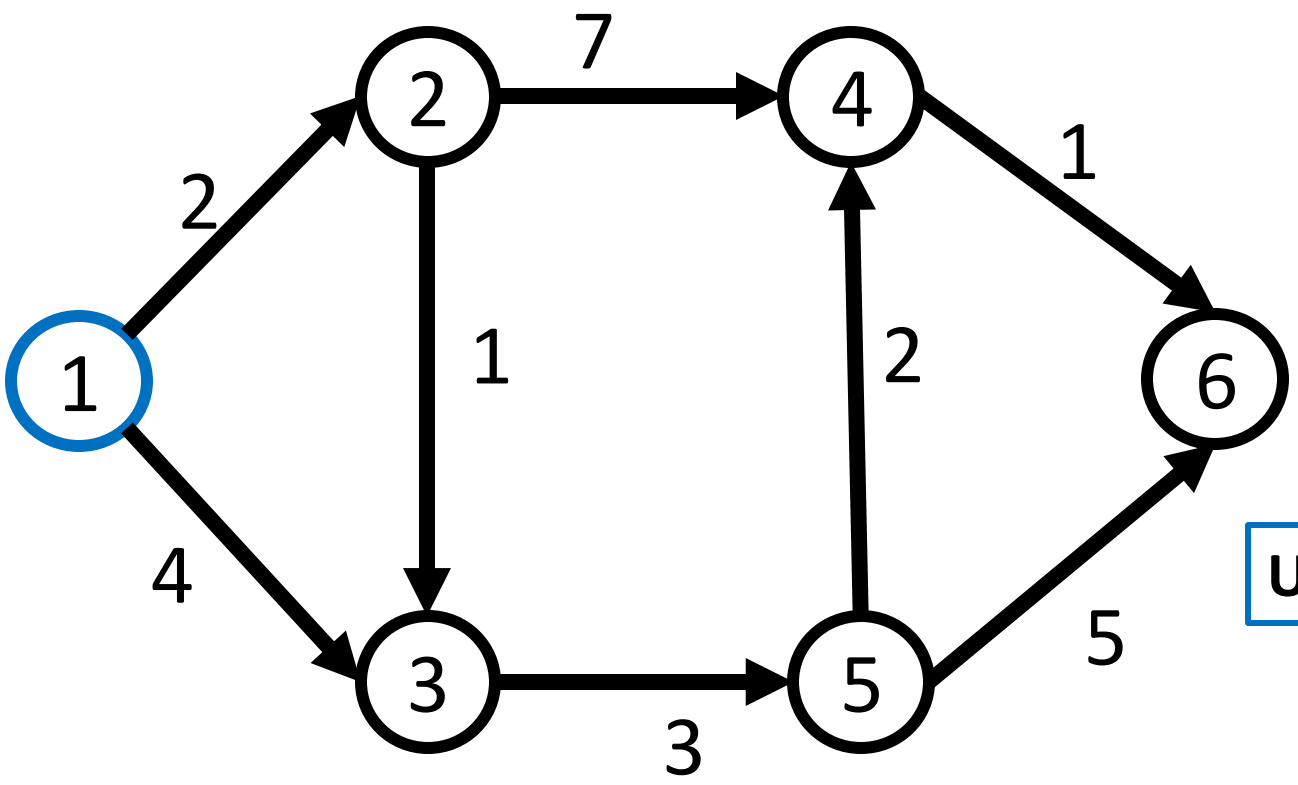
Next Step: examine the edges leaving from node-1

Example: Dijkstra

For each v adjacent to u do

d[v] ← min{d[v], d[u]+w(u,v)}

//update shortest paths for each neighboring vertex



Unvisited Nodes: {1, 2, 3, 4, 5, 6,}

	1	2	3	4	5	6
Initial	**0**	**2**	**4**	**∞**	**∞**	**∞**

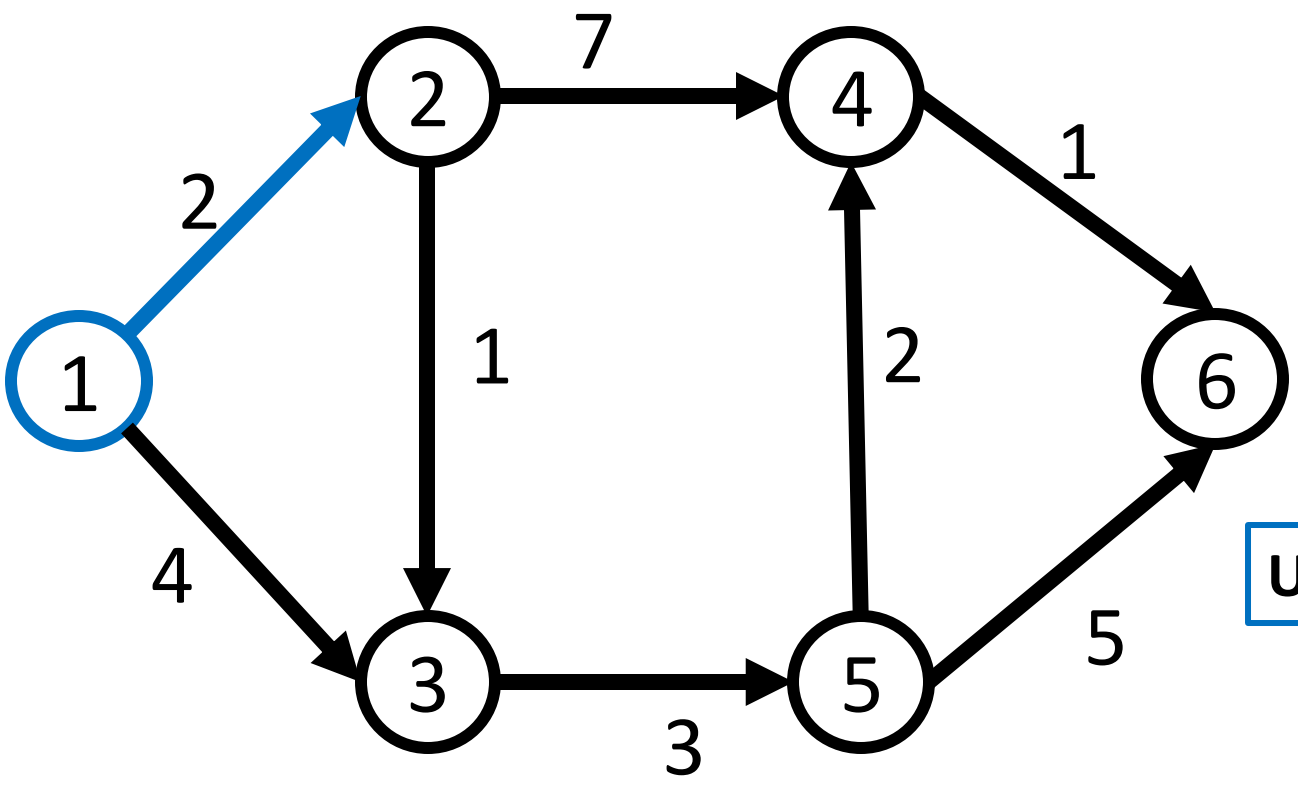
Next : from the table, pick the smallest edge of which the vertex has not been visited, node-2

Example: Dijkstra

For each v adjacent to u do

d[v] ← min{d[v], d[u]+w(u,v)}

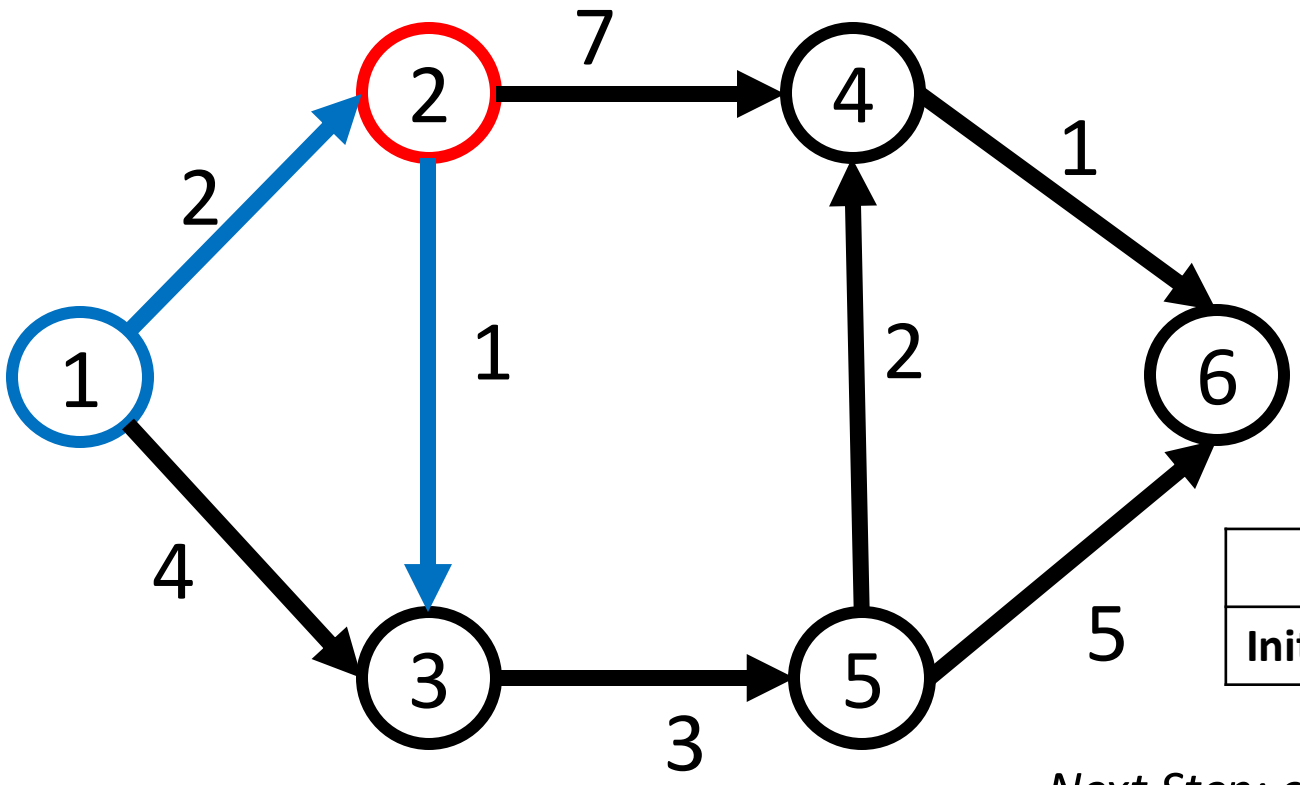
//update shortest paths for each neighboring vertex



Unvisited Nodes: {1, 2, 3, 4, 5, 6,}

	1	2	3	4	5	6
Initial	**0**	**2**	**4**	**∞**	**∞**	**∞**

Next : pick the smallest edge of which the vertex has not been visited, node-2, then mark node-2 is visited



Unvisited Nodes: {1, 2, 3, 4, 5, 6,}

	1	2	3	4	5	6
Initial	**0**	**2**	**4**	**∞**	**∞**	**∞**

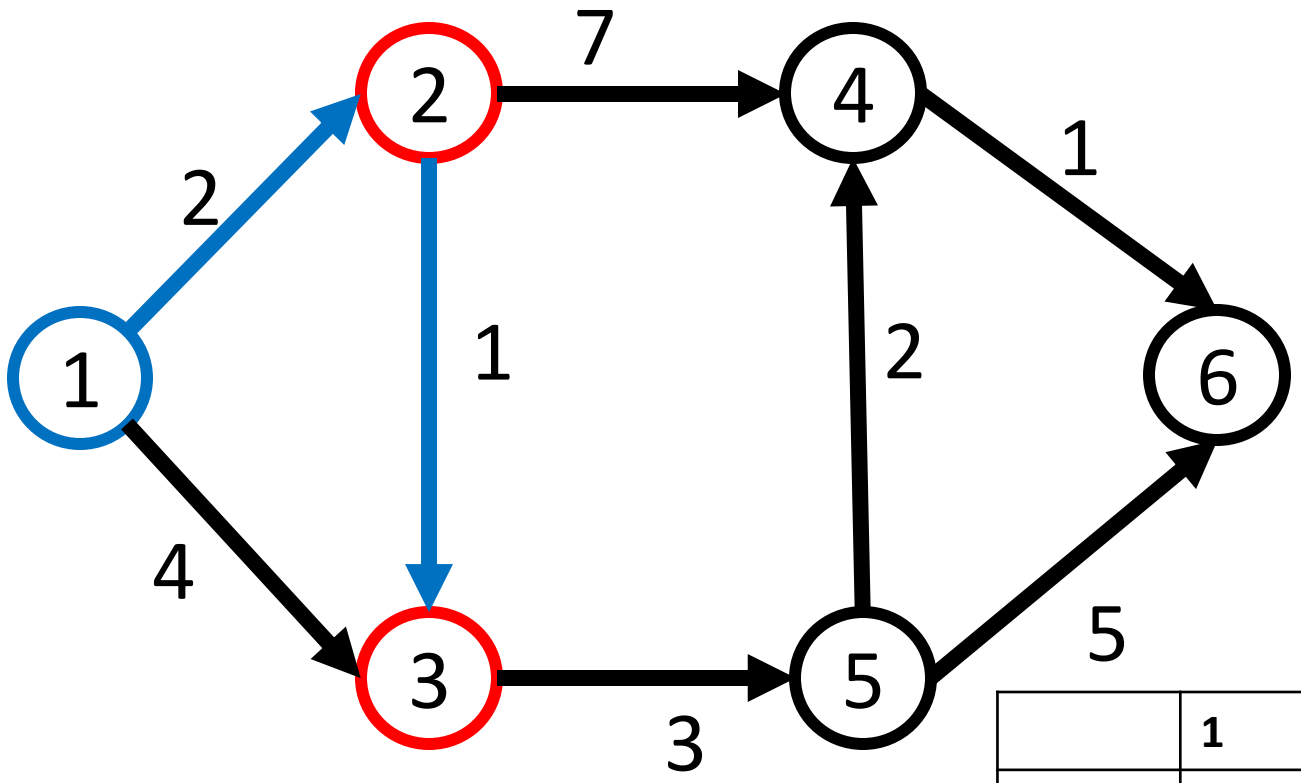
Next Step: examine the edges leaving from node-2, and update distance

For each v adjacent to u do

d[v] ← min{d[v], d[u]+w(u,v)}

//update shortest paths for each neighboring vertex

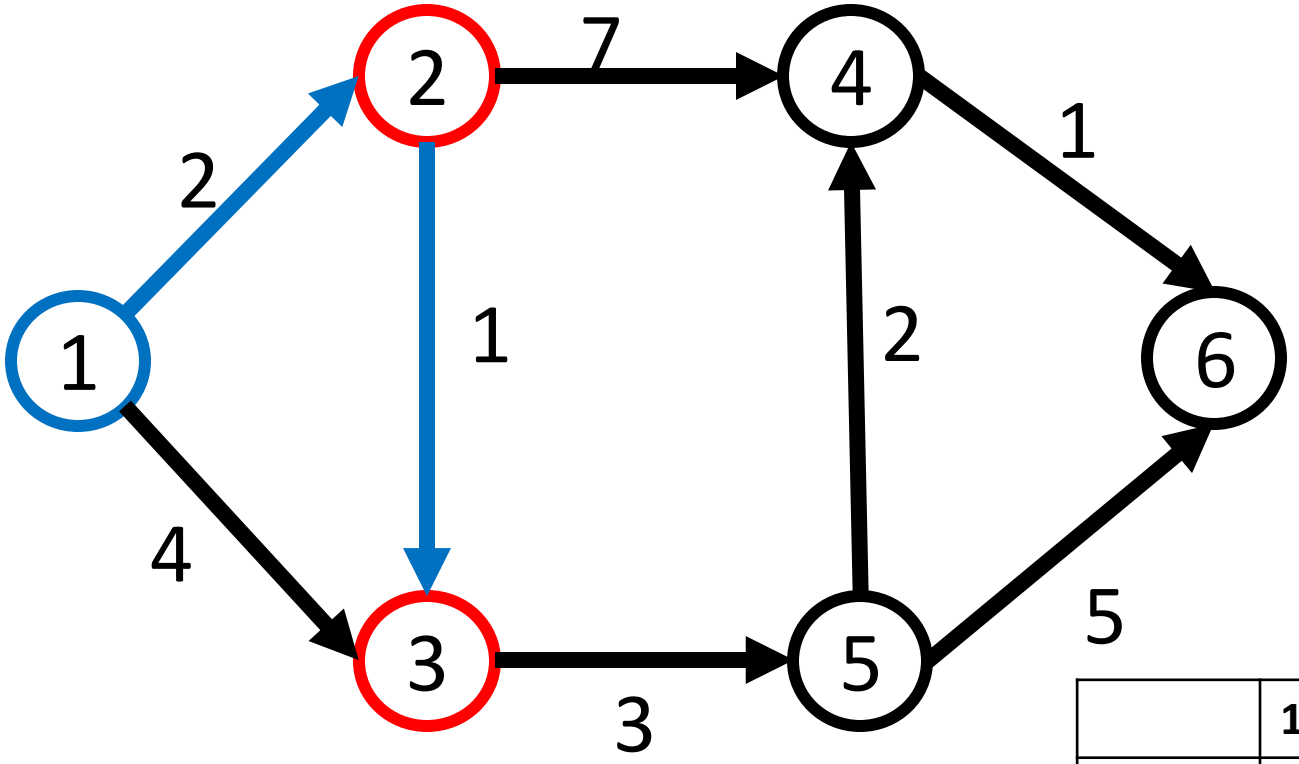
	1	2	3	4	5	6
Initial	**0**	**2**	**4**	**∞**	**∞**	**∞**
	0	**2**	~~**4**~~ **3**	**9**	**∞**	**∞**



Unvisited Nodes: {1, 2, 3, 4, 5, 6,}

	1	2	3	4	5	6
Initial	0	2	4	∞	∞	∞
	0	2	~~4~~ 3	9	∞	∞

Next : pick the smallest edge of which the vertex has not been visited, node-3, then mark node-3 is visited



Unvisited Nodes: {1, 2, 3, 4, 5, 6,}

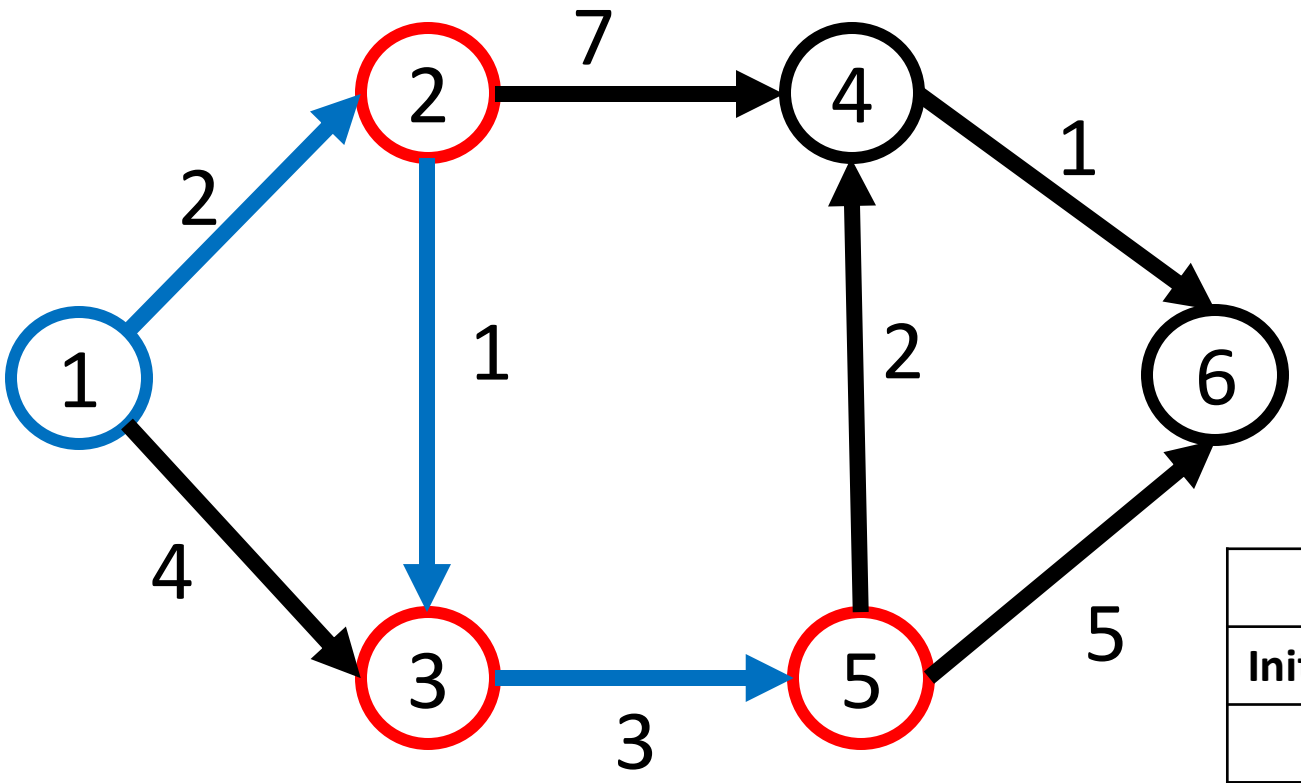
	1	2	3	4	5	6
Initial	**0**	**2**	**4**	**∞**	**∞**	**∞**
	0	**2**	**3**	**9**	**∞**	**∞**
	0	**2**	**3**	**9**	**6**	**∞**

Next Step: examine the edges leaving from node-3, and update distance

For each v adjacent to u do

d[v] ← min{d[v], d[u]+w(u,v)}

//update shortest paths for each neighboring vertex



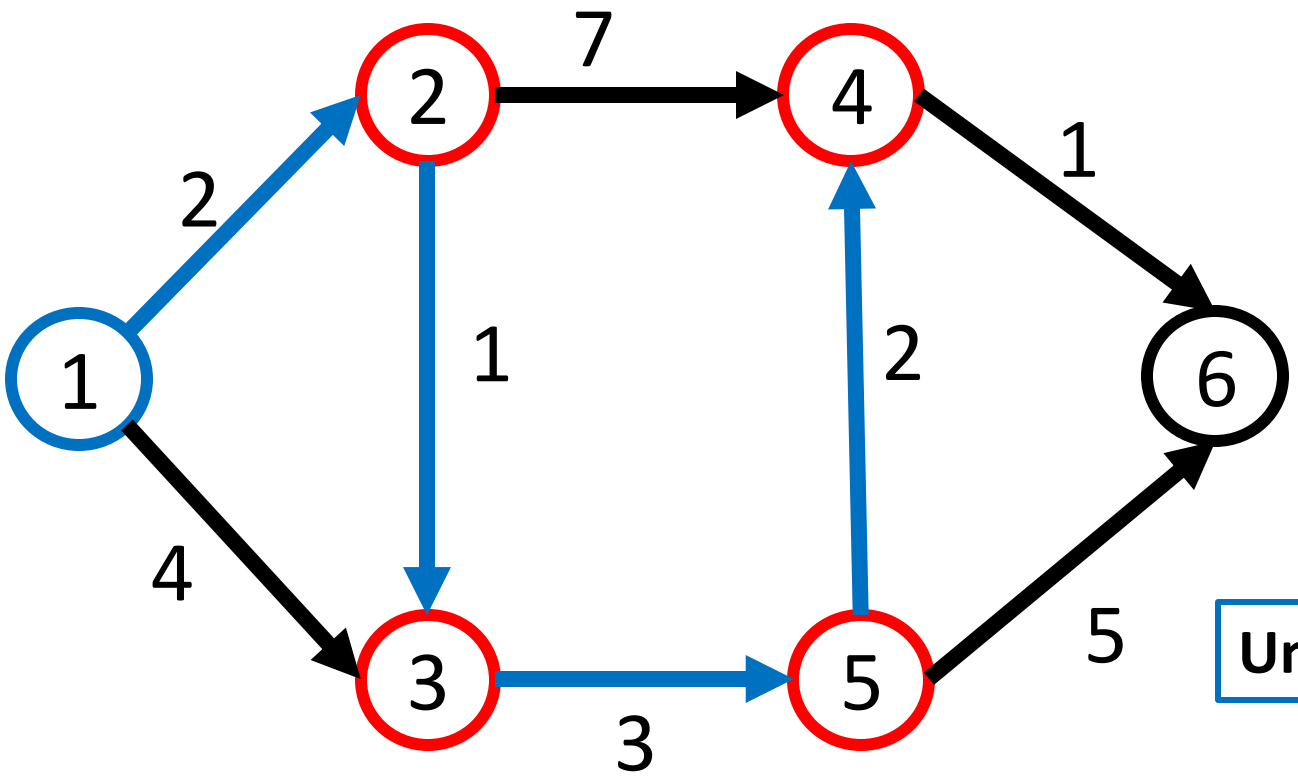
Unvisited Nodes: {1, 2, 3, 4, 5, 6,}

	1	2	3	4	5	6
Initial	**0**	**2**	**4**	**∞**	**∞**	**∞**
	0	**2**	**3**	**9**	**∞**	**∞**
	0	**2**	**3**	**9**	**6**	**∞**
	0	**2**	**3**	~~**9**~~ **8**	**6**	**11**

pick the smallest edge of which the vertex has not been visited, node-5, then mark node-5 is visited

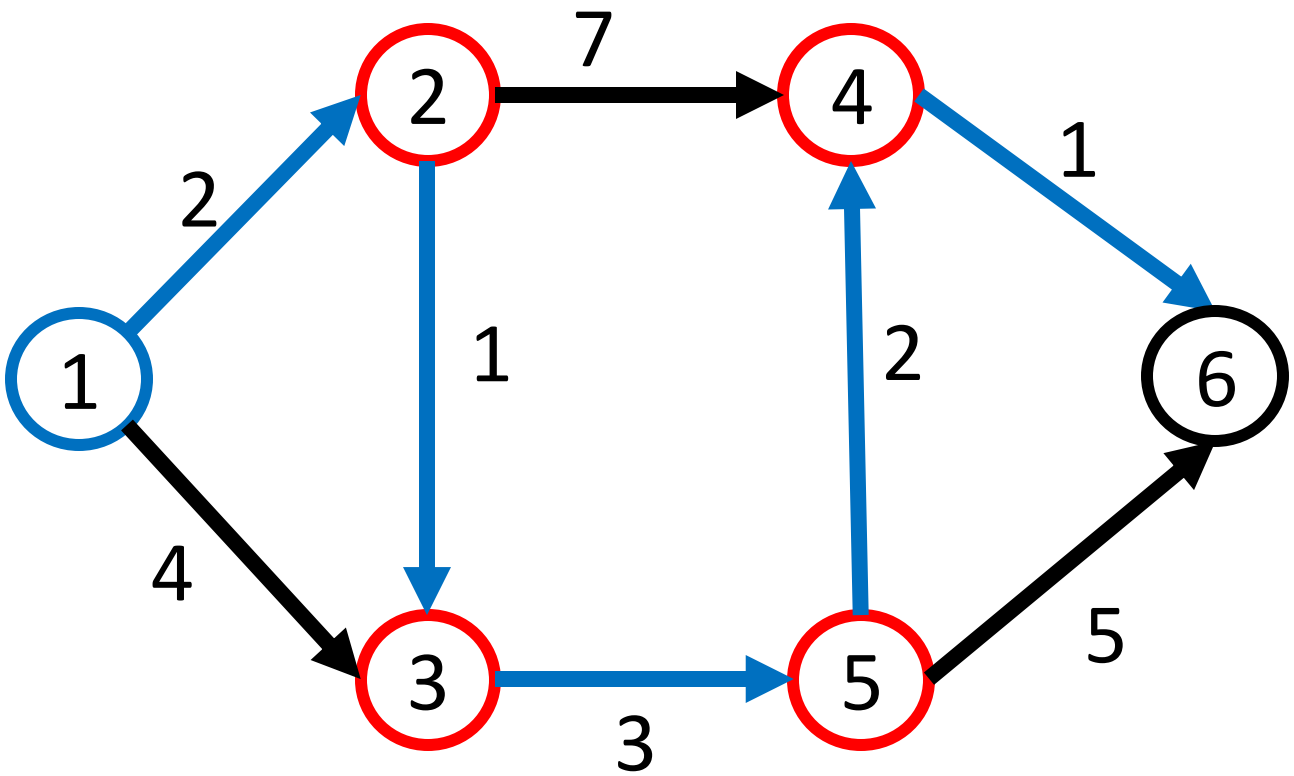
Next Step: examine the edges leaving from node-5, and update distance

For each v adjacent to u do
    d[v] ← min{d[v], d[u]+w(u,v)}
    //update shortest paths for each neighboring vertex



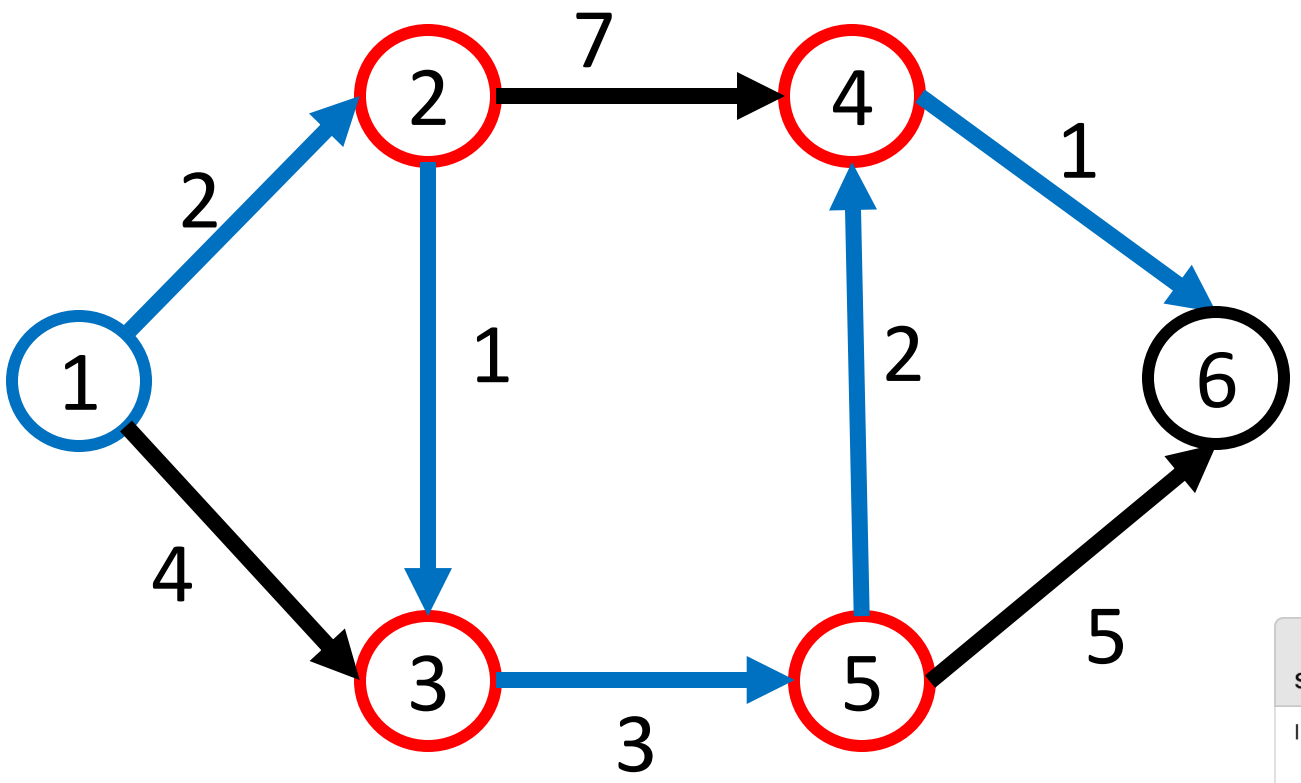
Unvisited Nodes: {1, 2, 3, 4, 5, 6,}

	1	2	3	4	5	6
Initial	0	2	4	∞	∞	∞
	0	2	3	9	∞	∞
	0	2	3	9	6	∞
	0	2	3	~~9~~ 8	6	11
	0	2	3	8	6	~~11~~ 9



v	d[v]
2	2
3	3
4	8
5	6
6	9

	1	2	3	4	5	6
Initial	**0**	**2**	**4**	**∞**	**∞**	**∞**
	0	**2**	**3**	**9**	**∞**	**∞**
	0	**2**	**3**	**9**	**6**	**∞**
	0	**2**	**3**	**~~9~~ 8**	**6**	**11**
	0	**2**	**3**	**8**	**6**	**~~11~~ 9**

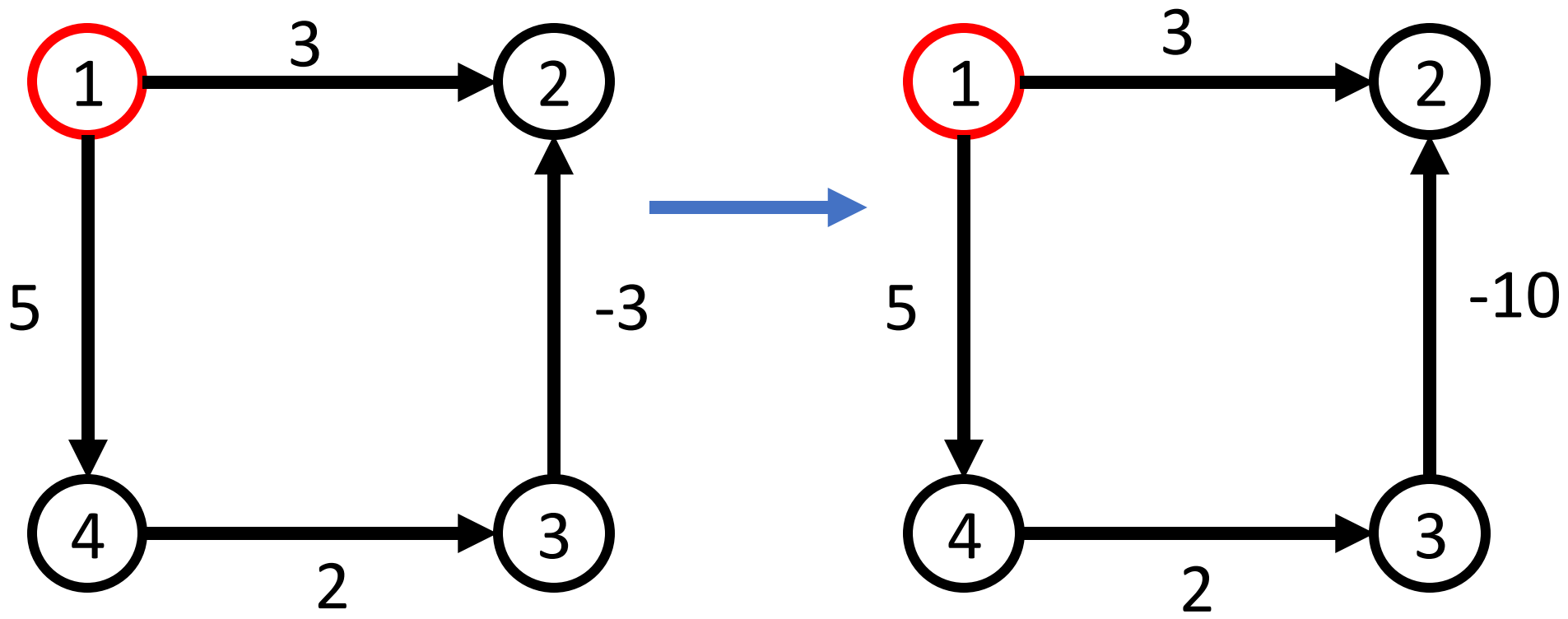


v	d[v]
2	2
3	3
4	8
5	6
6	9

Previous Nodes:

- $2:1, 3:2, 4:5, 5:3, 6:4$

Step	Node Processed	Tentative Distance	Previous Node
Initial	None	$1:0, 2:\infty, 3:\infty, 4:\infty, 5:\infty, 6:\infty$	$1:-, 2:-, 3:-, 4:-, 5:-, 6:-$
1	1	$1:0, 2:2, 3:4, 4:\infty, 5:\infty, 6:\infty$	$1:-, 2:1, 3:1, 4:-, 5:-, 6:-$
2	2	$1:0, 2:2, 3:3, 4:9, 5:\infty, 6:\infty$	$1:-, 2:1, 3:2, 4:2, 5:-, 6:-$
3	3	$1:0, 2:2, 3:3, 4:9, 5:6, 6:\infty$	$1:-, 2:1, 3:2, 4:2, 5:3, 6:-$
4	5	$1:0, 2:2, 3:3, 4:8, 5:6, 6:11$	$1:-, 2:1, 3:2, 4:5, 5:3, 6:5$
5	4	$1:0, 2:2, 3:3, 4:8, 5:6, 6:9$	$1:-, 2:1, 3:2, 4:5, 5:3, 6:4$


1
3
2
5
-3
4
2
3
1
3
2
5
-10
4
2
3

Bellman-Ford vs. Dijkstra

- Edge Weight:
  - Bellman-Ford can detect negative-weight cycle.
  - Dijkstra only deals with non-negative edge weights.
- Key Idea:
  - Bellman-Ford is based on BFS.
  - Dijkstra resembles both BFS and Prim's algorithm
- Running Time:
  - Bellman-Ford: $O(|V||E|)$
  - Dijkstra: $O(|V|^2)$, but is further reduced to $O(|E|+|V|\log(|V|))$ in an improved variant

All-Pair Shortest Paths

Shortest-Paths Problems

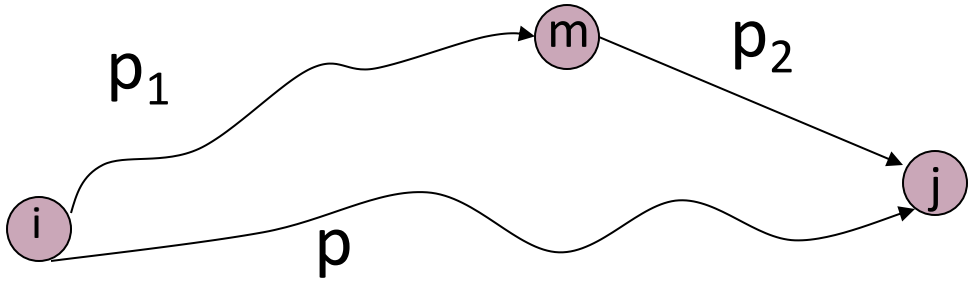
- Graph G(V, E)
  - V = $\{v_0, v_1, v_2, \ldots, v_n\}$: set of vertices
  - $w(v_i, v_j)$: weight of edge $e(v_i, v_j)$
  - $p(v_0, v_1, v_2, \ldots, v_k)$: a path from v_0 to v_k
  - $w(p) = w(v_0, v_1) + w(v_1, v_2) + \ldots + w(v_{k-1}, v_k)$: weight of path p
- Problems:
  - Single-Source Shortest-Paths: find a shortest path **from a source s to every vertex v in V**.
  - All-Pair Shortest Paths: find a shortest path **between every pair of nodes in graph G**.

Intuitive Methods

- For each vertex in the graph, run Dijkstra algorithm
- Totally, Dijkstra algorithm needs to be performed |V| times
- In Dijkstra, each node is checked to update shortest paths
- Can we do it in another way?

Intermediate Node

- If m is not an intermediate vertex of path p, i.e., there are m-1 intermediate vertices on path p, then all intermediate vertices of path p are in the set {1, 2, ..., m-1}.
- If m is an intermediate vertex of path p, p_1 is a shortest path from i to m with all intermediate vertices in the set {1, 2, ..., m-1}. Similarly, p_2 is a shortest path from vertex m to vertex j with all intermediate vertices in the set {1, 2, ..., m-1} .

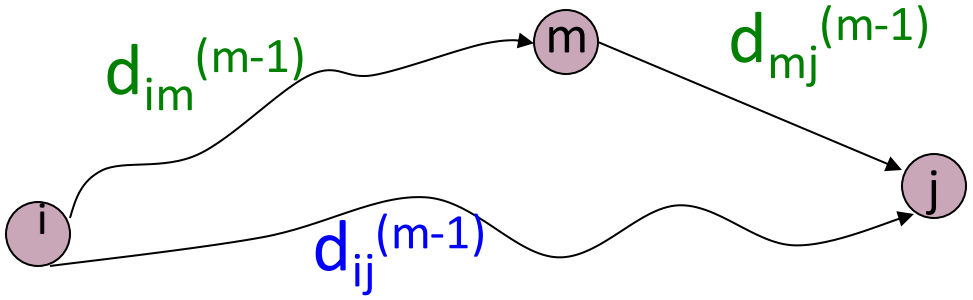


intermediate nodes in {1, 2, ..., m}

Floyd-Warshall Algorithm

- Dynamic Programming (DP) –based method
- $d_{ij}^{(m)}$ = length of a shortest path from i to j with intermediate vertices in the set {1, 2, …, m}
- DP: compute $d_{ij}^{(m)}$ in terms of smaller $d_{ij}^{(m-1)}$
  - $d_{ij}^{(0)} = w_{ij}$ //there is no intermediate vertex
  - $d_{ij}^{(m)} = \min \{d_{ij}^{(m-1)}, d_{im}^{(m-1)} + d_{mj}^{(m-1)}\}$

    //m is a new intermediate vertex or not



intermediate nodes in {1, 2, …, m}

Floyd-Warshall Algorithm

- Difference from previous: we do not check all possible intermediate vertices.
- Implementation

```
for m=1… |V| do
    for i=1… |V| do
        for j = 1… |V| do
```

$$d_{ij}^{(m)} = \min \{d_{ij}^{(m-1)}, d_{im}^{(m-1)} + d_{mj}^{(m-1)}\}$$

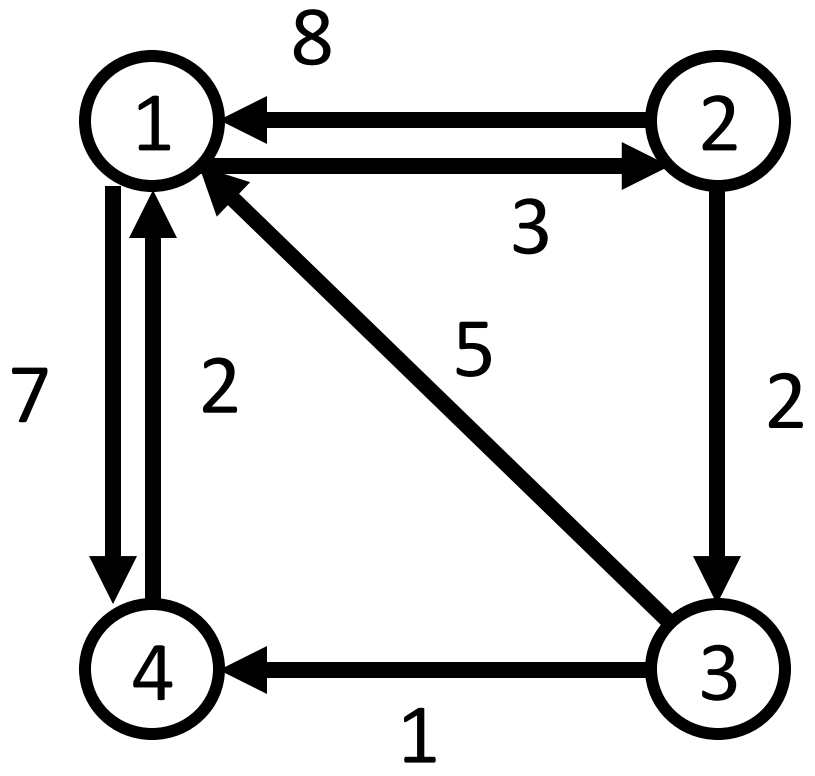
- Runtime: $O(|V|^3)$

Example

Initially, $d_{ij}^{(0)} = w_{ij}$, because no intermediate node is selected.

A0

From\To	1	2	3	4
1	0	3	∞	7
2	8	0	2	∞
3	5	∞	0	1
4	2	∞	∞	0



Example

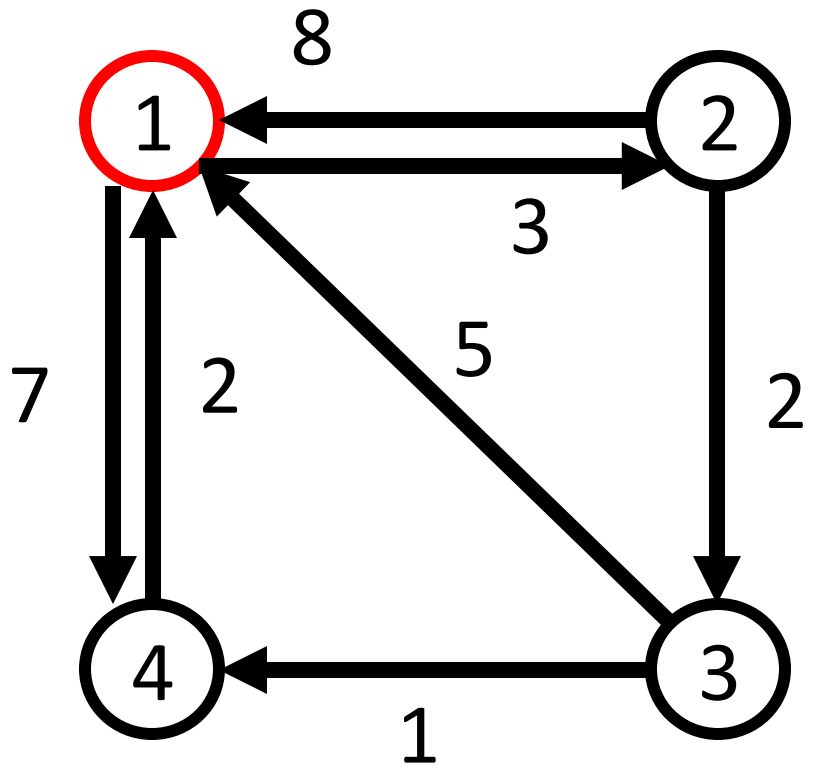
Initially, $d_{ij}^{(0)} = w_{ij}$, because no intermediate node is selected.

A0

From\To	1	2	3	4
1	0	3	∞	7
2	8	0	2	∞
3	5	∞	0	1
4	2	∞	∞	0

A1

	1	2	3	4
1	0	3	∞	7
2	8	0	2	15
3	5	8	0	1
4	2	5	∞	0



Example

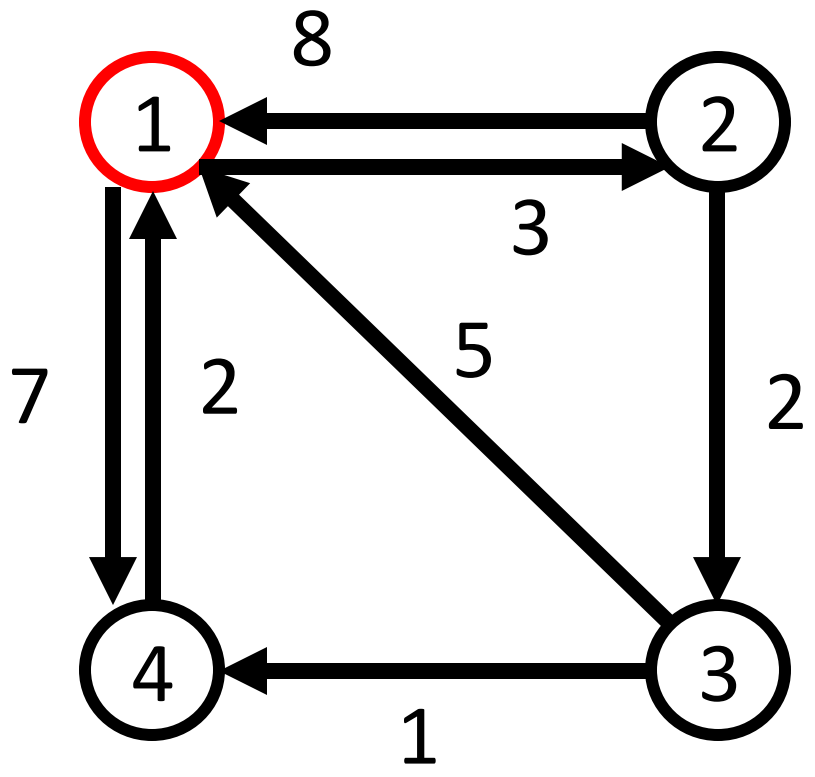
Initially, $d_{ij}^{(0)} = w_{ij}$, because no intermediate node is selected.

A1

	1	2	3	4
1	0	3	∞	7
2	8	0	2	15
3	5	8	0	1
4	2	5	∞	0

A2

	1	2	3	4
1	0	3	5	7
2	8	0	2	15
3	5	8	0	1
4	2	5	∞	0



Example

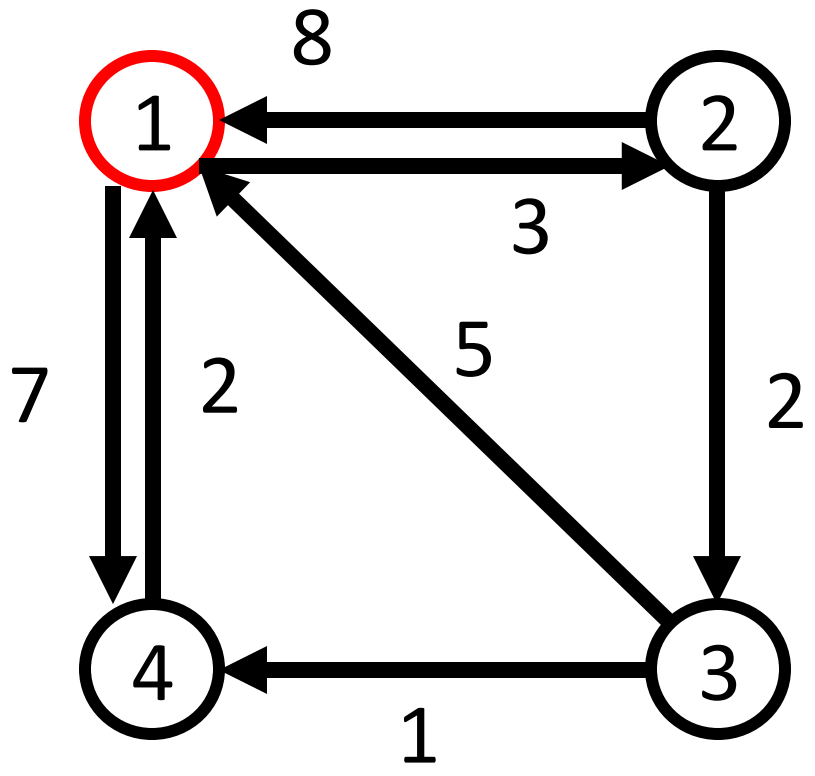
Initially, $d_{ij}^{(0)} = w_{ij}$, because no intermediate node is selected.

A2

	1	2	3	4
1	0	3	5	7
2	8	0	2	15
3	5	8	0	1
4	2	5	∞	0

A3

	1	2	3	4
1	0	3	5	6
2	7	0	2	3
3	5	8	0	1
4	2	5	∞	0



Example

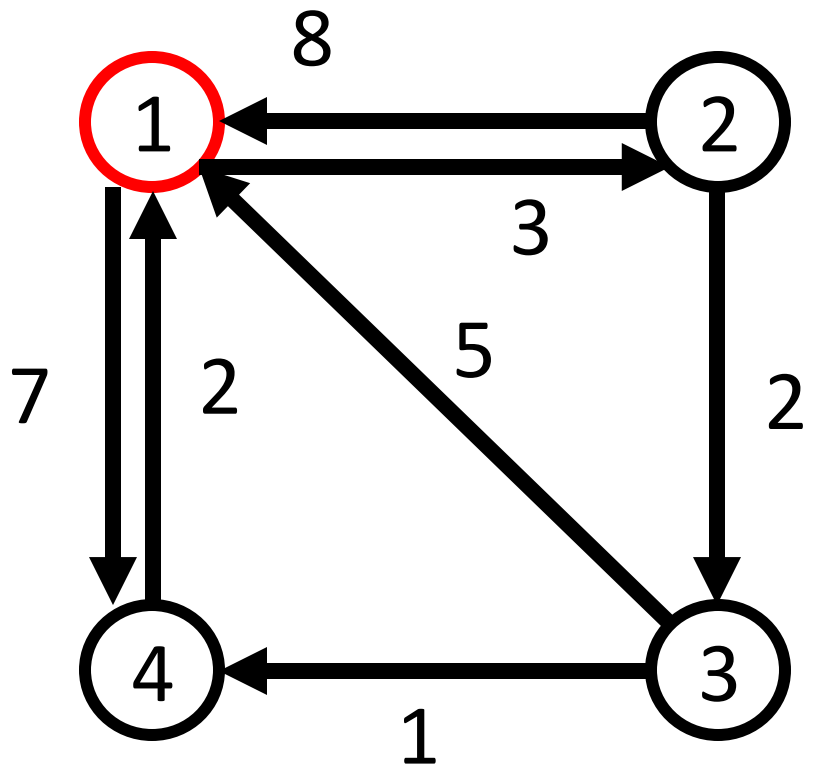
Initially, $d_{ij}^{(0)} = w_{ij}$, because no intermediate node is selected.

A3

	1	2	3	4
1	0	3	5	6
2	7	0	2	3
3	5	8	0	1
4	2	5	∞	0

A4

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



Example

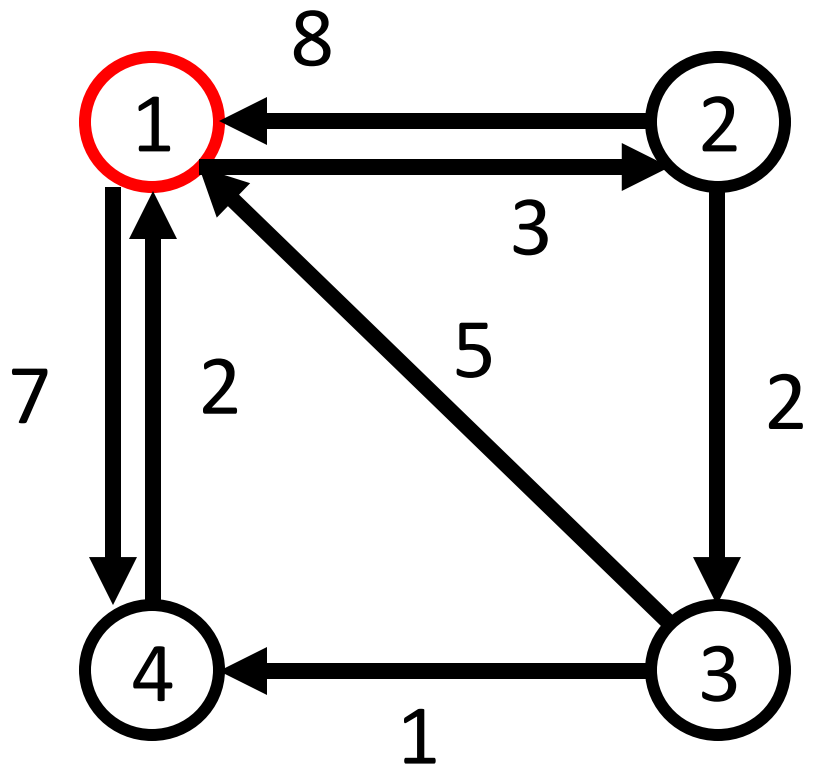
Initially, $d_{ij}^{(0)} = w_{ij}$, because no intermediate node is selected.

A0

	1	2	3	4
1	0	3	∞	7
2	8	0	2	∞
3	5	∞	0	1
4	2	∞	∞	0

A4

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



Example

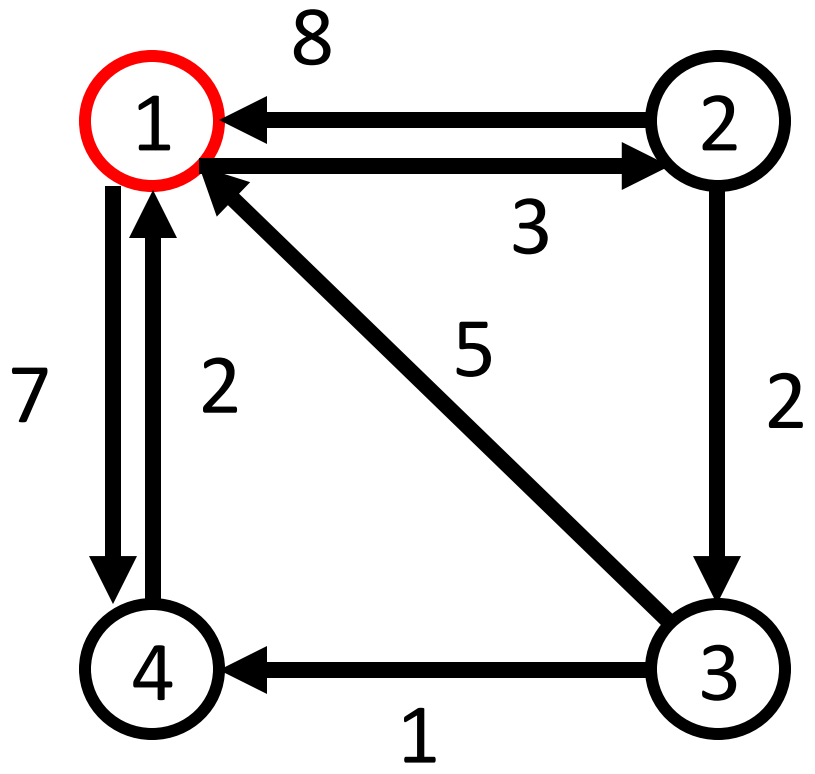
Initially, $d_{ij}^{(0)} = w_{ij}$, because no intermediate node is selected.

A0

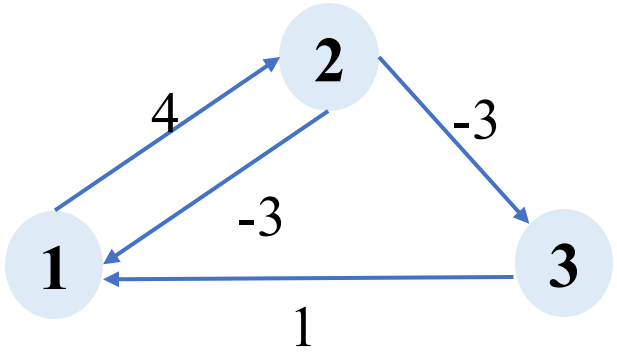
	1	2	3	4
1	0	3	∞	7
2	8	0	2	∞
3	5	∞	0	1
4	2	∞	∞	0

A4

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



Floyd-Warshall



FLOYD-WARSHALL(W)

1 $n \leftarrow rows[W]$
2 $D^{(0)} \leftarrow W$
3 **for** $k \leftarrow 1$ **to** n
4 **do for** $i \leftarrow 1$ **to** n
5 **do for** $j \leftarrow 1$ **to** n
6 **do** $d_{ij}^{(k)} \leftarrow \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right)$
7 **return** $D^{(n)}$

G contains a negative cycle if and only if $D^{(n)}(G)$ contains a negative diagonal element.

$$D^{(0)} = \begin{pmatrix} 0 & 4 & \infty \\ -3 & 0 & -3 \\ 1 & \infty & 0 \end{pmatrix}$$

$$D^{(1)} = \begin{pmatrix} 0 & 4 & \infty \\ -3 & 0 & -3 \\ 1 & 5 & 0 \end{pmatrix}$$

$$D^{(2)} = \begin{pmatrix} 0 & 4 & 1 \\ -3 & 0 & -3 \\ 1 & 5 & 0 \end{pmatrix}$$

$$D^{(3)} = \begin{pmatrix} 0 & 4 & 1 \\ -3 & 0 & -3 \\ 1 & 5 & 0 \end{pmatrix}$$

$D^{(3)}$ does not have negative diagonal element.
So there is no negative cycle in graph G.