TRƯỜNG ĐẠI HỌC BÁCH KHOA HÀ NỘI
VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

Discrete Mathematics

Nguyễn Khánh Phương

Department of Computer Science
School of Information and Communication Technology
E-mail: phuongnk@soict.hust.edu.vn

PART 1
COMBINATORIAL THEORY
(Lý thuyết tổ hợp)

PART 2
GRAPH THEORY
(Lý thuyết đồ thị)

Content of Part 2

NGUYỄN KHÁNH PHƯƠNG
Bộ môn KHMT – ĐHBK HN

TRƯỜNG ĐẠI HỌC BÁCH KHOA HÀ NỘI
VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

Chapter 5

Shortest path problem

NGUYỄN KHÁNH PHƯƠNG
Bộ môn KHMT – ĐHBK HN

Content

1. **Shortest path problem**
2. Shortest path properties, Reduce upper bound
3. Bellman-Ford algorithm
4. Dijkstra algorithm
5. Shortest path in acyclic graph
6. Floyd-Warshall algorithm

1. Shortest Path

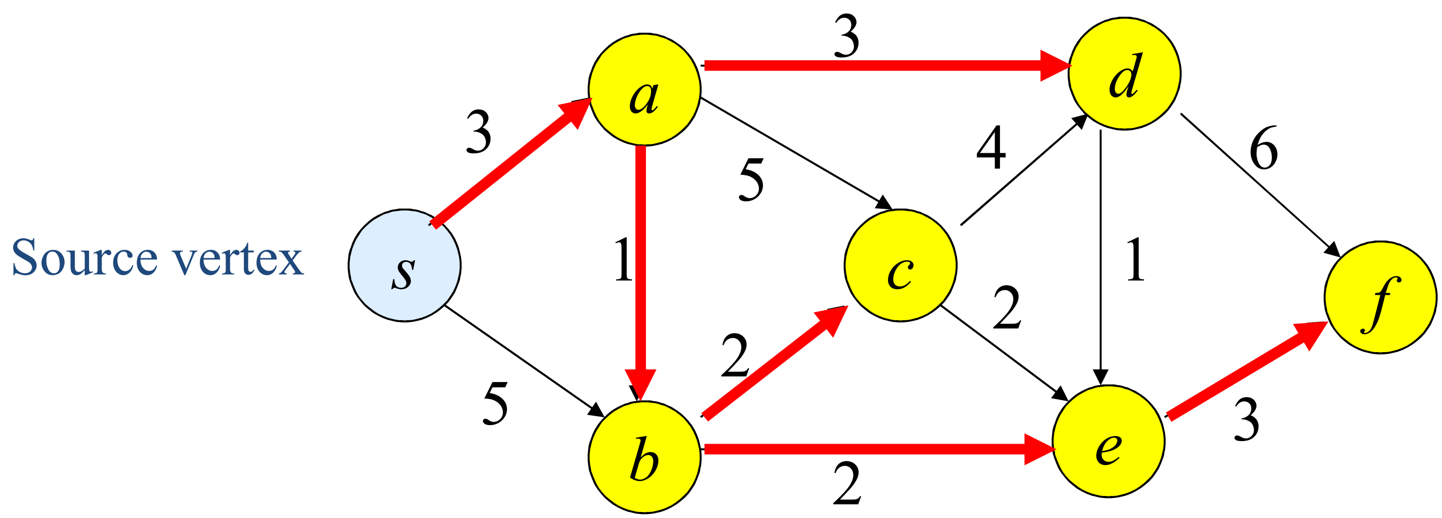
- Generalize distance to weighted setting
- Digraph $G = (V,E)$ with weight function $W: E \rightarrow R$ (assigning real values to edges)
- Weight of path $p = v_1 \xrightarrow{w(v_1,v_2)} v_2 \xrightarrow{w(v_2,v_3)} \ldots \xrightarrow{w(v_{k-1},v_k)} v_k$ is $\quad w(p) = \sum_{i=1}^{k-1} w(v_i, v_{i+1})$
- Shortest path = a path of the minimum weight

$$\delta(u,v) = \begin{cases} \min\{\omega(p) : u \overset{p}{\rightsquigarrow} v\}; & \text{if there is a path from } u \text{ to } v, \\ \infty & \text{otherwise.} \end{cases}$$

- Applications
  - static/dynamic network routing
  - robot motion planning
  - map/route generation in traffic
  - speech interpretation (best interpretation of a spoken sentence)
  - medical imaging

Example

Graph $G = (V, E)$, source vertex $s \in V$, find the shortest path from s to each of remaining vertices.



	s	a	b	c	d	e	f
path	s	s,a	s,a,b	s,a,b,c	s,a,d	s,a,b,e	s,a,b,e,f
weight	0	3	4	6	6	6	9

Shortest-Path Variants

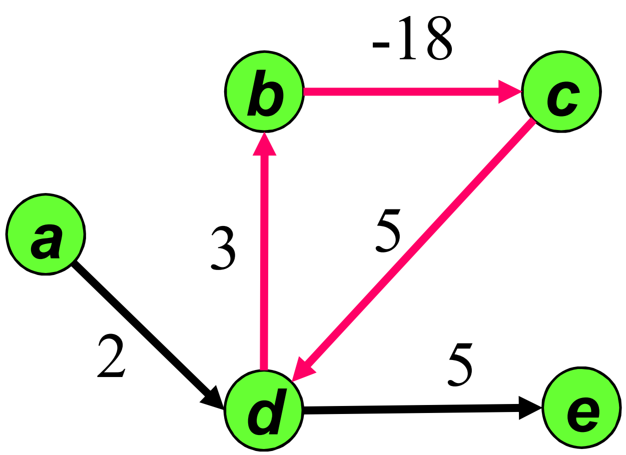
- **Single-source shortest-paths problem**
  - Find a shortest path from a given source (vertex s) to each of the vertices.
- **Single-destination shortest-paths problem**
  - Find a shortest path to a given *destination* vertex t from each vertex v.
- **Single-pair shortest-path problem**
  - Given two vertices, find a shortest path between them. Solution to single-source problem solves this problem efficiently, too.
- **All-pairs shortest-paths problem**
  - Find a shortest path from u to v for every pair of vertices u and v

Comment:

- The problems are arranged in order from simple to complex
- Whenever there is an efficient algorithm for solving one of the three problems, the algorithm can also be used to solve the remaining two problems.

Negative Weights and Cycles?

- Negative edges are OK.
- *Negative weight cycles:* NO (otherwise paths with arbitrary small "lengths" would be possible)



Cycle: $(d \to b \to c \to d)$

Length = -10

Path from a to e:

P: $a \to \sigma(d \to b \to c \to d) \to e$

$w(P) = 7\text{-}10\sigma \to -\infty$, khi $\sigma \to +\infty$

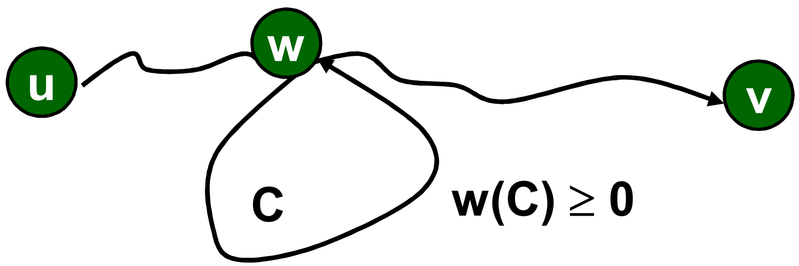
Assumption:

Graph does not contain negative weight cycles

Properties of shortest paths

- ***Property* 1.** Shortest-paths can have no cycles (= The shortest path can always be found among single paths). Path where vertices are distinct.

Proof: Removing a cycle with positive length could reduce the length of the path.



- ***Property 2.*** Any shortest-path in graph *G* can not traverse through more than n – 1 edges, where n is the number of vertices
  - *Consequence of Property 1.*

>= n edges → not the simple path

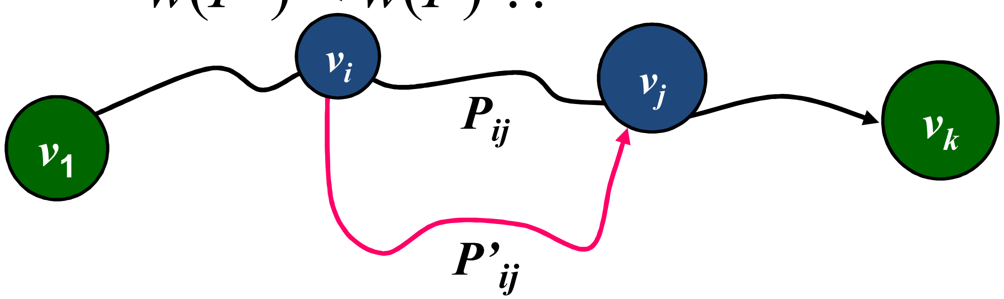
Properties of shortest paths

Property 3: Assume $P = \langle v_1, v_2, \ldots, v_k \rangle$ is the shortest path from v_1 to v_k. Then, $P_{ij} = \langle v_i, v_{i+1}, \ldots, v_j \rangle$ is the shortest path from v_i to v_j, where $1 \leq i \leq j \leq k$.

(In words: subpaths of shortest paths are also shortest paths)

Proof (by contradiction). If P_{ij} is not the shortest path from v_i to v_j, then one can find P'_{ij} is the shortest path from v_i to v_j satisfying $w(P'_{ij}) < w(P_{ij})$. Then we get P' is the path obtained from P by substituing P_{ij} by P'_{ij}, thus:

$w(P') < w(P)$?!



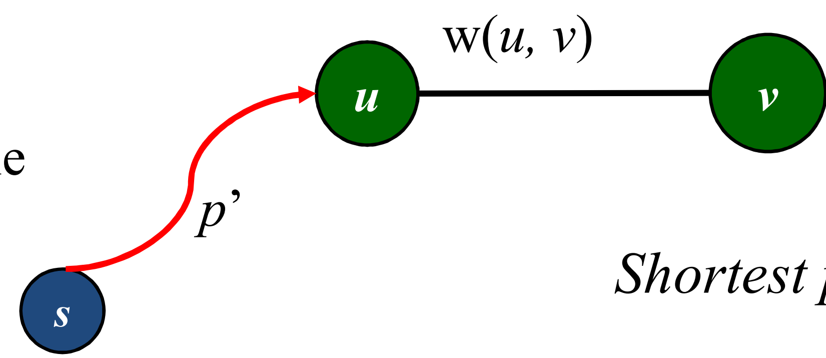
if some subpaths were not the shortest paths, one could substitute the shorter subpath and create a shorter total path

Properties of shortest paths

Denote: $\delta(u, v)$ = length of the shortest path from u to v

Assume P is the shortest path from s to v, where $P = s \ldots \xrightarrow{p'} .. u \to v$.
Then $\delta(s, v) = \delta(s, u) + w(u, v)$.

$\delta(s, u)$:length of the shortest path from s to v



Shortest path from s to v:

$P = s \ldots \xrightarrow{p'} .. \quad u \to v$

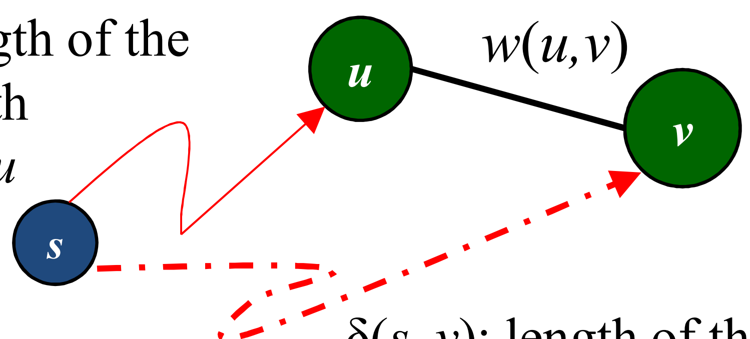
Properties of shortest paths

Denote: $\delta(u, v)$ = length of the shortest path from u to v

Assume P is the shortest path from s to v, where $P = s \ldots \xrightarrow{p'} \ldots u \to v$.
Then $\delta(s, v) = \delta(s, u) + w(u, v)$.

<u>Property 4:</u> Assume $s \in V$. For each edge $(u,v) \in E$, we have $\delta(s, v) \leq \delta(s, u) + w(u,v)$.

$\delta(s, u)$: length of the shortest path from s ton u



$\delta(s, v)$: length of the shortest path from s to v

~~If $\delta(s, v) > \delta(s, u) + w(u,v)$???~~

If $\delta(s, v) < \delta(s, u) + w(u,v)$???

If $\delta(s, v) = \delta(s, u) + w(u,v)$???

Shortest-Path Variants

- **Single-source shortest-paths problem**
  – Find a shortest path from a given source (vertex *s*) to each of the vertices.
- **Single-destination shortest-paths problem**
  – Find a shortest path to a given *destination* vertex *t* from each vertex *v*.
- **Single-pair shortest-path problem**
  – Given two vertices, find a shortest path between them. Solution to single-source problem solves this problem efficiently, too.
- **All-pairs shortest-paths problem**
  – Find a shortest path from *u* to *v* for every pair of vertices *u* and *v*

Comment:

- The problems are arranged in order from simple to complex
- Whenever there is an efficient algorithm for solving one of the three problems, the algorithm can also be used to solve the remaining two problems.

TRƯỜNG ĐẠI HỌC BÁCH KHOA HÀ NỘI
VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

Single-source shortest paths

Shortest path representation

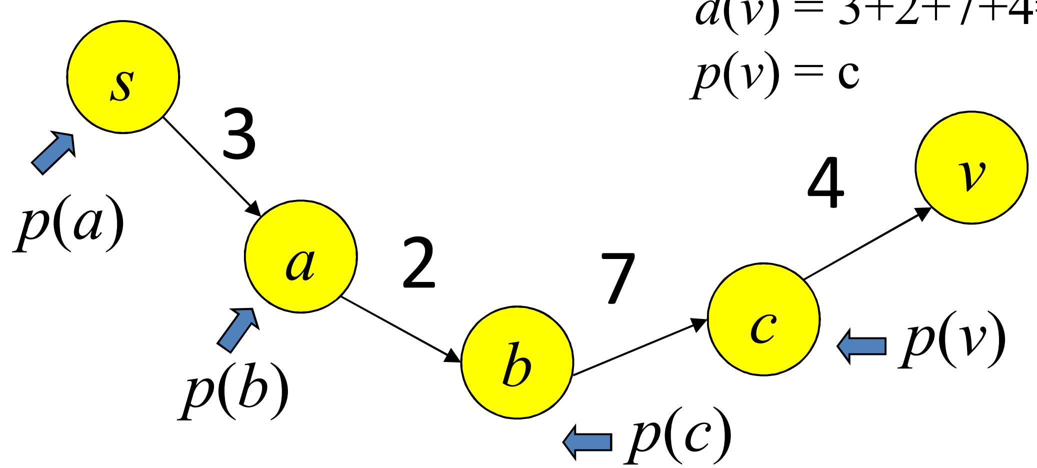
Shortest path algorithms works on 2 arrays:

- $d(v)$ = the length of shortest path from s to v that algorithm found so far
  (upper bound for the length of the shortest path from s to v).
- $p(v)$ = a predecessor of v in this shortest path
  (used to back trace the path from s to v) .

$\delta(s,v) \le d(v)$

Initialization

for $v \in V(G)$
    do $d[v] \leftarrow \infty$
        $p[v] \leftarrow$ NULL
$d[s] \leftarrow 0$

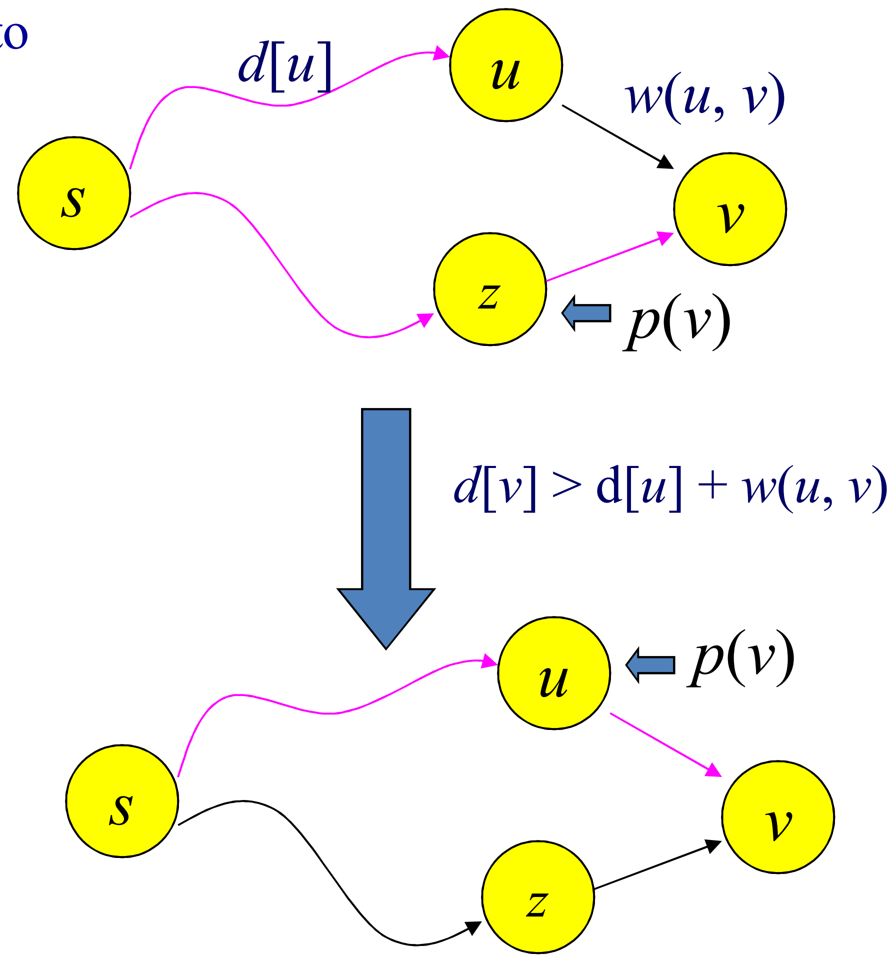


Relaxation

Building the shortest path from s to v:

Assume the current known shortest path from s to v: $\mathbf{s \dots z \rightarrow v}$

Relaxing an edge (u,v) means testing whether we can improve the shortest path to v found so far by going through u



Relax(u, v)
  if ($d[v] > d[u] + w(u, v)$)
  { $d[v] \leftarrow d[u] + w(u, v)$
    $p[v] \leftarrow u$
  }

Properties of Relaxation

Relax(u, v)
    if ($d[v] > d[u] + w(u, v)$)
    {
        $d[v] \leftarrow d[u] + w(u, v)$
        $p[v] \leftarrow u$
    }

Shortest path algorithms differ in

- *how many times* they relax each edge, and
- *the order* in which they relax edges

Single source shortest path

1. **Bellman-Ford algorithm**
2. Dijkstra algorithm

Bellman-Ford algorithm

Richard Bellman
1920-1984

Lester R. Ford, Jr.
1927-2017

Bellman-Ford algorithm

Bellman-Ford algorithm is used to find the shortest path from a vertex *s* to each other vertex in the graph.

- **Input:** A directed graph $G=(V,E)$ and weight matrix $w[u,v] \in R$ where $u,v \in V$, source vertex $s \in V$;
  >= 0
  <0
  – *G does not contain* negative-weight cycle

- **Output:** Each $v \in V$
  $d[v] = \boxed{\delta(s, v);}$ Length of the shortest path from *s* to *v*
  $p[v]$ - the predecessor of *v* in this shortest path from *s to v.*

Bellman-Ford algorithm: Full version

```
Bellman-Ford(G, w, s)
// Step 1: Initialize shortest paths of with at most 0 edges
    Initialize-Single-Source(G, s)
/* Step 2: Calculate shortest paths with at most i edges from shortest
   paths with at most i-1 edges */
    for i in range (1, |V|)
      for each edge (u, v) ∈ E
           Relax(u, v)
    for each edge (u, v) ∈ E
      if d[v] > d[u] + w(u, v)
           return False    // there is a negative cycle
    return True
```

```
Relax(u, v)
 if d[v] > d[u] + w(u, v)
 {
     d[v] = d[u] + w[u,v] ;
     p[v] = u ;
 }
```

```
Initialize-Single-Source(G, s)
for v ∈ V\s
{
     d[v] = ∞;
     p[v] = Null;
}
p[s]=Null; d[s]=0;
```

Bellman-Ford algorithm: Full version

```
Bellman-Ford(G, w, s)
// Step 1: Initialize shortest paths of with at most 0 edges
1.  Initialize-Single-Source(G, s)
/* Step 2: Calculate shortest paths with at most i edges from shortest
   paths with at most i-1 edges */
2.  for i in range (1, |V|)
3.    for each edge (u, v) ∈ E
4.        Relax(u, v)
5.  for each edge (u, v) ∈ E
6.    if d[v] > d[u] + w(u, v)
7.        return False    // there is a negative cycle
8.  return True
```

O(|V|)

O(|V||E|)

Lines (2-4): First nested for-loop performs |V|-1 relaxation iterations; relax every edge at each iteration ➔ Running time $O(|V||E|)$

O(|E|)

- Running time $O(|V||E|)$
- Memmory space: $O(|V|^2)$

Bellman-Ford algorithm

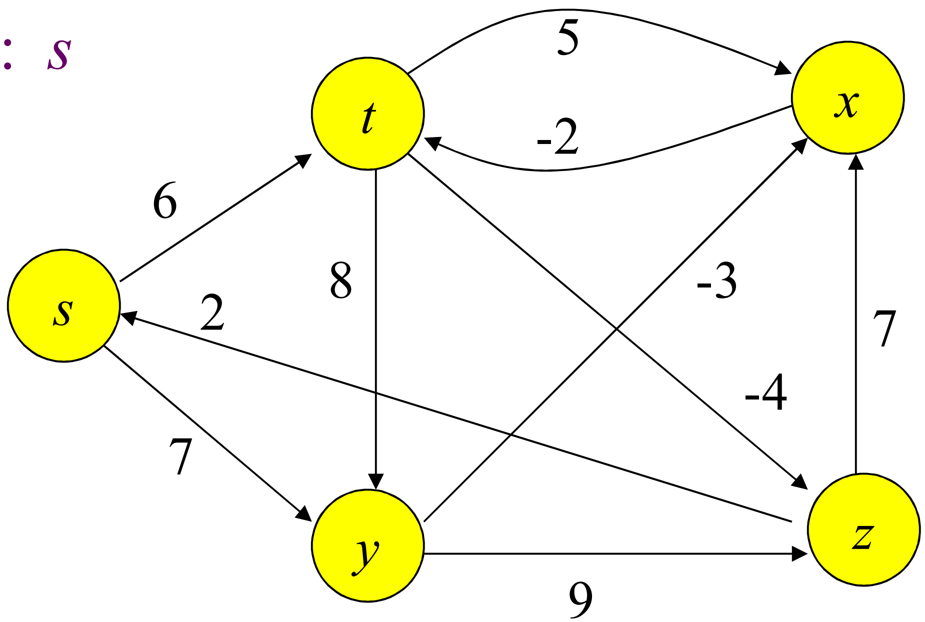
```
Bellman-Ford(G, w, s)
Initialize-Single-Source(G, s)                      O(|V|)
for i in range (1,|V|)                              O(|V||E|)
  for each edge (u, v) ∈ E
      Relax(u, v)
for each edge (u, v) ∈ E                            O(|E|)
  if d[v] > d[u] + w(u, v)
      return False   // there is a negative cycle
return True
```

if Ford-Bellman has not converged after |V| - 1 iterations, then there cannot be a shortest path tree, so there must be a negative weight cycle.

Example

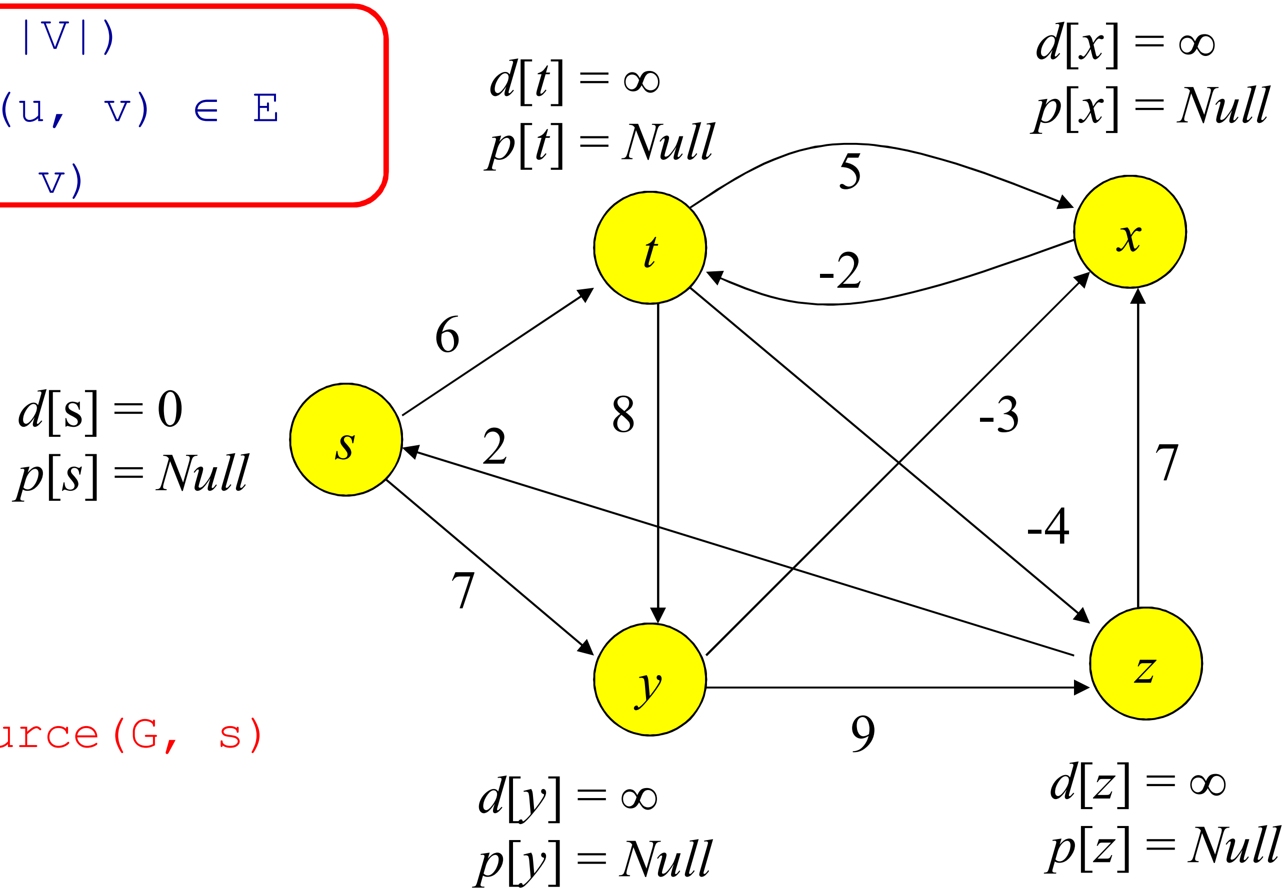
Apply Bellman-Ford algorithm to find the shortest path from *s* to all other vertices in the graph

Source: *s*



Initialize

```
Bellman-Ford(G, w, s)
// Step 1: Initialize shortest paths of with at most 0 edges
    Initialize-Single-Source(G, s)
    for i in range (1,|V|)
        for each edge (u, v) ∈ E
            Relax(u, v)
```




```
Initialize-Single-Source(G, s)
for v ∈ V\s
{
     d[v] = ∞;
     p[v] = Null;
}
p[s]=Null; d[s]=0;
```

$i = 1$

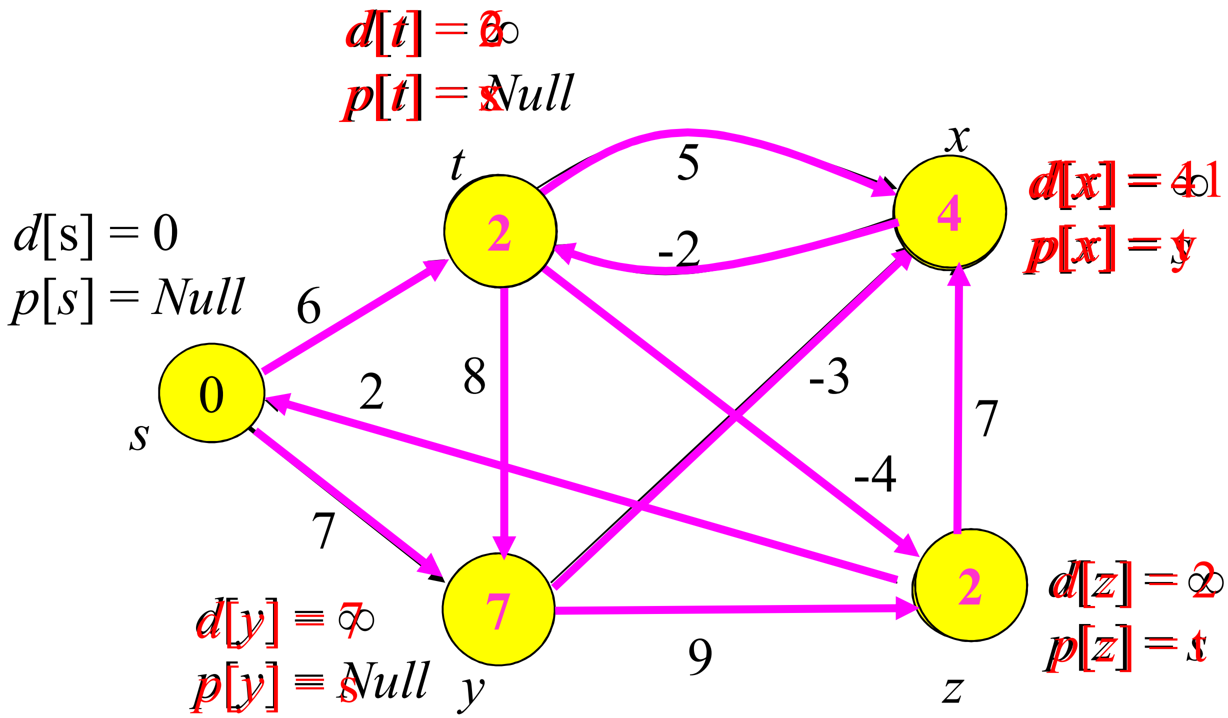
```
/* Step 2: Calculate shortest paths with at most i edges from shortest paths with at
   most i-1 edges */
for i in range (1, |V|)
    for each edge (u, v) ∈ E
            Relax(u, v)
```

Relax(u, v);

if $(d[v] > d[u] + w[u,v])$ {
    $d[v] = d[u] + w[u,v]$;
    $p[v] = u$;
}

The order of edges examined (call Relax):

(s, t), if $(\infty > 0+6)$
(s, y), if $(\infty > 0+7)$
(t, x), if $(\infty > 6+5)$
(t, y), if $(7 > 6+8)$
(t, z), if $(\infty > 6+(-4))$
(y, x), if $(11 > 7+(-3))$
(y, z), if $(2 > 7+9)$
(x, t), if $(6 > 4+(-2))$
(z, s), if $(0 > 2+2)$
(z, x), if $(4 > 2+7)$



$i = 2$

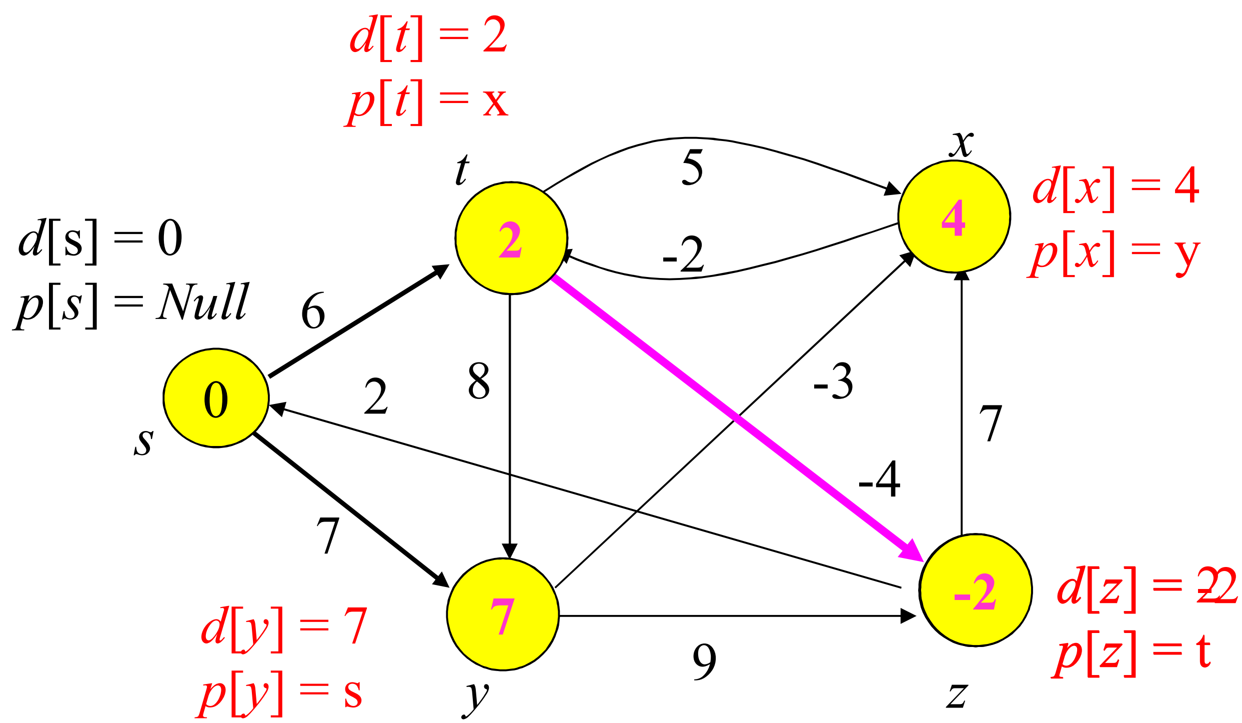
```
/* Step 2: Calculate shortest paths with at most i edges from shortest paths with at
   most i-1 edges */
2.  for i in range (1, |V|)
3.     for each edge (u, v) ∈ E
4.                Relax(u, v)
```

Relax(u, v);

if (d[v] > d[u] + w[u,v]) {
    d[v] = d[u] + w[u,v] ;
    p[v] = u ;
}

The order of edges examined (call Relax):

(s, t),
(s, y),
(t, x),
(t, y),
(t, z), if (2 > 2+(-4))
(y, x),
(y, z),
(x, t),
(z, s),
(z, x),



$i = 3$

```
/* Step 2: Calculate shortest paths with at most i edges from shortest paths with at
   most i-1 edges */
for i in range (1,|V|)
    for each edge (u, v) ∈ E
            Relax(u, v)
```

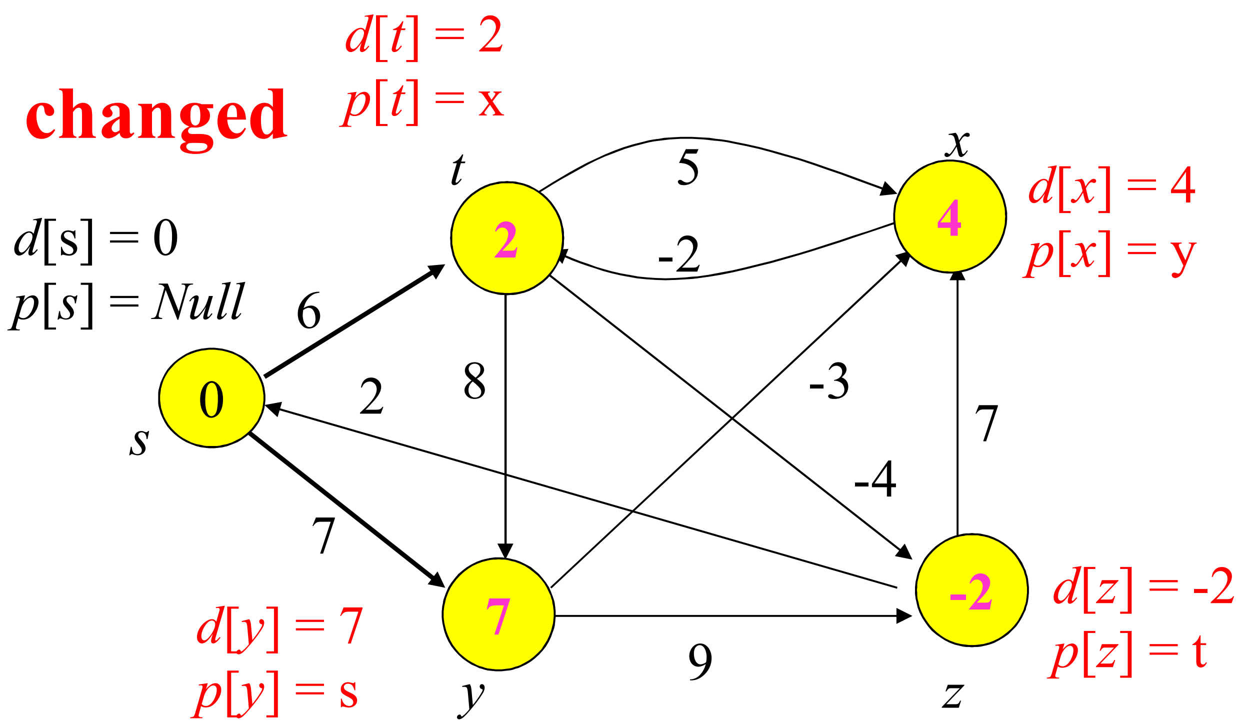
Relax(u, v);

```
if  (d[v] > d[u] + w[u,v]) {
     d[v] = d[u] + w[u,v] ;
      p[v] = u ;
}
```

The order of edges examined (call Relax):

(s, t),
(s, y),
(t, x),
(t, y),
(t, z),
(y, x),
(y, z),
(x, t),
(z, s),
(z, x),

None of d[u] changed



Comments

Bellman-Ford(G, w, s)

```
Initialize-Single-Source(G, s)
for i in range (1,|V|)
  for each edge (u, v) ∈ E
      Relax(u, v)
```

- Practical improvements:
  - No need to check edges of the form (u, v) unless $d[u]$ changed in previous iteration.
  - If there is not any d[u] value changed in iteration i → stop algorithm

  ➔ In this example: algorithm is stopped after 3 iterations as at 3rd iteration, there is not any $d[u]$ changed (|V| = 5 → need 4 iterations) .

  ➔ Improve algorithm speed.
- The values of d and p obtained at each iteration and the number of iterations have to be executed (algorithm speed) depends on the order of edges to be examined.
- The value of d of one vertex could be updated more than once at the same iteration.

Comments

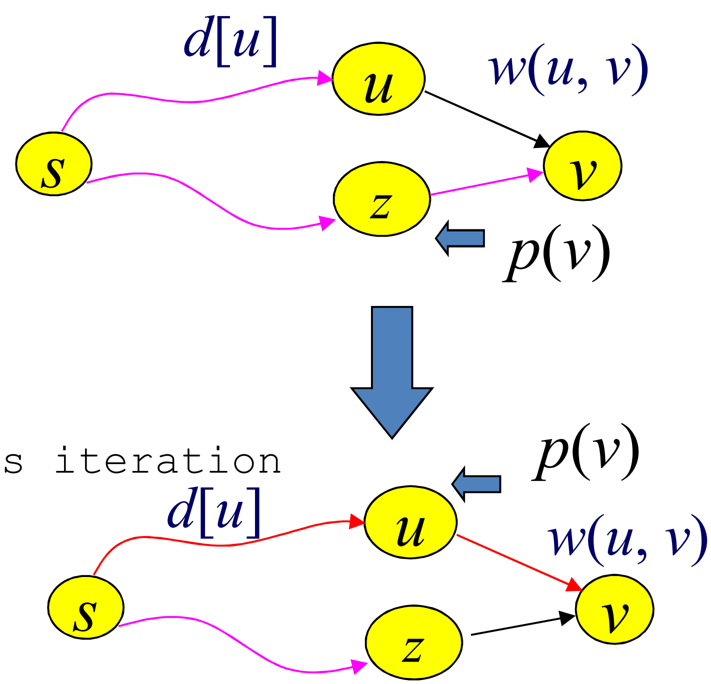
```
Bellman-Ford(G, w, s)
    Initialize-Single-Source(G, s)
    for i in range (1,|V|)
      for each edge (u, v) ∈ E
          Relax(u, v)
```

Practical improvements:

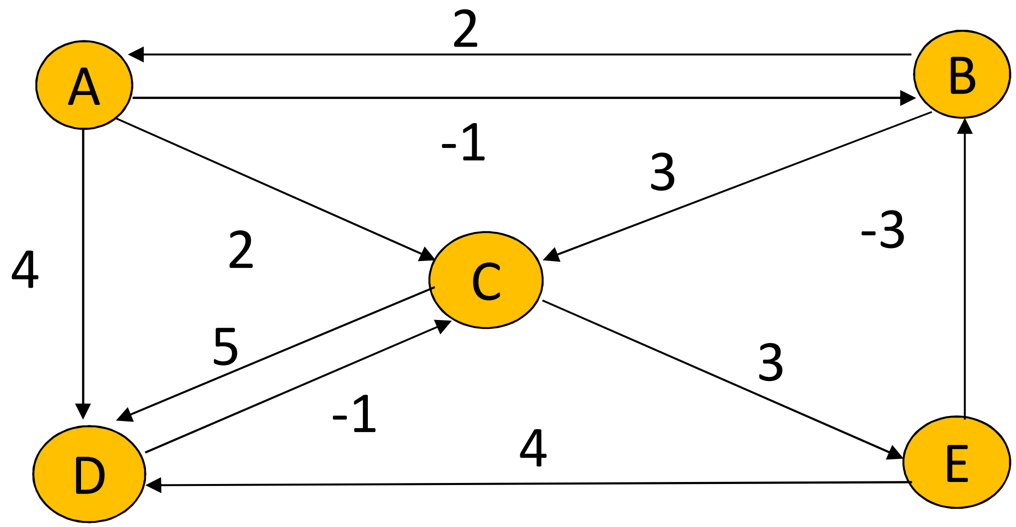
- No need to check edges of the form (u, v) unless $d[u]$ changed in previous iteration.
- If there is no d[u] value changed in iteration i → stop algorithm

Bellman-Ford efficient implementation:

```
Bellman-Ford(G, w, s)
    Initialize-Single-Source(G, s)
    for i in range (1, |V|)
    {
      for each node u ∈ V
      {
        if d[u] has been updated in previous iteration
          for each edge (u, v) ∈ E
            Relax(u, v)
      }
      if no d[u] value changed in iteration i, stop
    }
```



Example 1: Apply Bellman-Ford algorithm to find the shortest path from A to all other vertices in the graph

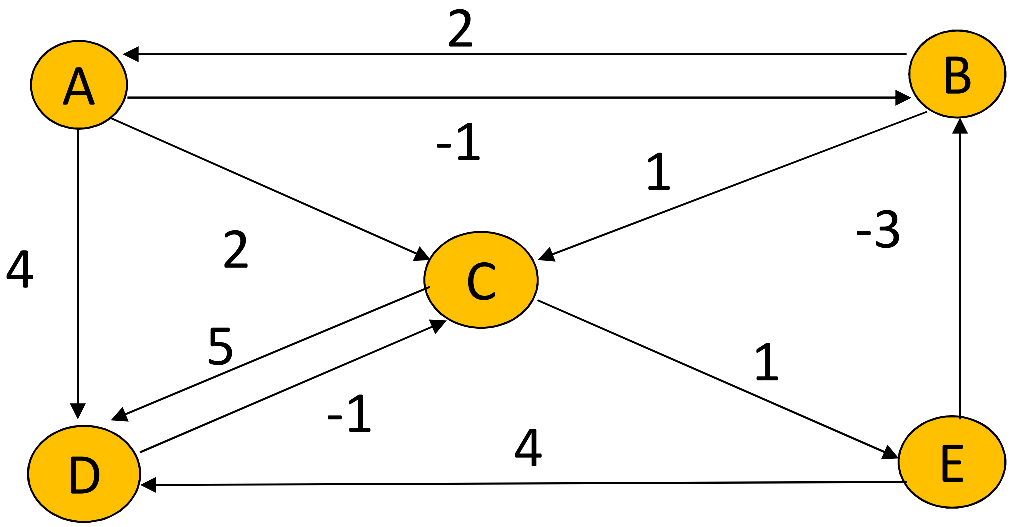


The order of edges examined

1. (A, B)
2. (A, C)
3. (A, D)
4. (B, A)
5. (B, C)
6. (C, D)
7. (C, E)
8. (D, C)
9. (E, B)
10. (E, D)

		Iterations			
Vertex	Initalize	i=1	i=2	i=3	i=4
A	0, –				
B	∞, –				
C	∞, –				
D	∞, –				
E	∞, –				

Example 2: Apply Bellman-Ford algorithm to find the shortest path from A to all other vertices in the graph



The order of edges examined

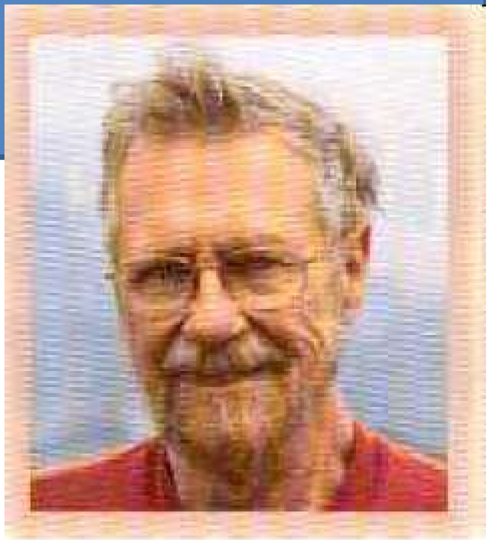
1. (A, B)
2. (A, C)
3. (A, D)
4. (B, A)
5. (B, C)
6. (C, D)
7. (C, E)
8. (D, C)
9. (E, B)
10. (E, D)

		Iterations			
Vertex	Initalize	i=1	i=2	i=3	i=4
A	0, –				
B	∞, –				
C	∞, –				
D	∞, –				
E	∞, –				

Single source shortest path

1. Bellman-Ford algorithm
2. **Dijkstra algorithm**

Dijkstra Algorithm



Edsger W.Dijkstra

(1930-2002)

- In case the weights on the edges are non-negative, the algorithm proposed by Dijkstra is more efficient than the Ford-Bellman algorithm.
- Algorithms are built by labeling vertices. The label of the vertices is initially temporary. At each iteration there is a temporary label that becomes a permanent label. If the label of a vertex u becomes fixed, $d[u]$ gives us the length of the shortest path from the source s to u. Algorithm ends when the labels of all vertices become fixed.

Dijkstra algorithm

- **Input:** A directed graph $G=(V,E)$ and weight matrix $w[u,v] \geq 0$ where $u,v \in V$, source vertex $s \in V$;
  - *G does not have* negative-weight cycle
- **Output:** Each $v \in V$

  $d[v] = \boxed{\delta(s, v);}$ Length of the shortest path from s to v

  $p[v]$ - the predecessor of v in this shortest path from s *to* v.

Use greedy algorithm:

Maintain a set S of vertices for which we know the shortest path

At each iteration:

- grow S by one vertex , choosing shortest path through S to any other vertex not in S
- If the cost from S to any other vertex has decreased, update it

Dijkstra algorithm

```
Dijkstra ( )
{
    for  v ∈ V    // Initialize
    {
        d[v] = w[s,v] ;
        p[v] = s;
    }
    d[s] = 0;   S = {s};    // S: the set of vertices with fixed label (shortest path from s to it has been found)
    T = V \ {s};            // T: the set of vertices with temporary label
    while  (T ≠ ∅)                //Loop
    {
        Find vertex  u ∈ T satisfying  d[u] = min{ d[z] :  z  ∈  T};
        T =  T\ {u};  S= S  ∪ {u};      //Fixed label of vertex u
        for v ∈ adj[u] and v ∈ T //Assign new label to each vertex v of T if necessary (if value d[v] is decreased)
            if  (d[v] > d[u] + w[u,v])
            {
                d[v] = d[u] + w[u,v] ;
                p[v] = u ;
            }
    }
}
```

Use greedy algorithm:

Maintain a set S of vertices for which we know the shortest path

At each iteration:

- grow S by one vertex , choosing shortest path through S to any other vertex not in S
- If the cost from S to any other vertex has decreased, update it

Dijkstra algorithm

```
void Dijkstra ( )
{
    for  v ∈ V   // Initialize
    {
        d[v] = w[s,v] ;
        p[v]=s;
    }
    d[s] = 0;   S = {s};
    T = V \ {s};
    while  (T ≠ ∅)              //Loop
    {
        Find vertex  u ∈ T satisfying  d[u] = min{ d[z] :  z  ∈  T};
        T =  T\ {u};  S= S  ∪ {u};
        for   v ∈ adj[u] and v ∈ T
            if   (d[v] > d[u] + w[u,v])
            {
                d[v] = d[u] + w[u,v] ;
                p[v] = u ;
            }
    }
}
```

- $O(|V|^2)$ operations
  - (|V|-1) iterations: 1 for each vertex u added to the distinguished set S.
  - (|V|-1) iterations: for each adjacent vertex of the one added to the distinguished set.

Dijkstra algorithm

- **Comment:** If only need to find the shortest path from s to t then the algorithm could stop when t has fixed label ($t \in S$).

- $O(|V|^2)$ operations
  - ($|V|$-1) iterations: 1 for each vertex added to the distinguished set S.
  - ($|V|$-1) iterations: for each adjacent vertex of the one added to the distinguished set.

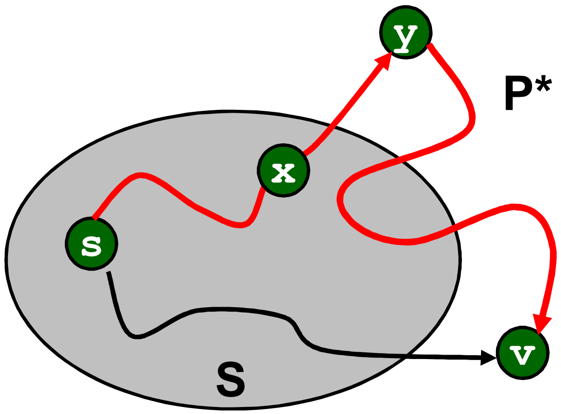
Running time is

$O(|V|^2)$ using linear array for priority queue.

$O((|V| + |E|) \lg |V|)$ using binary heap.

$O(|V| \lg |V| + |E|)$ using Fibonacci heap.

The correctness of the Dijkstra algorithm



We need to proof for each $v \in S$, $d(v) = \delta(s, v)$.

- Induction for |S|.
- <u>Basic case</u>: $|S| = 1$, $d(s) = \delta(s, v) = 0$ is correct.
- <u>Inductive step:</u>
  - Assume algorithm add vertex v into the set S

    → $\boxed{d(v) = \min\{ d[z] : z \notin S\};}$
  - if $d(v)$ is not the length of shortest path from s to v, then denote P* is the shortest path from s to v

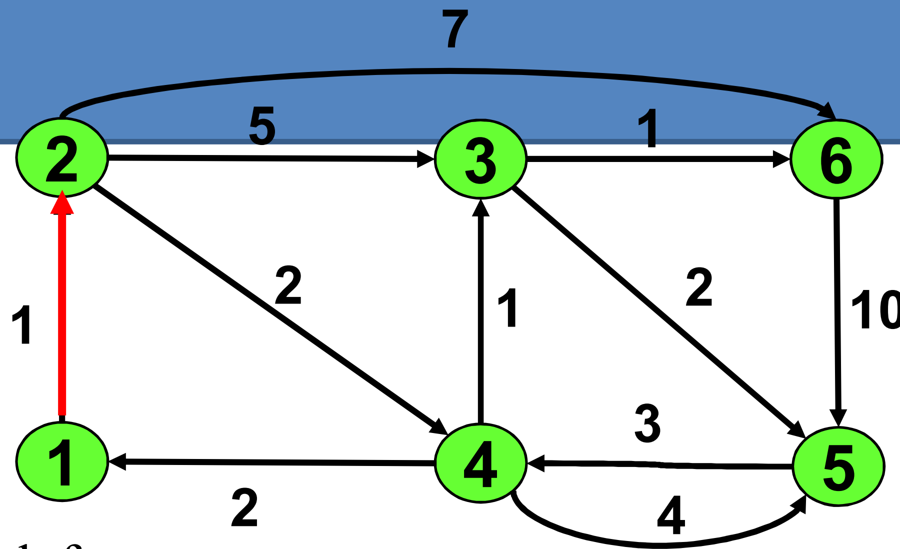
    → P* has to use the edge going out of S, assume (x, y)

then $d(v)$	$>$	$\delta(s, v)$	d(v) is not the length of the shortest path
	$=$	$\delta(s, x) + w(x, y) + \delta(y, v)$	properties 3:all the subpath of the shortest path is also the shortest path
	$\geq$	$\delta(s, x) + w(x, y)$	$\delta(y, v) >= 0$
	$=$	$d(x) + w(x, y)$	induction hypothesis
	$\geq$	$d(y)$	according algorithm

→ $d(v) > d(y)$

So the Dijkstra algorithm select y rather than v ?!

Example



Find the shortest path from vertex 1 to each other vertex

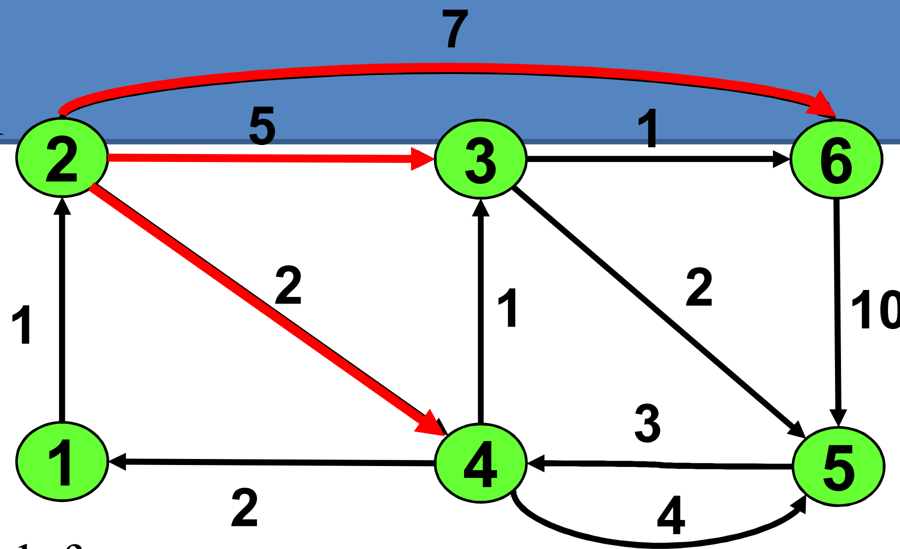
$d[v] = \delta(s, v)$;
$p[v]$ - a predecessor of v in the shortest path from s to v.

	Vertex 1	Vertex 2	Vertex 3	Vertex 4	Vertex 5	Vertex 6
Initialize	[0,1]	[1,1]	[∞,1]	[∞,1]	[∞,1]	[∞,1]
1						
2						
3						
4						
5						

Example

Find the shortest path from vertex 1 to each other vertex

$d[2] = 1$
$p[2]=1$



$d[v] = \delta(s, v)$;
$p[v]$ - a predecessor of v in the shortest path from s to v.

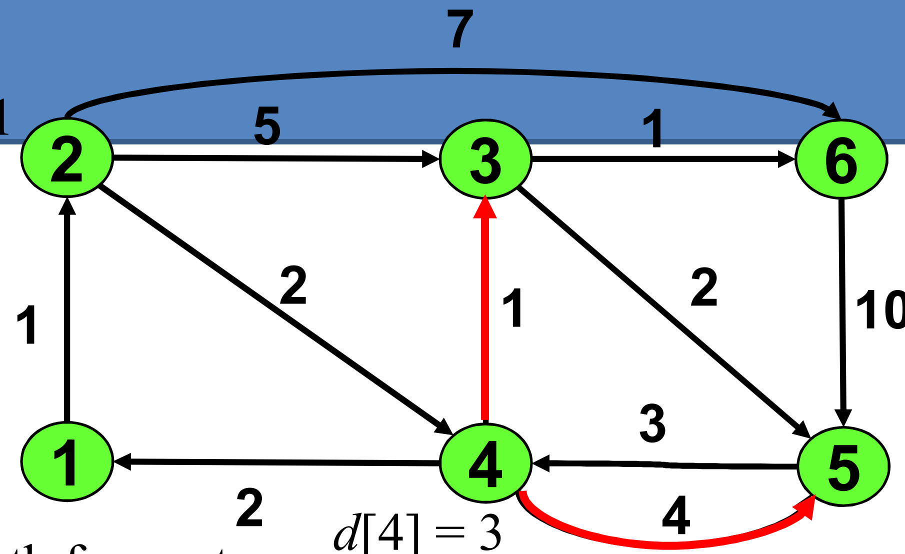
Update label for remaining vertices

	Vertex 1	Vertex 2	Vertex 3	Vertex 4	Vertex 5	Vertex 6
Initialize	[0,1]	[1,1]*	[∞,1]	[∞,1]	[∞,1]	[∞,1]
1	-	-	[6, 2]	[3, 2]	[∞,1]	[8,2]
2		Vertex 3: if (∞ > 1+5) → update label for vertex 3				
3		Vertex 4: if (∞ > 1+2) → update label for vertex 4				
4		Vertex 5: if (∞ > 1+ ∞)				
5		Vertex 6: if (∞ > 1+ 7) → update label for vertex 6				

Example

Find the shortest path from vertex 1 to each other vertex

$d[2] = 1$
$p[2]=1$



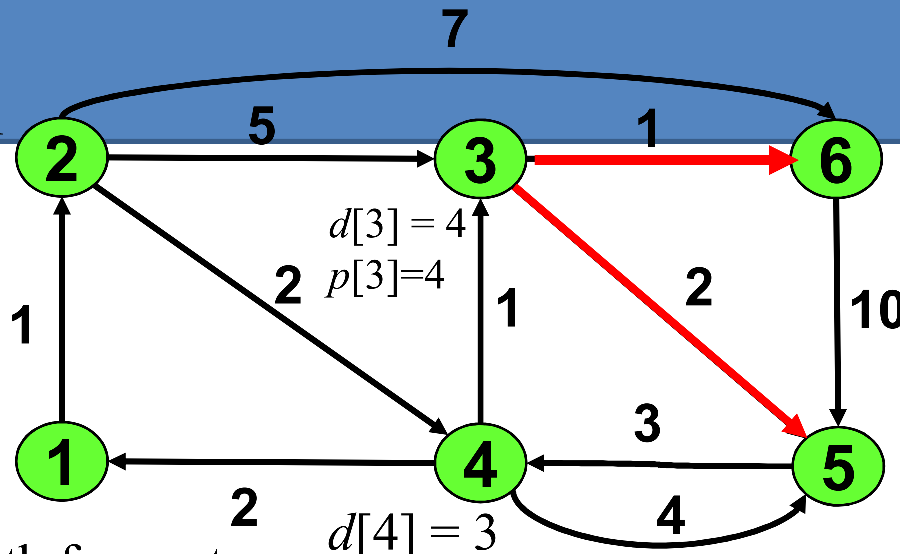
$d[4] = 3$
$p[4]=2$

$d[v] = \delta(s, v)$;
$p[v]$ - a predecessor of v in the shortest path from s to v.

	Vertex 1	Vertex 2	Vertex 3	Vertex 4	Vertex 5	Vertex 6
Initialize	[0,1]	**[1,1]***	[∞,1]	[∞,1]	[∞,1]	[∞,1]
1	-	-	[6, 2]	**[3, 2]***	[∞,1]	[8,2]
2	-	-	[4, 4]	-	[7, 4]	[8,2]
3		Vertex 3: if (6 > 3+1) → update label for vertex 3				
4		Vertex 5: if (∞ > 3+ 4) → update label for vertex 5				
5		Vertex 6: if (8 > 3+ ∞)				

Example

Find the shortest path from vertex 1 to each other vertex

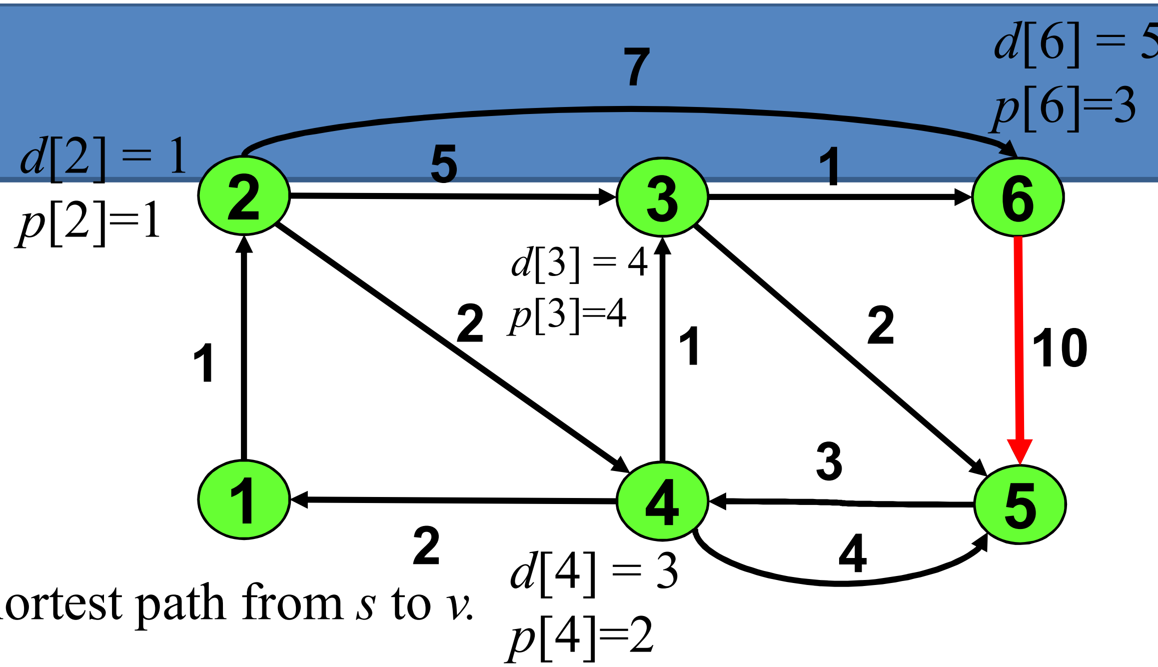


$d[v] = \delta(s, v)$;
$p[v]$ - a predecessor of v in the shortest path from s to v.

	Vertex 1	Vertex 2	Vertex 3	Vertex 4	Vertex 5	Vertex 6
Initialize	[0,1]	**[1,1]***	[∞,1]	[∞,1]	[∞,1]	[∞,1]
1	-	-	[6, 2]	**[3, 2]***	[∞,1]	[8,2]
2	-	-	**[4, 4]***	-	[7, 4]	[8,2]
3	-	-	-	-	[6, 3]	[5, 3]
4		Vertex 5: if (7 > 4+ 2)→ update label for vertex 5				
5		Vertex 6: if (8 > 4+ 1) → update label for vertex 6				

Example

Find the shortest path from vertex 1 to each other vertex



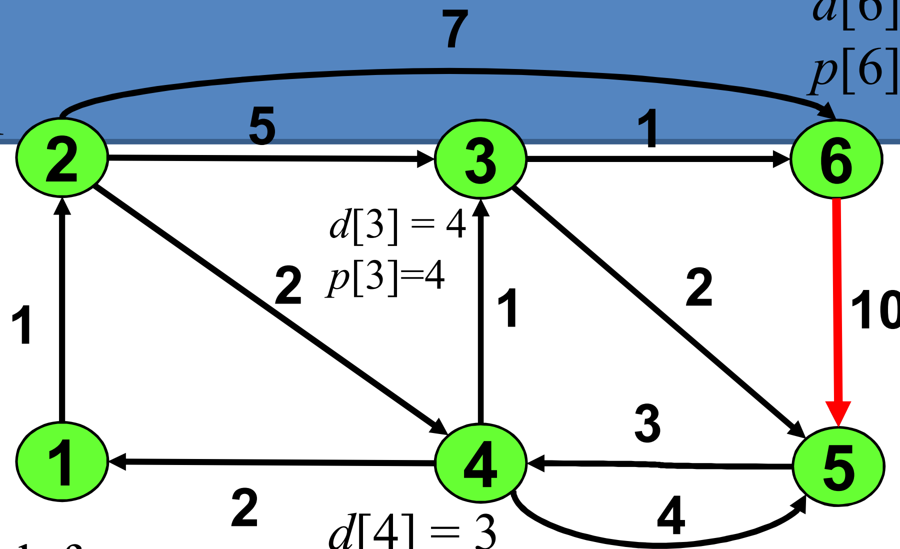
$d[v] = \delta(s, v)$;
$p[v]$ - a predecessor of v in the shortest path from s to v.

	Vertex 1	Vertex 2	Vertex 3	Vertex 4	Vertex 5	Vertex 6
Initialize	[0,1]	**[1,1]***	[∞,1]	[∞,1]	[∞,1]	[∞,1]
1	-	-	[6, 2]	**[3, 2]***	[∞,1]	[8,2]
2	-	-	**[4, 4]***	-	[7, 4]	[8,2]
3	-	-	-	-	[6, 3]	**[5, 3]***
4	-	-	-	-	[6, 3]	-
5						

Vertex 5: if (6 > 5+ 10)

Example

Find the shortest path from vertex 1 to each other vertex



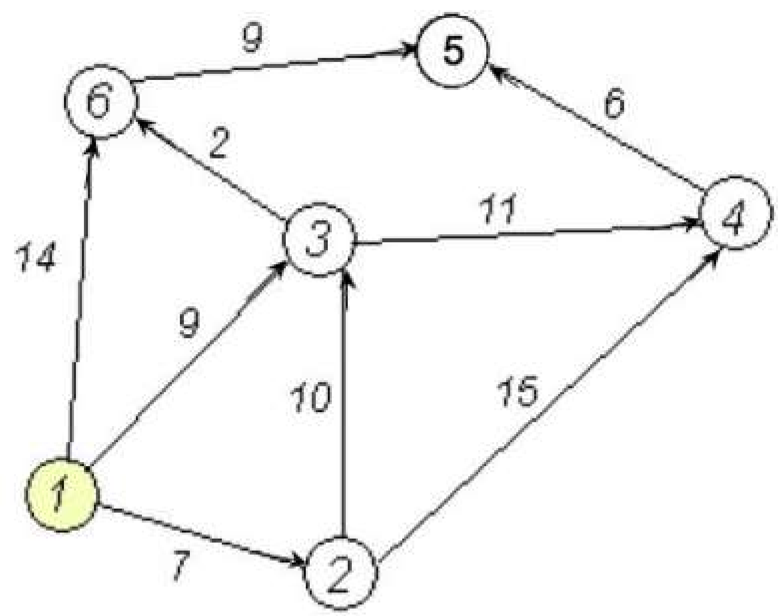
$d[v] = \delta(s, v)$;
$p[v]$ - a predecessor of v in the shortest path from s to v.

	Vertex 1	Vertex 2	Vertex 3	Vertex 4	Vertex 5	Vertex 6
Initialize	[0,1]	**[1,1]***	[∞,1]	[∞,1]	[∞,1]	[∞,1]
1	-	-	[6, 2]	**[3, 2]***	[∞,1]	[8,2]
2	-	-	**[4, 4]***	-	[7, 4]	[8,2]
3	-	-	-	-	[6, 3]	**[5, 3]***
4	-	-	-	-	**[6, 3]***	-
5	-	-	-	-	-	-

Conclusion: The shortest path from vertex 1 to each other vertex of the graph.

Example

- Apply Dijkstra algorithm to find the shortest path from vertex 1 to each other vertex of the graph



Shortest path problems

1. Bellman-Ford algorithm

Edge weight: >, <, = 0

Running time: O(|V||E|)

Shortest path in the graph having cycle

Non-negative length cycle

2. Dijkstra algorithm

Edge weight >=0

Running time: $O(|V|^2)$

3. Shortest path in the directed graph with no cycles

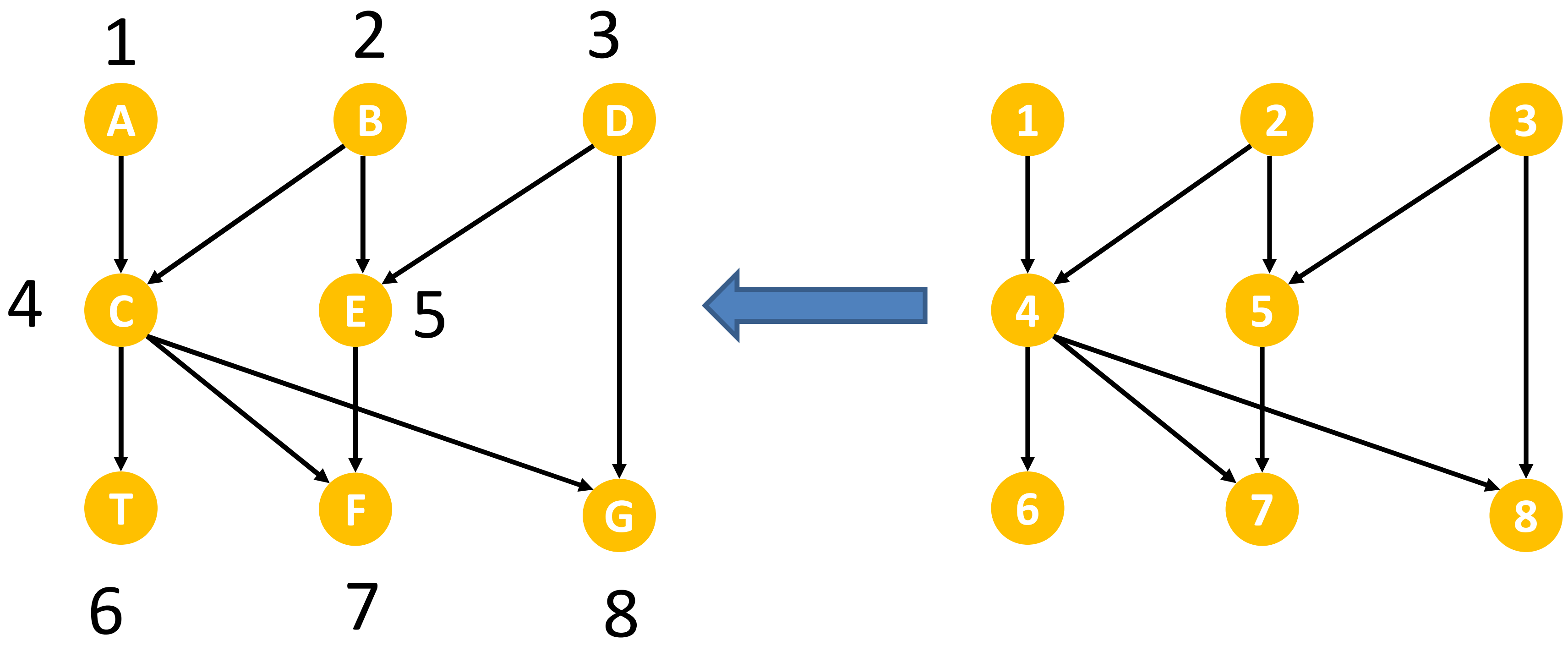
(Directed acyclic graph (DAG))

Edge weight: >, <, = 0

Running time: O(|E|)

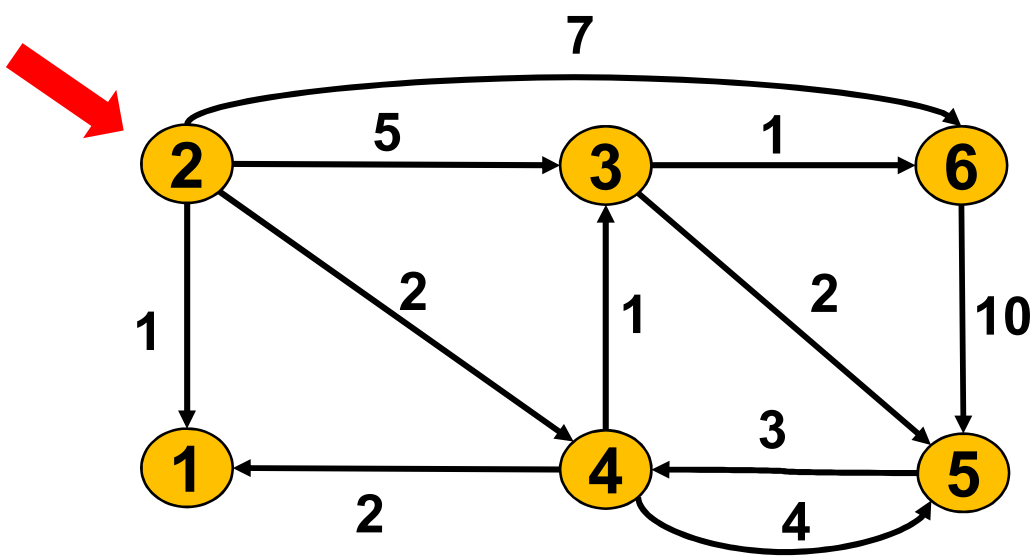
Single-Source Shortest Paths in DAGs

A **topological sort** or **topological ordering** of a DAG is a linear ordering of its vertices such that for every directed edge (*u, v*) from vertex *u* to vertex v, *u* comes before *v* in the ordering. (In orther words: its vertexes can be numbered so that each directed edge starting from the vertex with the smaller index to the vertex with a larger index)



Topological sorting algorithm

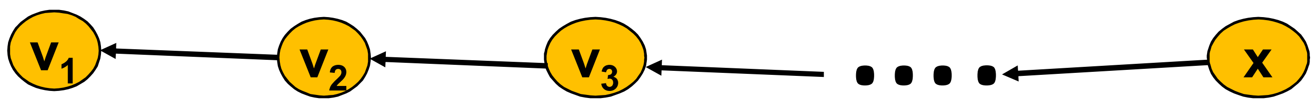
- We see that: *In the DAG, there always exists a vertex with in-dgree = 0*



Topological sorting algorithm

- We see that: *In the DAG, there always exists a vertex with in-dgree = 0*

Indeed, starting at vertex v_1 if there is an incoming edge to it from vertex v_2 then we move to v_2. If there is an edge from v_3 to v_2, then we switch to v_3, ... Since there is not any cycle in the graph, so after a finite number of such transfers we have to go to the vertex without incoming edge.



- **Topological sorting algorithm:**

First, finding all vertices with in-dgree = 0. We index these vertices starting from 1.

Next, removing from graphs vertices that have just been indexed together with the edge going out of them, we get a new graph also without cycle, and we again starting index vertices on this new graph.

The process is repeated until all vertices of the graph has been indexed.

Topological sorting algorithm

- **Input:** DAG $G=(V,E)$ *with the adjacent list* $Adj(v)$, $v \in V$.
- **Ouput:** *For each* $v \in V$ *the index* $NR[v]$ *satisfying*: Each directed edge (u, v): $NR[u] < NR[v]$.

Topological sorting algorithm

void Numbering ( ) Calculate in-dgree for each vertex of the graph

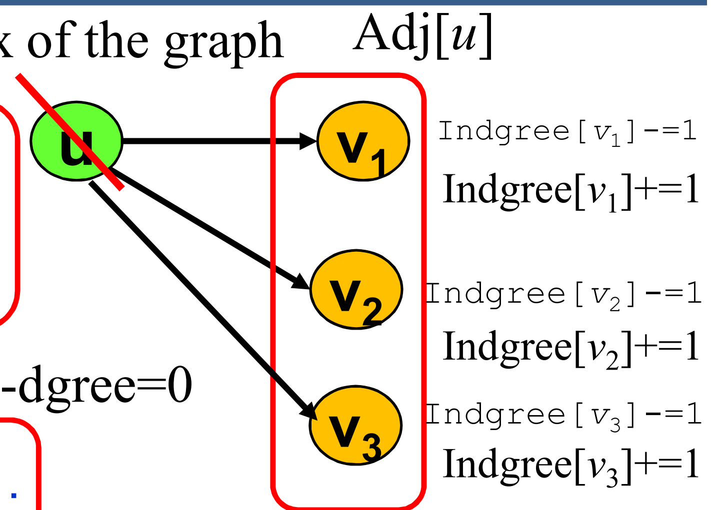
```
{
    for  v ∈ V     Indgree[v] := 0;
    for  u ∈ V           // Tính Indgree[v] = in-dgree of v
         for  v ∈ Ke(u)   Indgree[v] = Indgree[v] + 1 ;
    QUEUE = ∅ ;
    for  v ∈ V
         if  (Indgree[v] == 0) QUEUE = QUEUE ∪ { v } ;
    num = 0;
    while  (QUEUE  ≠ ∅ )
    {
         u ⇐ QUEUE ; num = num + 1 ;  NR[u] = num ;
         for  v ∈ Adj(u)
         {
              Indgree[v] = Indgree[v] - 1 ;
              if  (Indgree[v] == 0) QUEUE = QUEUE ∪ { v } ;
         }
    }
}
```

QUEUE: the set of vertices with in-dgree=0

QUEUE still consists vertices with in-dgree=0

Insert vertices with in-dgree=0 into QUEUE

Indexed vertex u



Remove vertex u *that has just been indexed* from graph together with the edge going out of u

Topological sorting algorithm

- Obviously, in the initial step we must traverse through all the edges of the graph when calculating the in-dgree of the vertices, so that we take $O(|E|)$ operations. Next, each time indexing a vertex, in order to perform the removal of this indexed vertex along with the arcs going out of it, we traverse through all these edges. In order to index all the vertices of the graph we will have to traverse through all the edges of the graph again.
- Therefore, the running time: $O(|E|)$.

Single-Source Shortest Paths in DAGs

Shortest paths are always *well-defined* in *DAGS*

- no cycles => no negative-weight cycles even if there are negative-weight edges

In a DAG:

- Every path is a subsequence of the topologically sorted vertex order
- If we do topological sort and process vertices in that order
- We will process each path in forward order
  - Never relax edges out of a vertex until have processed all edges into the vertex

Thus, just 1 iteration is sufficient

```
DAG-SHORTEST PATHS(G, s)
   TOPOLOGICALLY-SORT the vertices of G
   INIT(G, s)
   for each vertex u taken in topologically sorted order do
      for each v ∈ Adj[u] do
         RELAX(u, v)
```

- Topological sorting: O(|E|)
- Initialzed-Single-Source: O(|E|)
- Nested for-loop: each edge is "traversed" exactly once. Hence, it takes O(|E|) time.

Hence, total running time: O(|E|)

Single-Source Shortest Paths in DAGs

- **Input:** *DAG* $G=(V, E)$ with topological sorting, $V=\{ v[1], v[2], \ldots , v[n] \}$.

  Each directed edge $(v[i], v[j]) \in E$, *we have* $i < j$.

  The adjacent list $Adj(v)$, $v \in V$.

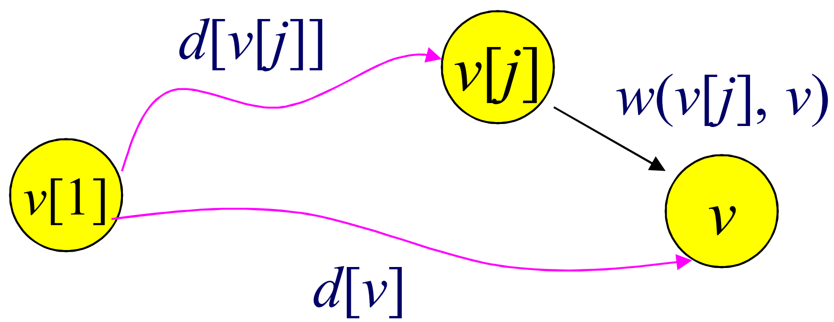
- **Output:** *The shortest path from* $v[1]$ to all other vertices stored in the array $d[v[i]]$, $i = 2, 3, \ldots, n$

```
DAG-SHORTEST PATHS(G, v[1])
  ~~TOPOLOGICALLY-SORT the vertices of G~~
   INIT(G, v[1])
   for each vertex u taken in topologically sorted order do
      for each v ∈ Adj[u] do
         RELAX(u, v)
```

Single-Source Shortest Paths in DAGs

```
Critical_Path ( )
{
     d[v[1]] = 0;
     for j in range (2, n+1) d[v[j]] =∞;
     for v[j] ∈ Adj[v[1]]
             d[v[j]] := w(v[1], v[j]) ;

     for  j in range (2, n+1)
       for  v ∈ Adj[v[j]]
             Relax(v[j],v)

}
```

Initialization:
Init(G, v[1])

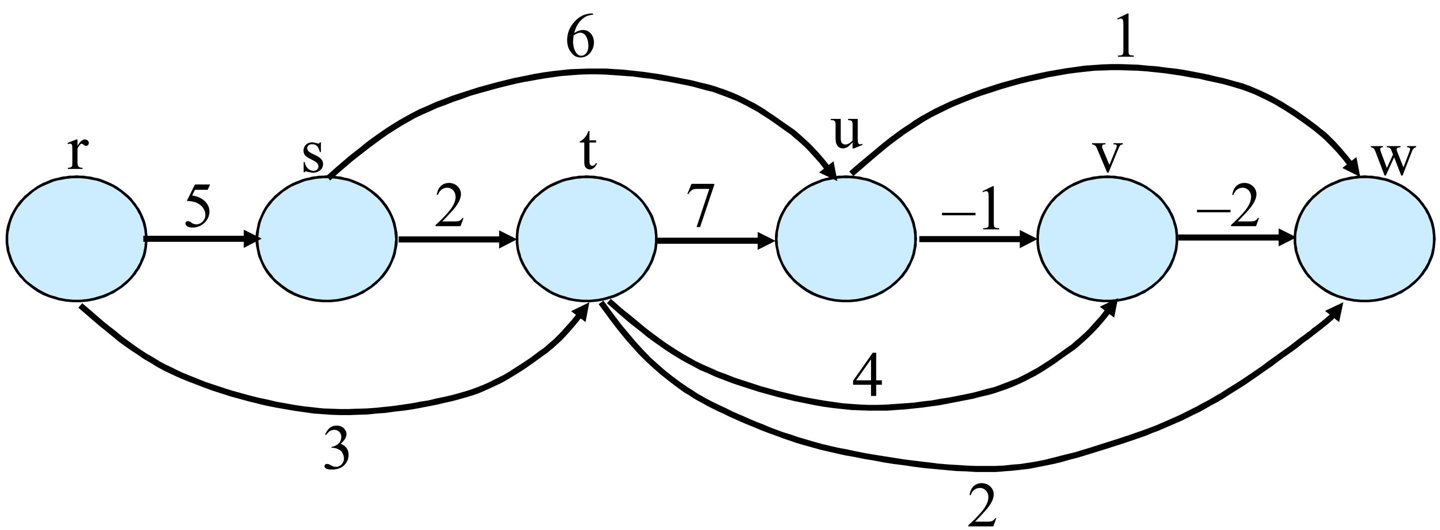



```
d[v] =  min (d[v],  d[v[j]] + w(v[j], v));
```

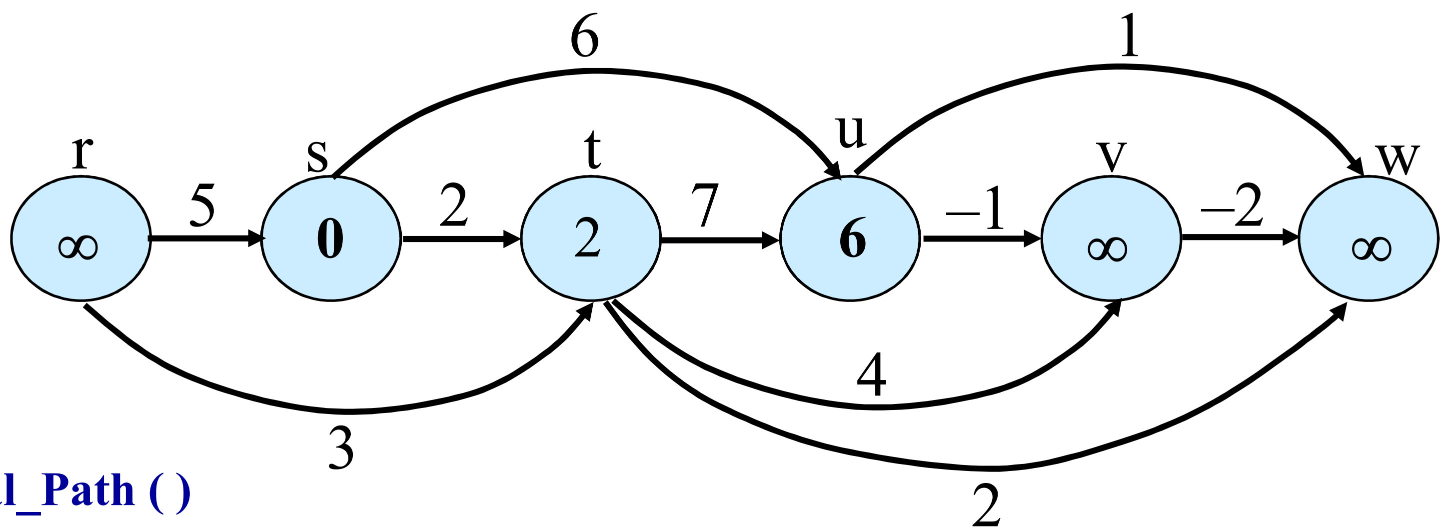
Running time: O(|E|), since each edge is traversed exactly once

Example

Find the shortest path from ***s*** to each other vertices of the DAG graph where vertices are already topological order



Example




```
Critical_Path ( )
{
  d[v[1]] = 0;
  for j in range (2, n+1) d[v[j]] =∞;
  for v[j] ∈ Adj[v[1]]
        d[v[j]] := w(v[1], v[j]) ;
  for j in range (2, n+1)
        for v ∈ Adj[v[j]]
              d[v] = min (d[v], d[v[j]] + w(v[j], v) ) ;
}
```

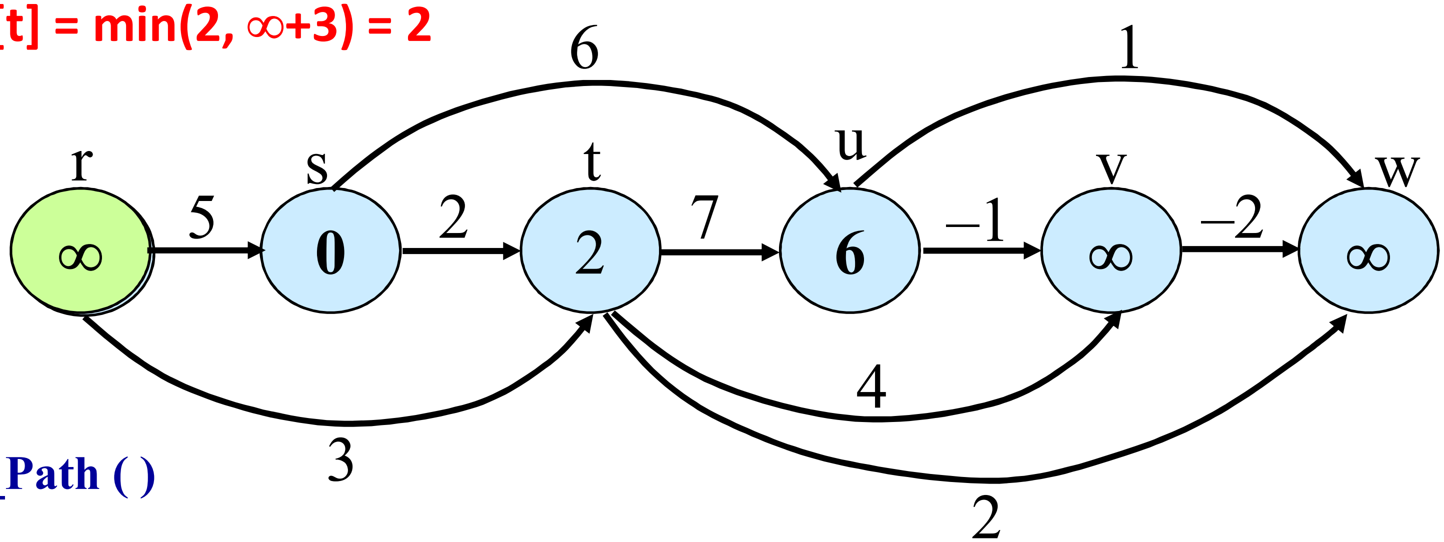
Initialization

j = 2: *v* = *t*

Adj[r] = {*s*, *t*}

d[s] = min(0, ∞+5) = 0

d[t] = min(2, ∞+3) = 2



Critical_Path ( )
```
{
   d[v[1]] = 0;
   for  j in range (2, n+1) d[v[j]] =∞;
   for   v[j] ∈ Adj[v[1]]
         d[v[j]] := w(v[1], v[j]) ;
   for  j in range (2, n+1)
         for  v ∈ Adj[v[j]]
               d[v] =  min (d[v],  d[v[j]] + w(v[j], v) ) ;
}
```

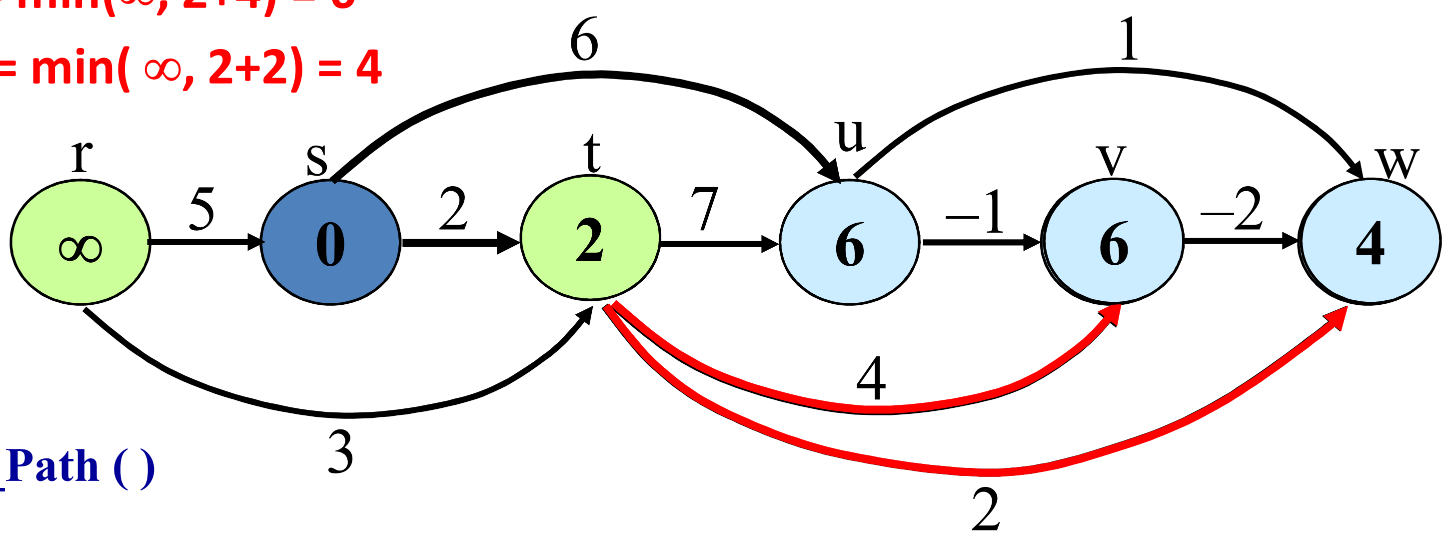
Traverse through vertices: r, t, u, v, w

$j = 3$: $v = t$

Adj[t] = {u, *v*, *w*}

d[v] = min(∞, 2+4) = 6

d[w] = min(∞, 2+2) = 4



Critical_Path ( )

{

d[v[1]] = 0;

for j in range (2, n+1) d[v[j]] =∞;

for v[j] ∈ Adj[v[1]]

d[v[j]] := w(v[1], v[j]) ;

for j in range (2, n+1)

for v ∈ Adj[v[j]]

d[v] = min (d[v], d[v[j]] + w(v[j], v)) ;

}

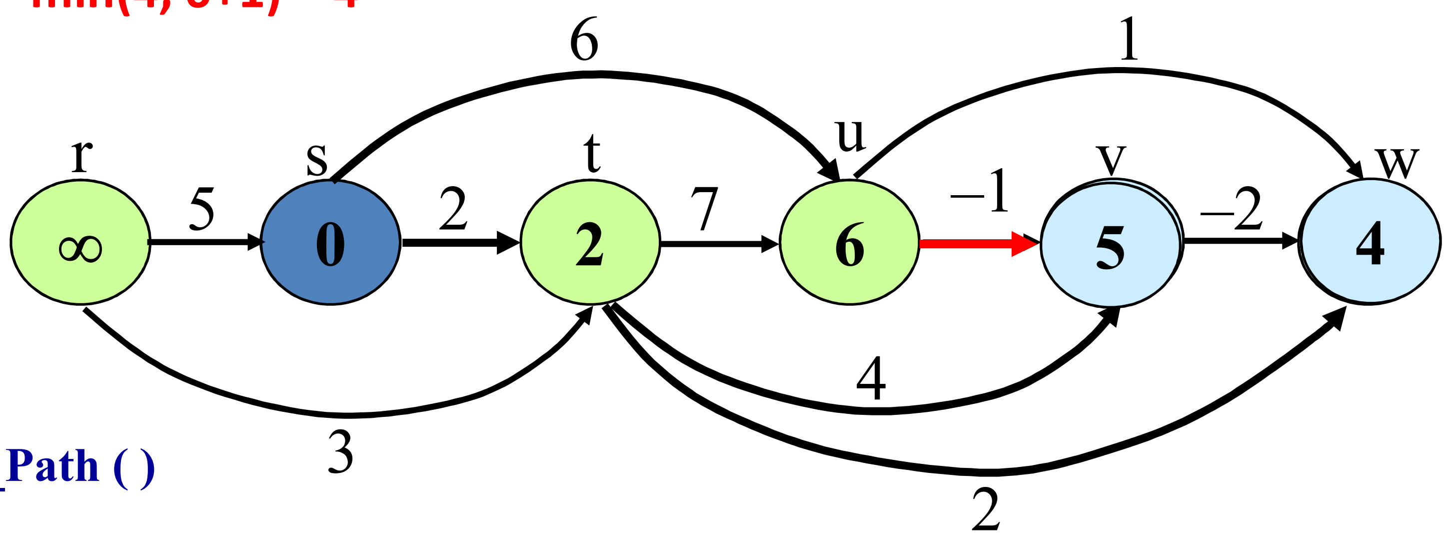
Traverse through vertices: r, t, u, v, w

$j = 4$: $v = u$

Adj[u] = {*v*, *w*}

d[v] = min(6, 6-1) = 5

d[w] = min(4, 6+1) = 4



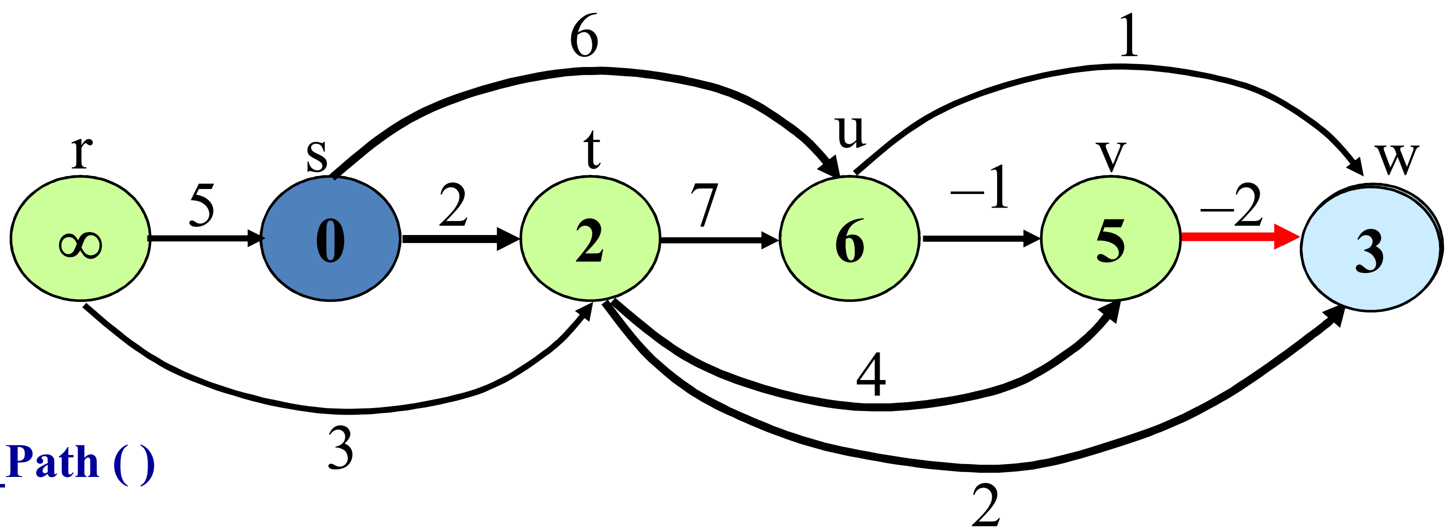

```
Critical_Path ( )
{
   d[v[1]] = 0;
   for  j in range (2, n+1) d[v[j]] =∞;
   for   v[j] ∈ Adj[v[1]]
          d[v[j]] := w(v[1], v[j]) ;
   for  j in range (2, n+1)
          for  v ∈ Adj[v[j]]
                d[v] =  min (d[v],  d[v[j]] + w(v[j], v) ) ;
}
```

Traverse through vertices: r, t, u, v, w

j = 5: v = v

Adj[v] = {*w*}

d[w] = min(4, 5-2) = 3



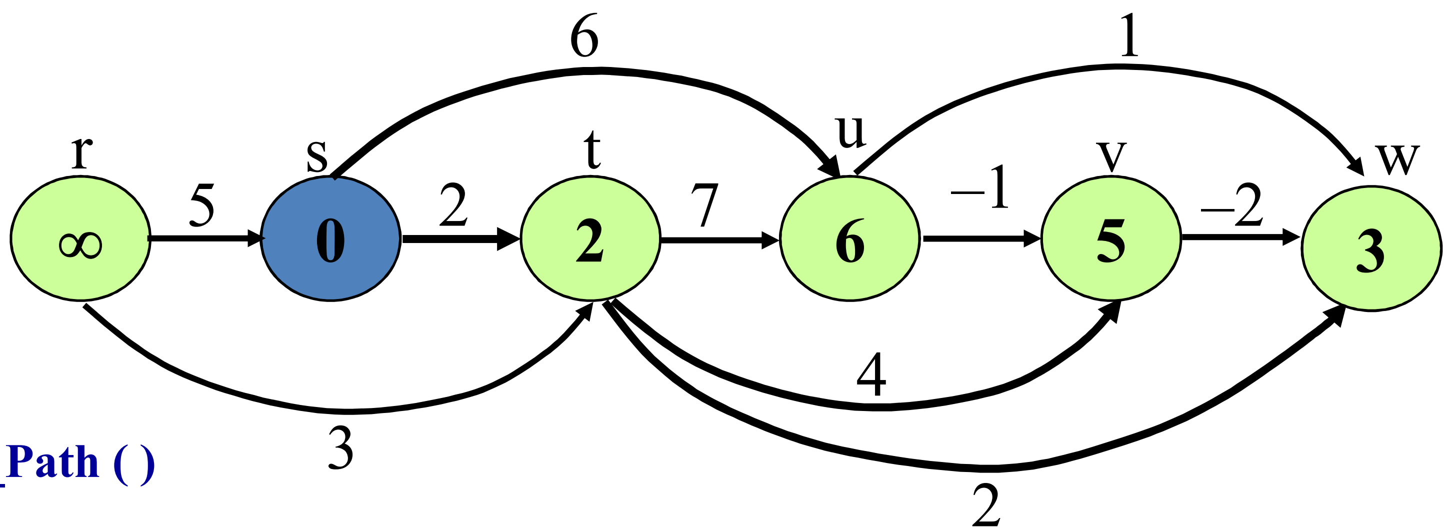
Critical_Path ( )

```
{
   d[v[1]] = 0;
   for  j in range (2, n+1) d[v[j]] =∞;
   for   v[j] ∈ Adj[v[1]]
          d[v[j]] := w(v[1], v[j]) ;
   for  j in range (2, n+1)
          for  v ∈ Adj[v[j]]
               d[v] =  min (d[v],  d[v[j]] + w(v[j], v) ) ;
}
```

Traverse through vertices: r, t, u, v, w

j = 6: v = w

Adj[w] = ∅



Critical_Path ( )

{

d[v[1]] = 0;

for j in range (2, n+1) d[v[j]] =∞;

for v[j] ∈ Adj[v[1]]

d[v[j]] := w(v[1], v[j]) ;

for j in range (2, n+1)

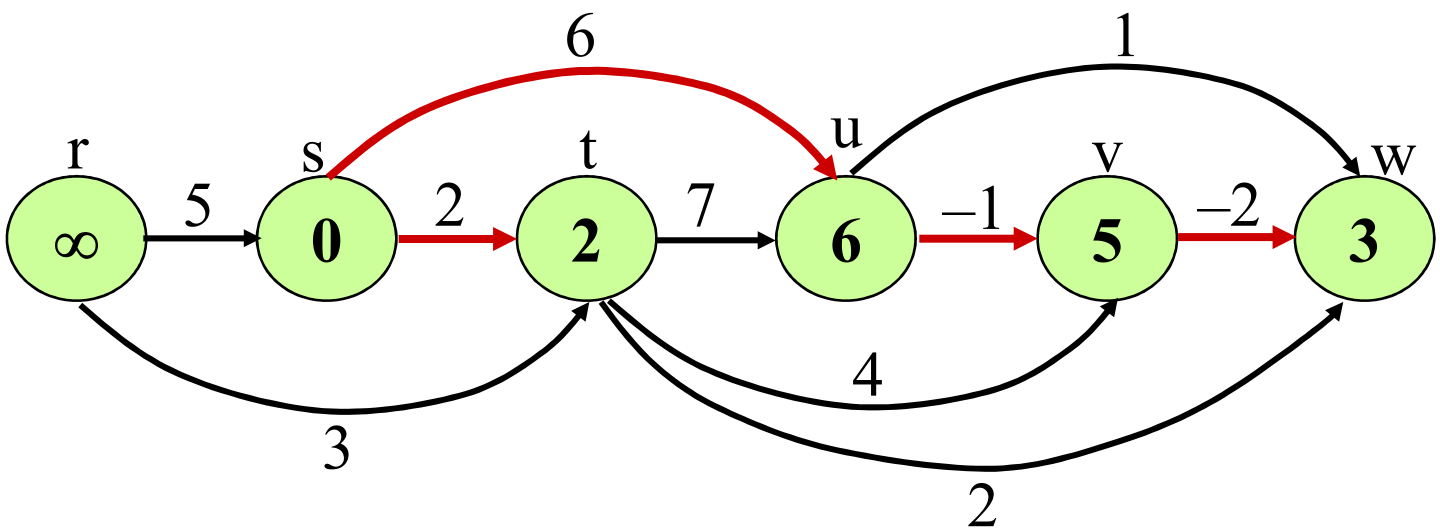
for v ∈ Adj[v[j]]

d[v] = min (d[v], d[v[j]] + w(v[j], v)) ;

}

Traverse through vertices: r, t, u, v, w

Example



Result: The shortest path tree from *s* represented by the red edges

Application: PERT

- PERT (*Project Evaluation and Review Technique*) or CDM (*Critical path Method*).

- The execution of a project is divided into n tasks, numbered from 1 to n. There are a number of tasks that the implementation of which is only carried out after some tasks have been completed. For each task i let $t[i]$ be the time it takes to complete the task ($i = 1, 2, \ldots, n$).

Application: PERT

- Data $n = 8$

Tasks	t[i] (weeks)	Tasks that have been done before it
1	15	None
2	30	1
3	80	None
4	45	2, 3
5	4	4
6	15	2, 3
7	15	5, 6
8	19	5

Application: PERT

PERT: Assume the time of commencement of project is 0. Find the construction progress (specify when each task must be started) to complete the project as soon as possible.

PERT algorithm

We can construct a directed graph with n vertices that represents the order in which the sequence of stasks is performed:

- Each vertex of the graph corresponds to one task.
- If task i has to be done before task j, then on the graph there is a directed edge (i, j), the weight on this edge is $t\,[i]$
- Add to graph two vertices 0 and $n + 1$ corresponding to two special events:
  - vertex 0 corresponds to the *commencement ceremony*, it must be done before all other tasks, and
  - Vertex $n+ 1$ corresponds to the *ribbon cutting ceremony*, it must be done after all tasks,
  - For $t[0] = t[n + 1] = 0$ (actually just connect vertex 0 to all vertices with in-degree=0 and connect all vertices with out-degree=0 to vertex $n + 1$).

The graph obtained is G.

Task	1	2	3	4	5	6	7	8
t[i]	15	30	80	45	4	15	15	19
Tasks must be completed before it	None	1	None	2, 3	4	2, 3	5, 6	5

Each vertex of the graph corresponds to one task.

If task i has to be done before task j, then on the graph there is a directed edge (i, j), the weight on this edge is $t[i]$

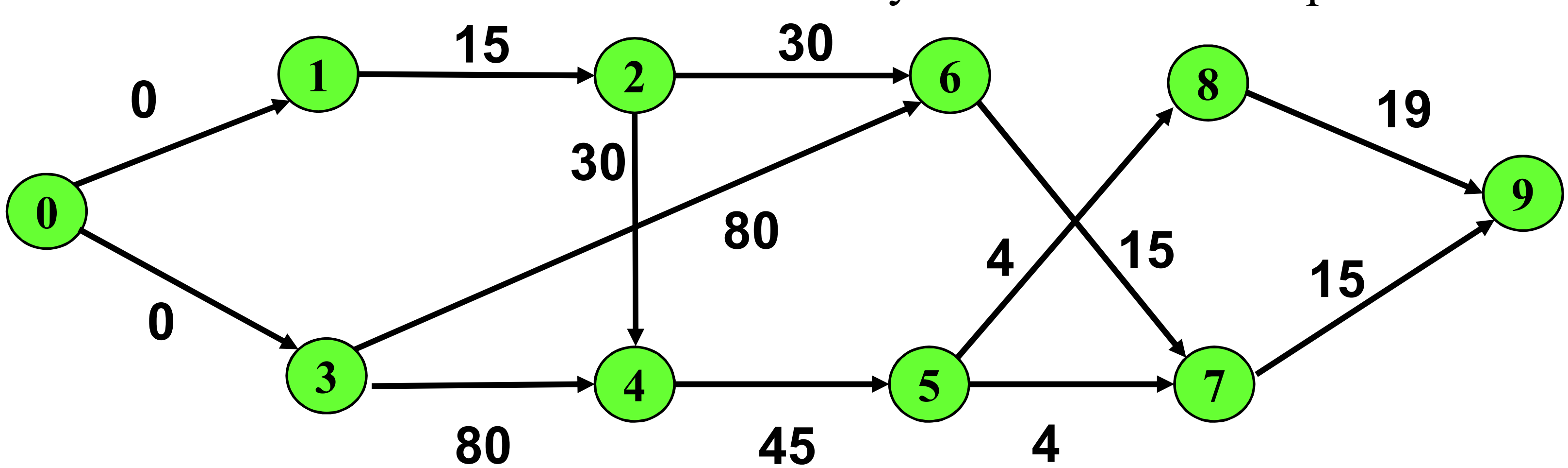
Add to graph two vertices 0 and $n + 1$ corresponding to two special events:

vertex 0 corresponds to the *commencement ceremony*, it must be done before all other tasks, and

Vertex $n+1$ corresponds to the *ribbon cutting ceremony*, it must be done after all tasks,

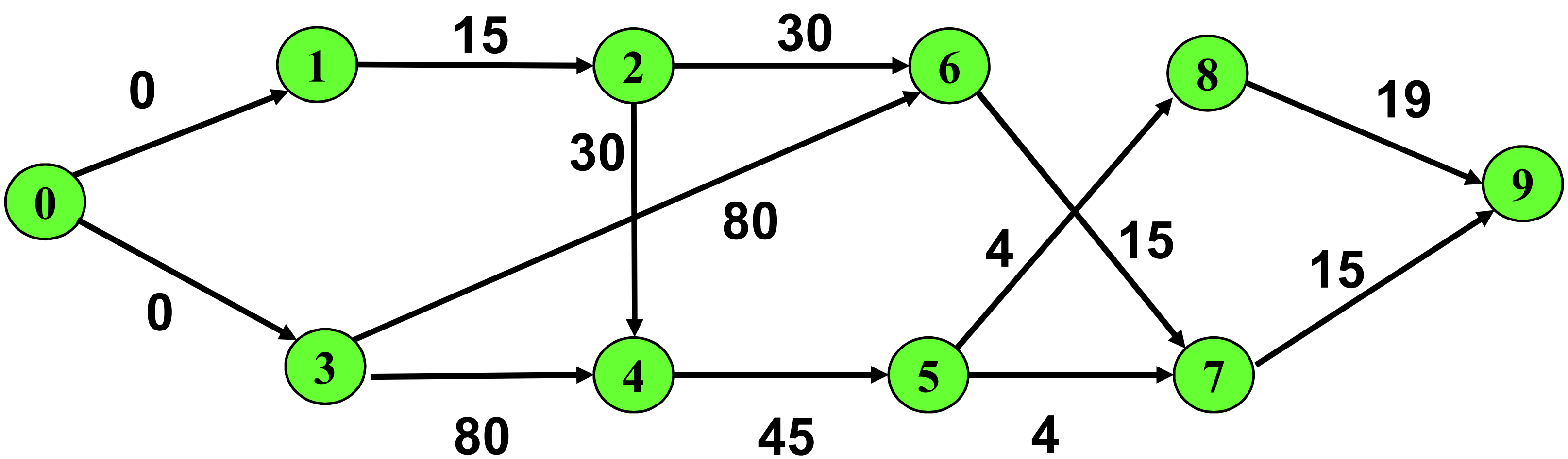
For $t[0] = t[n + 1] = 0$ (actually just connect vertex 0 to all vertices with in-degree=0 and connect all vertices with out-degree=0 to vertex $n + 1$).

Vertices do not have any tasks must be completed before it



PERT: Assume the time of commencement of project is 0. Find the construction progress (specify when each task must be started) to complete the project as soon as possible.

Tasks	1	2	3	4	5	6	7	8
t[i]	15	30	80	45	4	15	15	19
Tasks must be completed before it	None	1	None	2, 3	4	2, 3	5, 6	5



PERT problem leads to the problem finds the longest path from the vertex 0 to all the remaining vertices of graph G.

PERT problem

- Since graph G does not contain a cycle, it is possible to apply the Critical_Path algorithm to solve the given problem by simply changing the min operator to the max operator.
- At the end of the algorithm, we obtain $d[v]$ as the longest path length from veretex 0 to vertex v.
- Then $d[v]$ gives us the earliest possible moment to start the task v

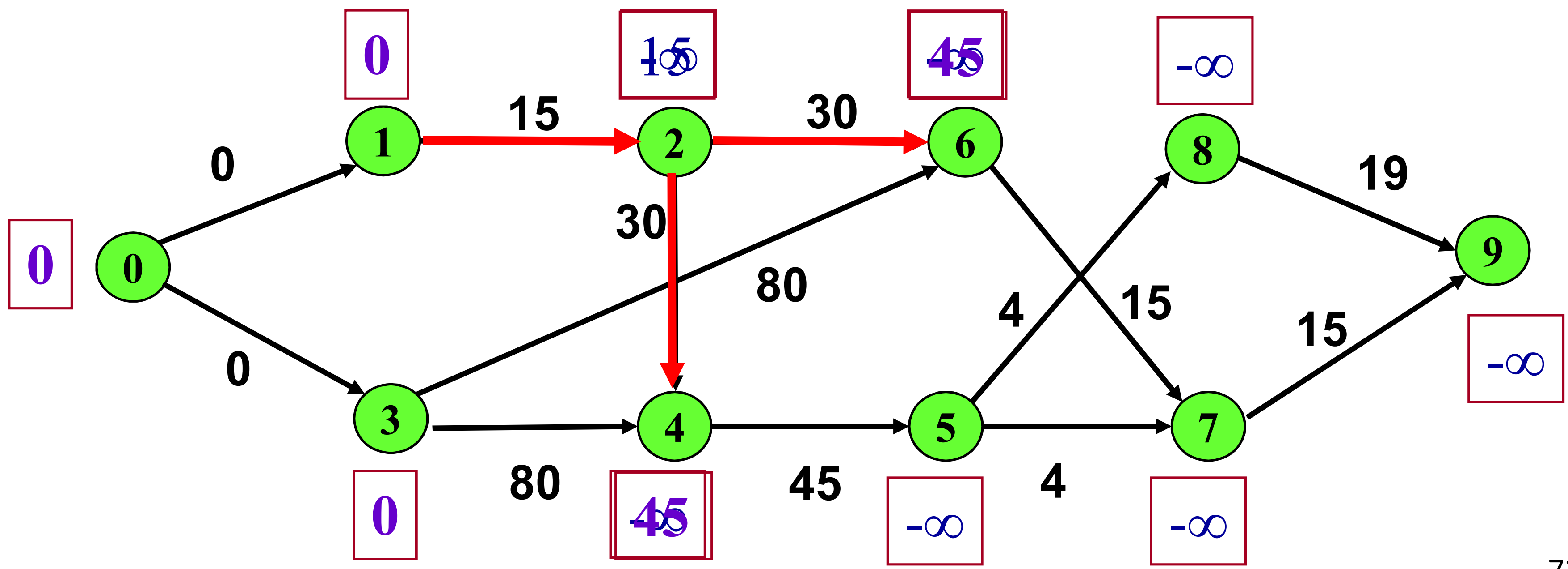
➔ $d\ [n + 1]$ is the earliest time that the ribbon can be cut, i.e. the earliest possible completion time.

```
Critical_Path ( )
{
  d[v[1]] = 0;
  for j in range (2, n+1) d[v[j]] = -∞;
  for   v[j] ∈ Adj[v[1]]
        d[v[j]] := w(v[1], v[j]) ;

  for  j in range (2, n+1)
      for  v ∈ Adj[v[j]]
          d[v] = max (d[v],  d[v[j]] + w(v[j], v) ) ;
}
```

Vertex traversed: 1 2 3 4 5 6 7 8 9

$$d[2] = \max\{-\infty, 0+15\} = 15$$

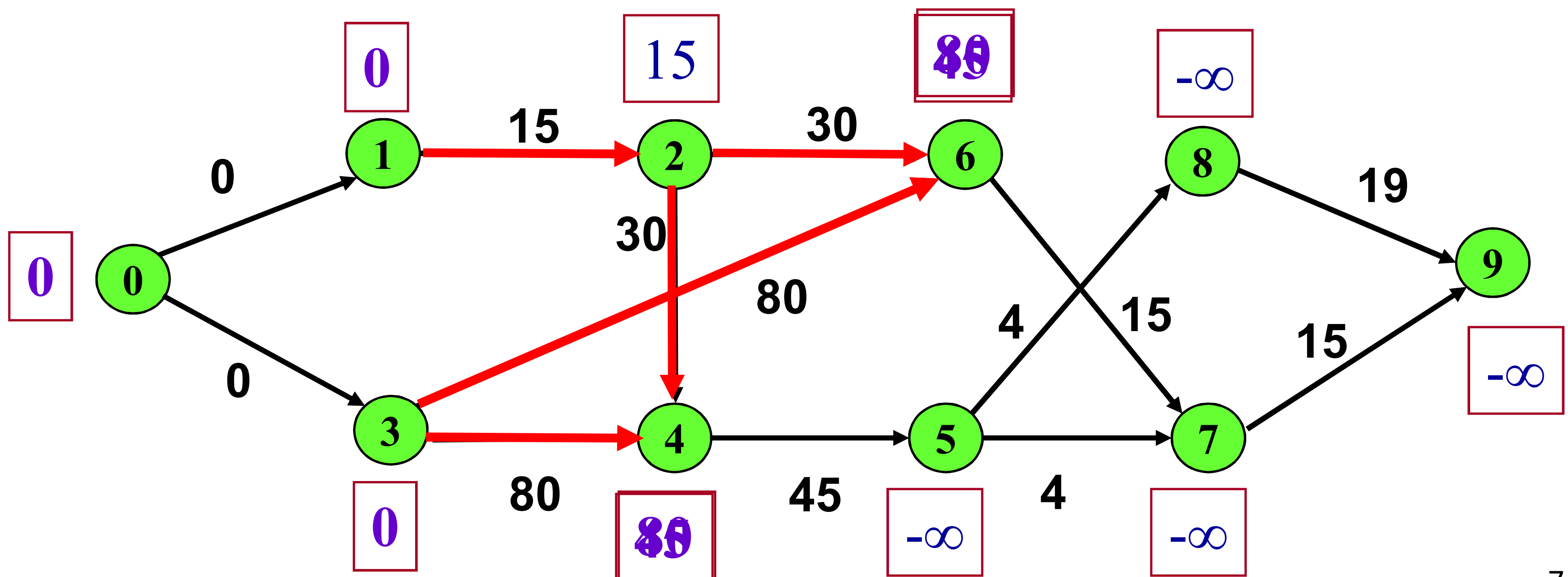
$$d[4] = \max\{-\infty, 15+30\} = 45 \quad d[6] = \max\{-\infty, 15+30\} = 45$$



```
Critical_Path ( )
{
   d[v[1]] = 0;
   for   j in range (2, n+1) d[v[j]] = -∞;
   for   v[j] ∈ Adj[v[1]]
          d[v[j]] := w(v[1], v[j]) ;

   for  j in range (2, n+1)
        for  v ∈ Adj[v[j]]
             d[v] =  max (d[v],  d[v[j]] + w(v[j], v) ) ;
}
```

Vertex traversed: 1 2 3 4 5 6 7 8 9

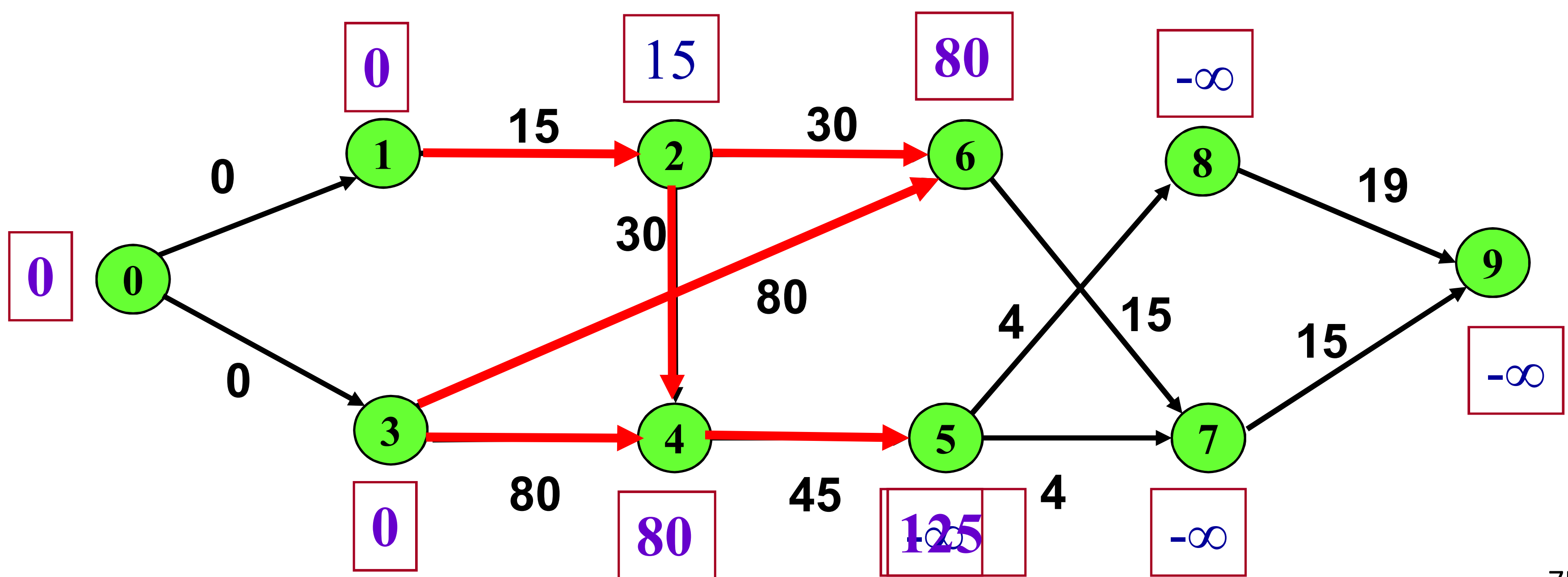
$d[4] = \max \{45, 0+80\} = 80$ $d[6] = \max \{45, 0+80\} = 80$



```
Critical_Path ( )
{
   d[v[1]] = 0;
   for  j in range (2, n+1) d[v[j]] = -∞;
   for  v[j] ∈ Adj[v[1]]
          d[v[j]] := w(v[1], v[j]) ;

   for  j in range (2, n+1)
      for  v ∈ Adj[v[j]]
            d[v] =  max (d[v],  d[v[j]] + w(v[j], v) ) ;
}
```

Vertex traversed: 1 2 3 4 5 6 7 8 9

$$d[5] = \max\{-\infty, 80+45\} = 125$$

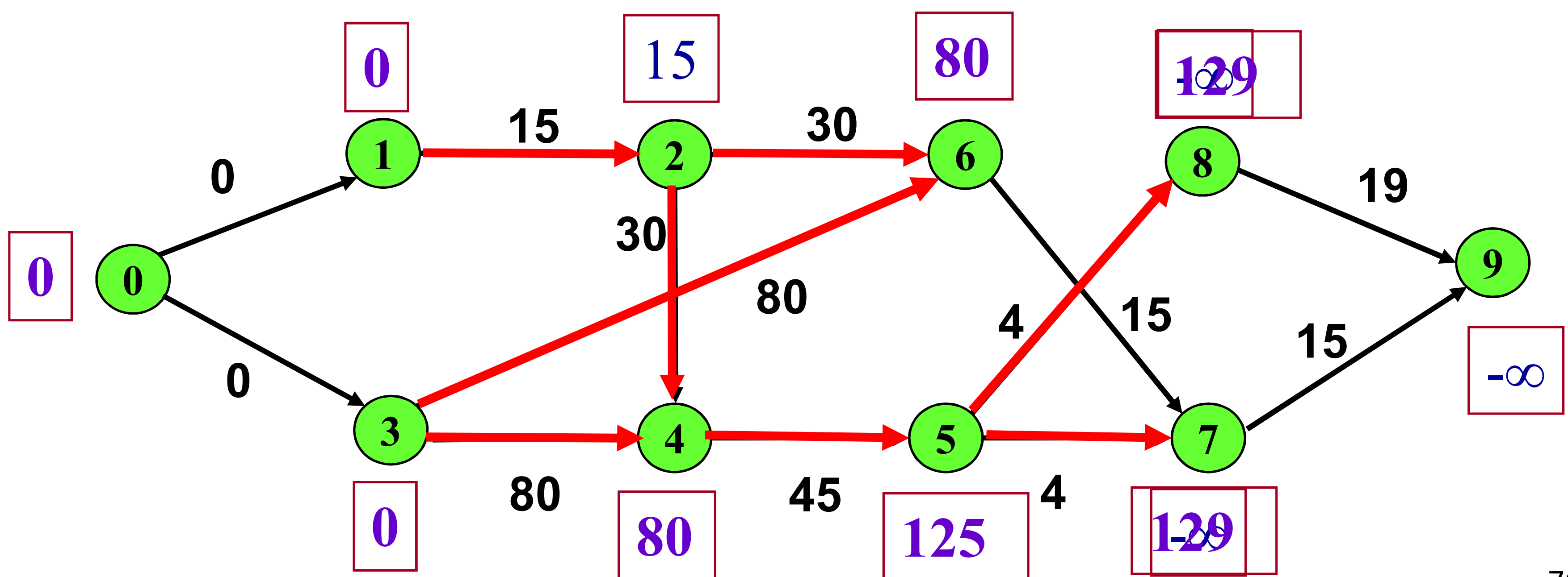


```
Critical_Path ( )
{
   d[v[1]] = 0;
   for   j in range (2, n+1) d[v[j]] = -∞;
   for   v[j] ∈ Adj[v[1]]
          d[v[j]] := w(v[1], v[j]) ;

   for  j in range (2, n+1)
          for  v ∈ Adj[v[j]]
                 d[v] =  max (d[v],  d[v[j]] + w(v[j], v) ) ;
}
```

Vertex traversed: 1 2 3 4 5 6 7 8 9

$d[7] = \max\{-\infty, 125+4\} = 129$

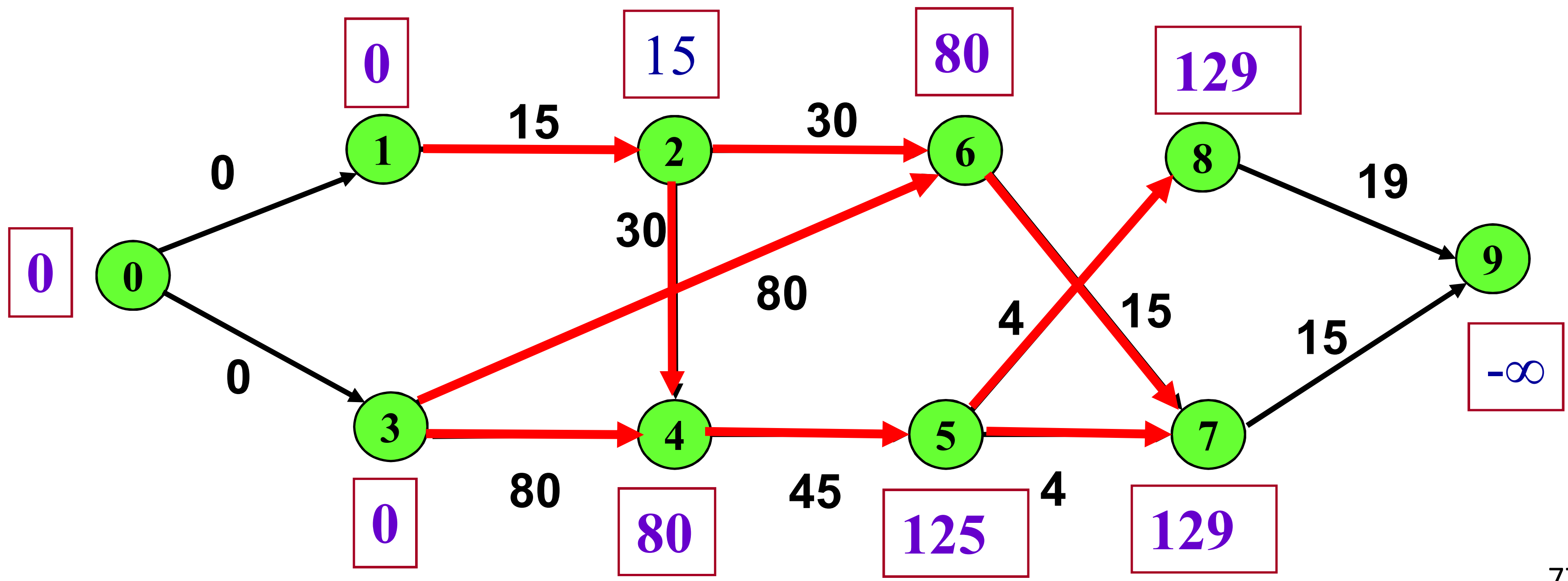
$d[8] = \max\{-\infty, 125+4\} = 129$



```
Critical_Path ( )
{
   d[v[1]] = 0;
   for  j in range (2, n+1) d[v[j]] = -∞;
   for   v[j] ∈ Adj[v[1]]
         d[v[j]] := w(v[1], v[j]) ;

   for  j in range (2, n+1)
         for  v ∈ Adj[v[j]]
               d[v] =  max (d[v],  d[v[j]] + w(v[j], v) ) ;
}
```

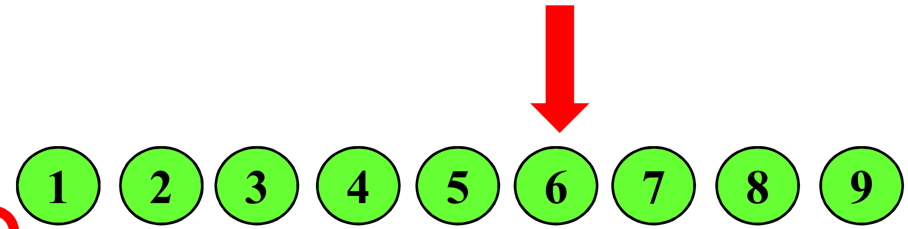
Vertex traversed: 1 2 3 4 5 6 7 8 9

$d[7] = \max\{129, 80+15\} = 129$

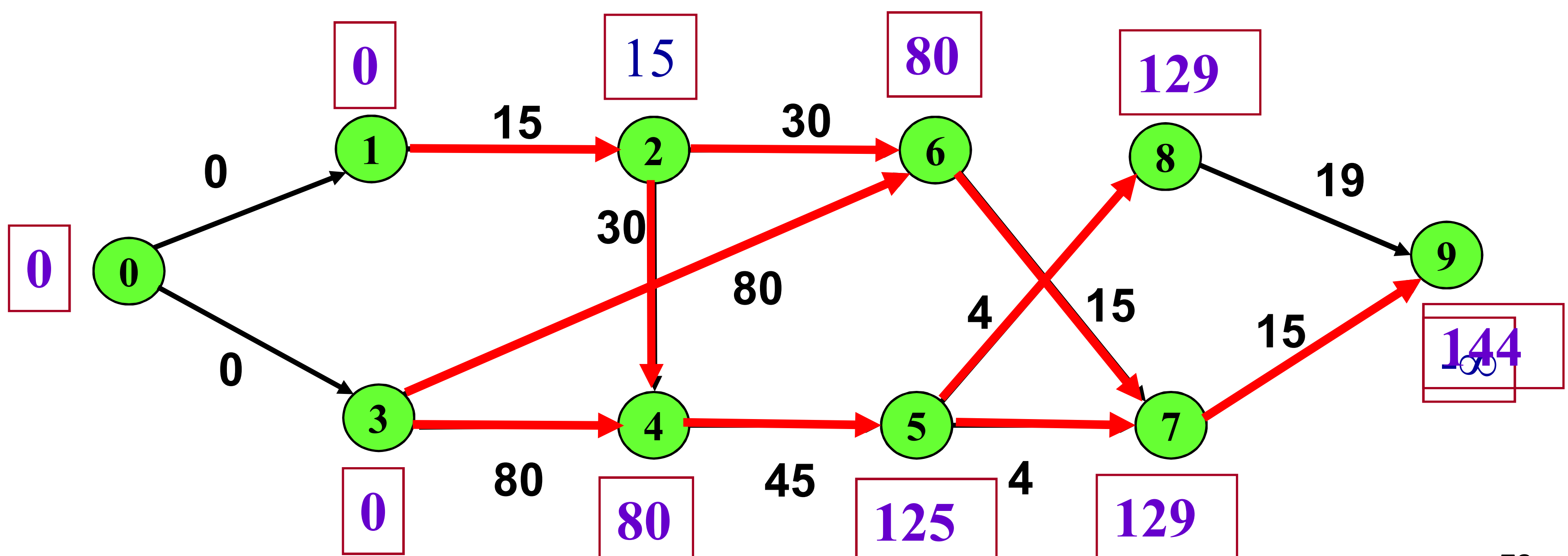


```
Critical_Path ( )
{
   d[v[1]] = 0;
   for  j in range (2, n+1) d[v[j]] = -∞;
   for   v[j] ∈ Adj[v[1]]
          d[v[j]] := w(v[1], v[j]) ;

   for  j in range (2, n+1)
          for  v ∈ Adj[v[j]]
                 d[v] =  max (d[v],  d[v[j]] + w(v[j], v) ) ;
}
```

Vertex traversed:



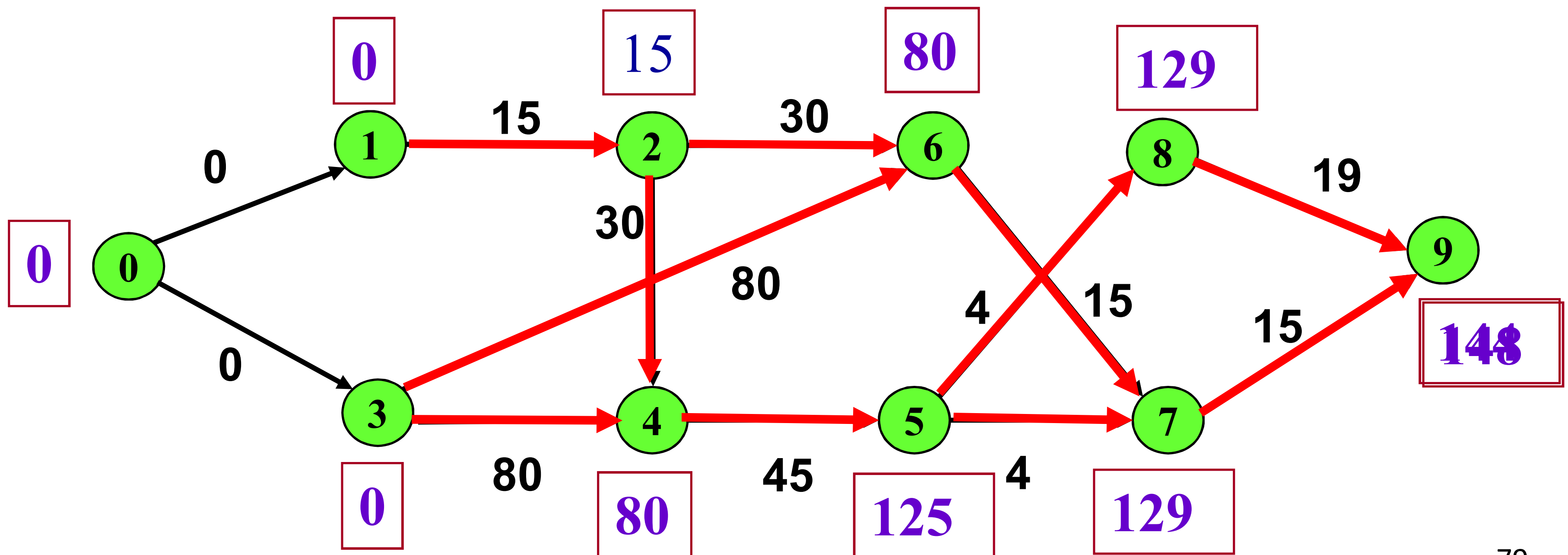
$$d[9] = \max \{-\infty, 129+15\} = 144$$



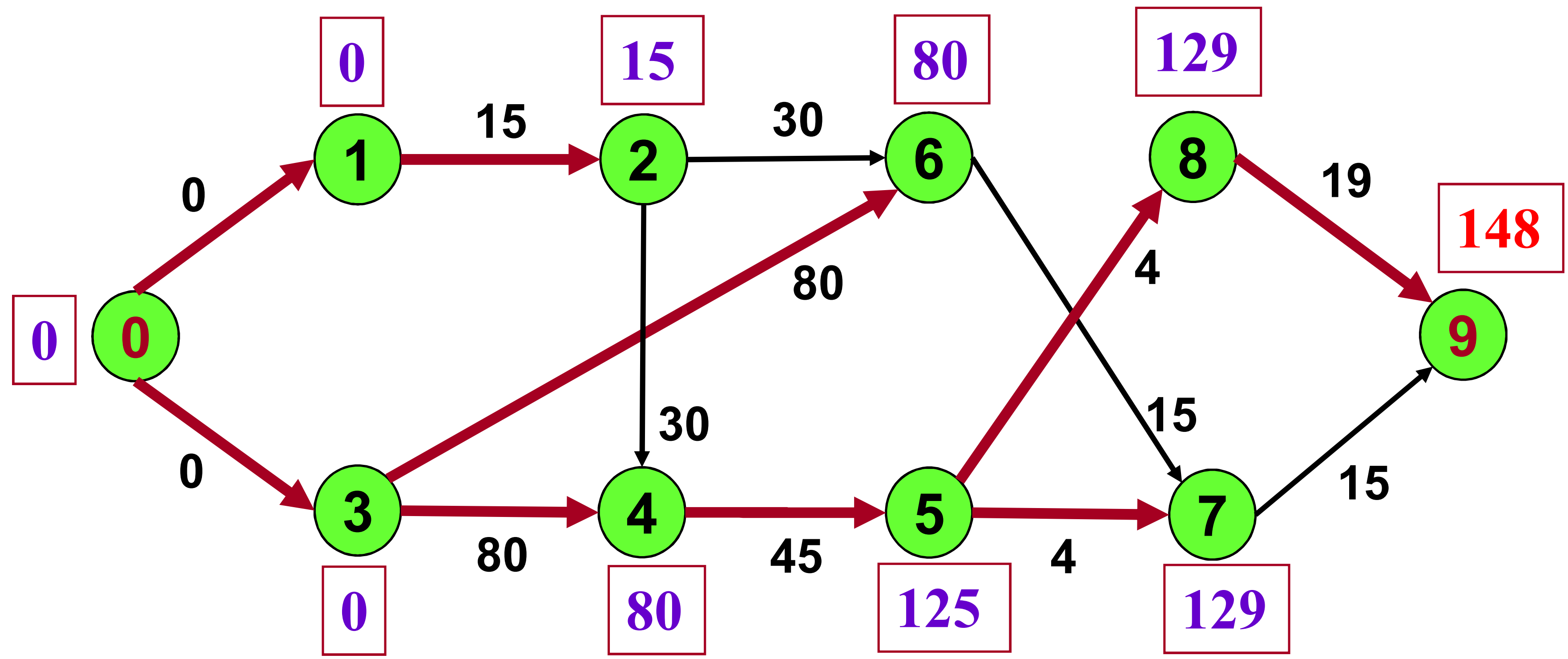
```
Critical_Path ( )
{
   d[v[1]] = 0;
   for  j in range (2, n+1) d[v[j]] = -∞;
   for   v[j] ∈ Adj[v[1]]
          d[v[j]] := w(v[1], v[j]) ;

   for  j in range (2, n+1)
          for  v ∈ Adj[v[j]]
                 d[v] =  max (d[v],  d[v[j]] + w(v[j], v) ) ;
}
```

Vertex traversed: 1 2 3 4 5 6 7 8 9

d[9] = max {144, 129+19} = 148



PERT: Assume the time of commencement of project is 0. Find the construction progress (specify when each task must be started) to complete the project as soon as possible.



Conclusion: The project is completed as early as 148 weeks

Project progress: ???

Example: Shopping Mall Renovation

Activity	Previous tasks	Duration(weeks)
A: Prepare initial design	-	3
B: Identify new potential clients	-	5
C: Develop prospectus for tenants	A	3
D: Prepare final design	A	8
E: Obtain planning permission	D	2
F: Obtain finance from bank	E	3
G: Select contractor	D	4
H: Construction	G, F	17
I: Finalize tenant contracts	B, C, E	13
J: Tenants move in	I, H	2

Shortest-Path Variants

- **Single-source shortest-paths problem**
  - Find a shortest path from a given source (vertex s) to each of the vertices.
- **Single-destination shortest-paths problem**
  - Find a shortest path to a given *destination* vertex t from each vertex v.
- **Single-pair shortest-path problem**
  - Given two vertices, find a shortest path between them. Solution to single-source problem solves this problem efficiently, too.
- **All-pairs shortest-paths problem**
  - Find a shortest path from u to v for every pair of vertices u and v

Comment:

- The problems are arranged in order from simple to complex
- Whenever there is an efficient algorithm for solving one of the three problems, the algorithm can also be used to solve the remaining two problems.

TRƯỜNG ĐẠI HỌC BÁCH KHOA HÀ NỘI
VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

Single-destination shortest paths

Bellman-Ford algorithm

Bellman-Ford algorithm is used to find the shortest path to a vertex t from each other vertex in the graph.

- **Input:** Graph $G=(V,E)$ and weight matrix $w[u,v] \in R$ where $u,v \in V$, source vertex $s \in V$;
  - *G does not contain* negative length cycle

>= 0
<0

- **Output:** Each $v \in V$

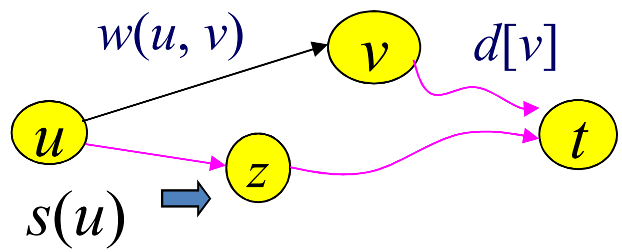
  $d[v] = \boxed{\delta(v, t);}$ Length of the shortest path from v to t

  $s[v]$ - the successor of v in this shortest path from v *to* t.

- If there are no negative edge costs, then any shortest path has at most |*V*|-1 edges. Therefore, algorithm terminates after |*V*|-1 iterations.

Bellman-Ford algorithm

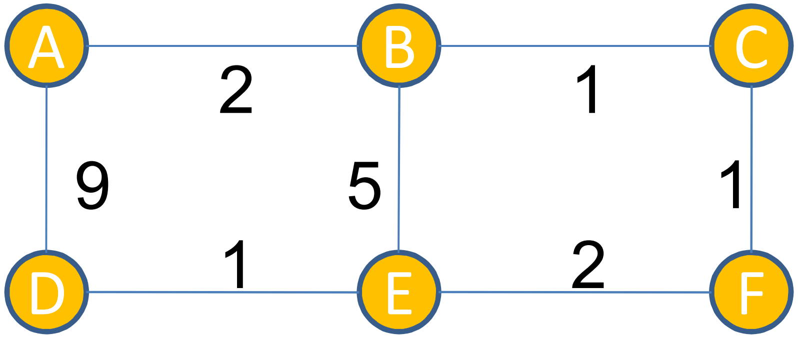
```
Bellman-Ford(G, w, s)
// Step 1: Initialize shortest paths of with at most 0 edges
    Initialize-Single-Destination(G, t)
/* Step 2: Calculate shortest paths with at most i edges from shortest
   paths with at most i-1 edges */
    for i in range (1, |V|)
      for each edge (u, v) ∈ E
          Relax_Des(u, v)
```




```
Initialize-Single-Destination(G, t)
for v ∈ V\t
{
    d[v] = ∞;
    s[v] = Null;
}
s[t]=Null; d[t]=0;
```

```
Relax_Des(u, v)
 if d[u] > w(u, v) + d[v]
 {
     d[u] = w[u,v] + d[v];
     s[u] = v;
 }
```

Example 1: Apply Bellman-Ford algorithm to find the shortest path to *A* from all other vertices in the graph



		Iterations				
Vertex	Init	i=1	i=2	i=3	i=4	i=5
A	0, -					
B	∞, -					
C	∞, -					
D	∞, -					
E	∞, -					
F	∞, -					

Shortest-Path Variants

- **Single-source shortest-paths problem**
  - Find a shortest path from a given source (vertex s) to each of the vertices.
- **Single-destination shortest-paths problem**
  - Find a shortest path to a given *destination* vertex t from each vertex v.
- **Single-pair shortest-path problem**
  - Given two vertices, find a shortest path between them. Solution to single-source problem solves this problem efficiently, too.
- **All-pairs shortest-paths problem**
  - Find a shortest path from u to v for every pair of vertices u and v

Comment:

- The problems are arranged in order from simple to complex
- Whenever there is an efficient algorithm for solving one of the three problems, the algorithm can also be used to solve the remaining two problems.

TRƯỜNG ĐẠI HỌC BÁCH KHOA HÀ NỘI
VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

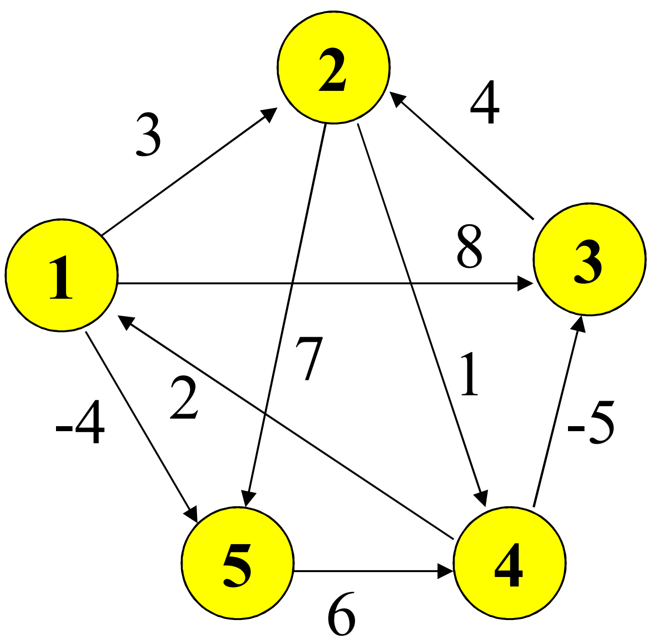
All pairs shortest-paths

All pairs shortest-paths

Problem Given directed graph $G = (V, E)$, with weight on each edge e is $w(e)$,
for each pair of vertices u, v of V, find the shortest path from u to v.

- Input: *weight matrix*
- Output *matrix*: element at row u column v is the length of the shortest path from u to v.
- Allow negative-weight edge
- **Assumption: None negative-length cycle**

Example



Input

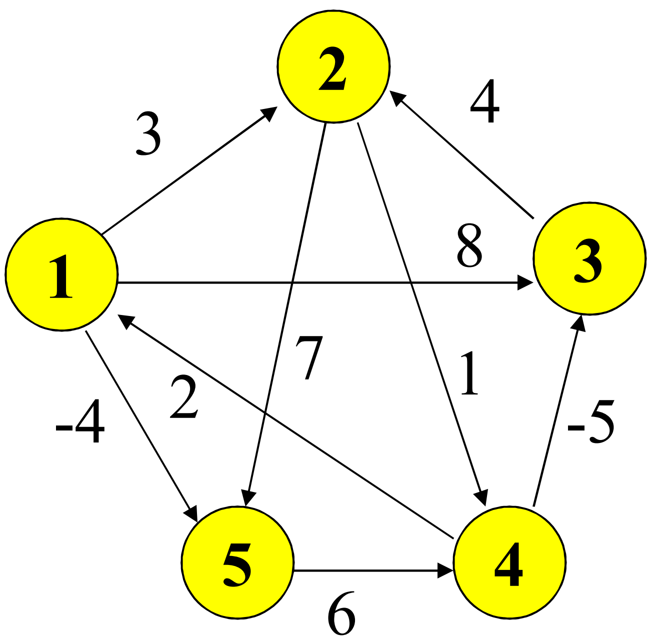
Weight matrix $W_{nxn} = (w)_{ij}$ where

$$w_{ij} = \begin{cases} 0 & \text{if } i = j \\ w(i,j) & \text{if } i \neq \text{j} \ \& \ (i,j) \in E \\ \infty & \text{otherwise} \end{cases}$$

$$W_{5x5} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix}$$

Output

Matrix: element at row u column v is the length of the shortest path from u to v.



Shortest path from 1 to 2 : 1- 5 - 4 - 3 - 2

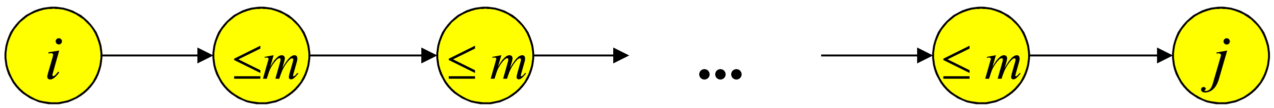
$= -\ 4 + 6 - 5 + 4$

$$\begin{pmatrix} 0 & \mathbf{1} & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ \mathbf{8} & 5 & 1 & 6 & 0 \end{pmatrix}$$

Shortest path from 5 to 1 : 5 - 4 - 1

Floyd-Warshall algorithm

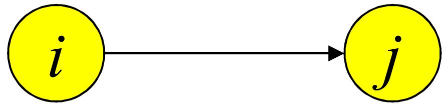
$d_{ij}^{(m)}$ = length of the shortest path from i to j using intermediate vertices in the vertex set $\{1, 2, \ldots, m\}$.



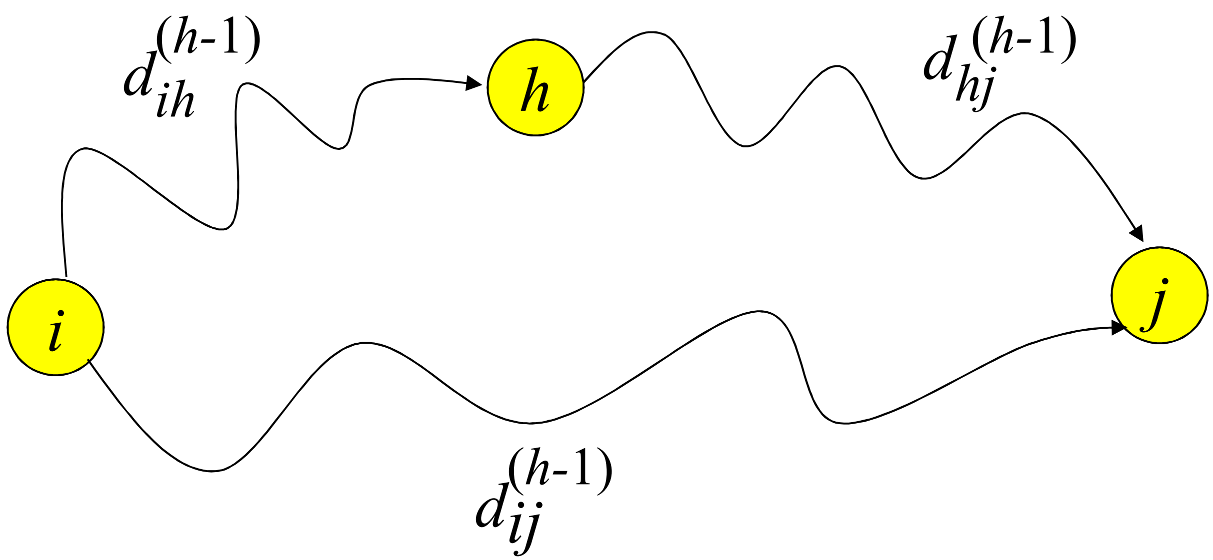
Grap with n vertices $\{1,2,..,n\}$ → length of the shortest path from i to j is $d_{ij}^{(n)}$

Recursive formula computed $d^{(h)}$

- $d_{ij}^{(0)} = w_{ij}$



- $d_{ij}^{(h)} = \min\left(d_{ij}^{(h-1)},\ d_{ih}^{(h-1)} + d_{hj}^{(h-1)} \right)$ if $h \geq 1$



Floyd-Warshall algorithm

```
void Floyd-Warshall(n, W)
{
    D^(0) ← W
    for k in range (1, n+1)
        for i in range (1, n+1)
            for j in range (1, n +1)
                d_ij^(k) ← min(d_ij^(k-1), d_ik^(k-1) + d_kj^(k-1))
    return D^(n);
}
```

Path going through only intermediate vertices selected from $\{1,2,..,k\}$

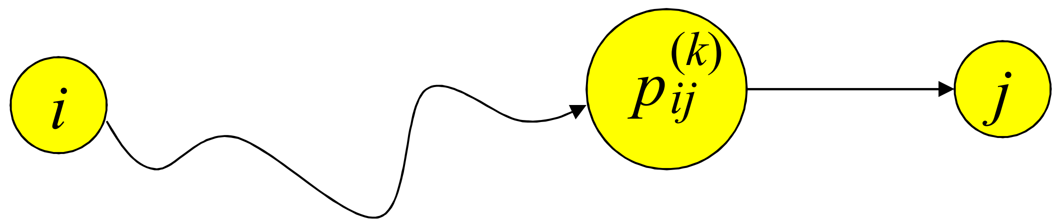
All pairs (i, j)

$$d_{ij}^{(k)} \leftarrow \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)})$$

Running time $\Theta(n^3)$!

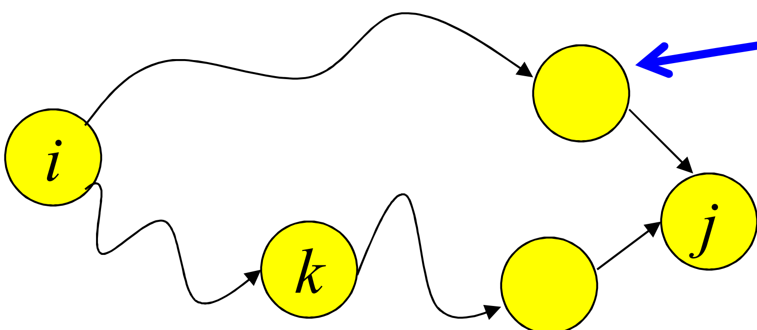
Build the shortest path

Predecessor matrix $P^{(k)} = (p_{ij}^{(k)})$:



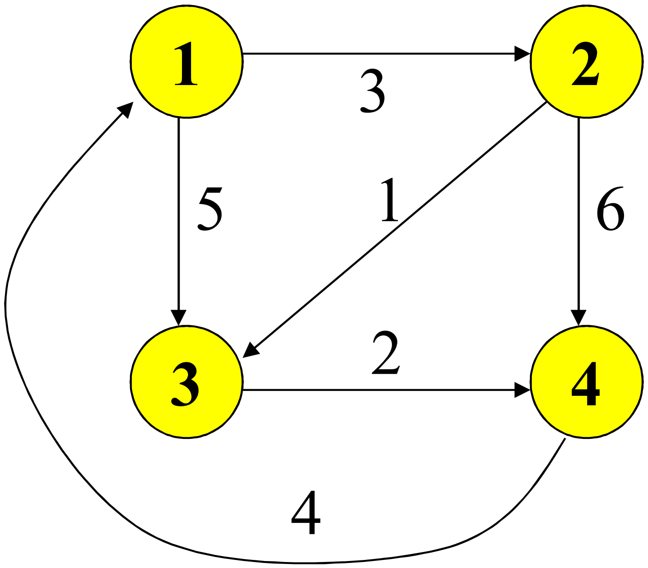
Shortest path from i to j going through intermediate vertices only selected from $\{1, 2, \ldots, k\}$.

$$p_{ij}^{(0)} = \begin{cases} i, & \text{if } (i,j) \in E \\ \text{Nil}, & \text{if } (i,j) \notin E \end{cases}$$



$$p_{ij}^{(k)} = \begin{cases} p_{ij}^{(k\text{-}1)} & \text{if } \quad d_{ij}^{(k\text{-}1)} \leq d_{ik}^{(k\text{-}1)} + d_{kj}^{(k\text{-}1)} \\ p_{kj}^{(k\text{-}1)} & \text{otherwise} \end{cases}$$

Example: Find the shortest path between every pairs of vertices



$$D^{(0)} \leftarrow W$$

$$p_{ij}^{(0)} = \begin{cases} i, & \text{if } (i,j) \in E \\ \text{Nil}, & \text{if } (i,j) \notin E \end{cases}$$

$$D^{(0)} \begin{pmatrix} 0 & 3 & 5 & \infty \\ \infty & 0 & 1 & 6 \\ \infty & \infty & 0 & 2 \\ 4 & \infty & \infty & 0 \end{pmatrix} \qquad P^{(0)} \begin{pmatrix} \text{Nil} & 1 & 1 & \text{Nil} \\ \text{Nil} & \text{Nil} & 2 & 2 \\ \text{Nil} & \text{Nil} & \text{Nil} & 3 \\ 4 & \text{Nil} & \text{Nil} & \text{Nil} \end{pmatrix}$$

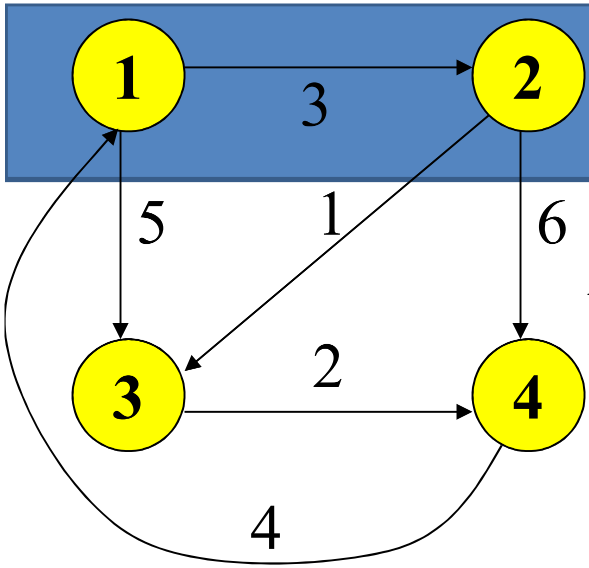
$$d_{ij}^{(k)} \leftarrow \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)})$$

Could use 1 as intermediate vertex:

$$D^{(1)} \begin{pmatrix} 0 & 3 & 5 & \infty \\ \infty & 0 & 1 & 6 \\ \infty & \infty & 0 & 2 \\ 4 & 7 & 9 & 0 \end{pmatrix} \qquad P^{(1)} \begin{pmatrix} \text{Nil} & 1 & 1 & \text{Nil} \\ \text{Nil} & \text{Nil} & 2 & 2 \\ \text{Nil} & \text{Nil} & \text{Nil} & 3 \\ 4 & 1 & 1 & \text{Nil} \end{pmatrix}$$

$= \min(\infty, 4+3)$

$= \min(\infty, 4+5)$



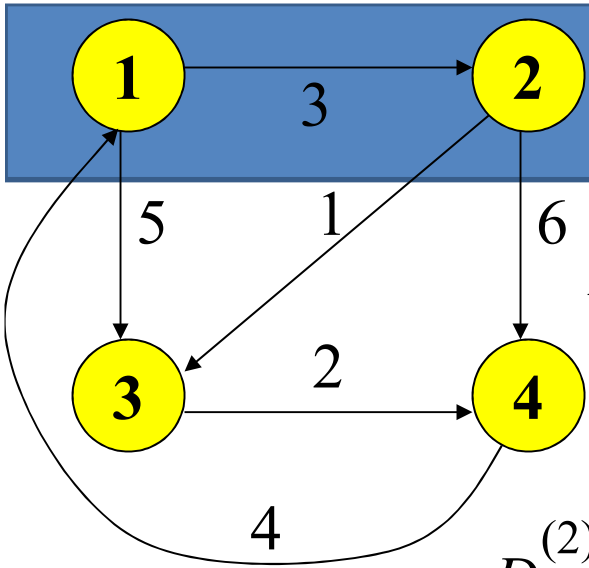
$$d_{ij}^{(k)} \leftarrow \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)})$$

$$D^{(1)} \begin{pmatrix} 0 & 3 & 5 & \infty \\ \infty & 0 & 1 & 6 \\ \infty & \infty & 0 & 2 \\ 4 & \mathbf{7} & \mathbf{9} & 0 \end{pmatrix} \qquad P^{(1)} \begin{pmatrix} \text{Nil} & 1 & 1 & \text{Nil} \\ \text{Nil} & \text{Nil} & 2 & 2 \\ \text{Nil} & \text{Nil} & \text{Nil} & 3 \\ 4 & \mathbf{1} & \mathbf{1} & \text{Nil} \end{pmatrix}$$

Could use 1 , 2 as intermediate vertex

$= \min(5,3+1)$ $\quad = \min(\infty,3+6)$

$$D^{(2)} \begin{pmatrix} 0 & 3 & \mathbf{4} & \mathbf{9} \\ \infty & 0 & 1 & 6 \\ \infty & \infty & 0 & 2 \\ 4 & 7 & \mathbf{8} & 0 \end{pmatrix} \qquad P^{(2)} \begin{pmatrix} \text{Nil} & 1 & \mathbf{2} & \mathbf{2} \\ \text{Nil} & \text{Nil} & 2 & 2 \\ \text{Nil} & \text{Nil} & \text{Nil} & 3 \\ 4 & 1 & \mathbf{2} & \text{Nil} \end{pmatrix}$$



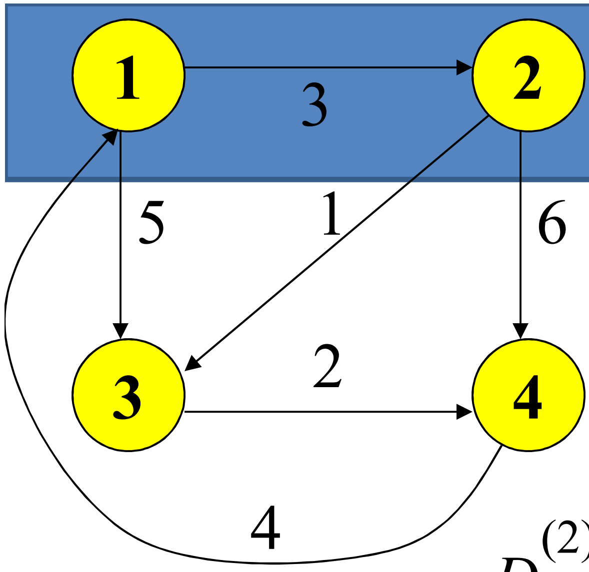
$$d_{ij}^{(k)} \leftarrow \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)})$$

$$D^{(1)} \begin{pmatrix} 0 & 3 & 5 & \infty \\ \infty & 0 & 1 & 6 \\ \infty & \infty & 0 & 2 \\ 4 & 7 & 9 & 0 \end{pmatrix} \quad P^{(1)} \begin{pmatrix} \text{Nil} & 1 & 1 & \text{Nil} \\ \text{Nil} & \text{Nil} & 2 & 2 \\ \text{Nil} & \text{Nil} & \text{Nil} & 3 \\ 4 & 1 & 1 & \text{Nil} \end{pmatrix}$$

$$D^{(2)} \begin{pmatrix} 0 & 3 & 4 & 9 \\ \infty & 0 & 1 & 6 \\ \infty & \infty & 0 & 2 \\ 4 & 7 & 8 & 0 \end{pmatrix} \quad P^{(2)} \begin{pmatrix} \text{Nil} & 1 & 2 & 2 \\ \text{Nil} & \text{Nil} & 2 & 2 \\ \text{Nil} & \text{Nil} & \text{Nil} & 3 \\ 4 & 1 & 2 & \text{Nil} \end{pmatrix}$$

$$D^{(3)} \begin{pmatrix} 0 & 3 & 4 & 6 \\ \infty & 0 & 1 & 3 \\ \infty & \infty & 0 & 2 \\ 4 & 7 & 8 & 0 \end{pmatrix} \quad P^{(3)} \begin{pmatrix} \text{Nil} & 1 & 2 & 3 \\ \text{Nil} & \text{Nil} & 2 & 3 \\ \text{Nil} & \text{Nil} & \text{Nil} & 3 \\ 4 & 1 & 2 & \text{Nil} \end{pmatrix}$$

$$D^{(4)} \begin{pmatrix} 0 & 3 & 4 & 6 \\ 7 & 0 & 1 & 3 \\ 6 & 9 & 0 & 2 \\ 4 & 7 & 8 & 0 \end{pmatrix} \quad P^{(4)} \begin{pmatrix} \text{Nil} & 1 & 2 & 3 \\ 4 & \text{Nil} & 2 & 3 \\ 4 & 1 & \text{Nil} & 3 \\ 4 & 1 & 2 & \text{Nil} \end{pmatrix}$$



$$d_{ij}^{(k)} \leftarrow \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)})$$

Conclusion: shortest path between every pair of vertices ???

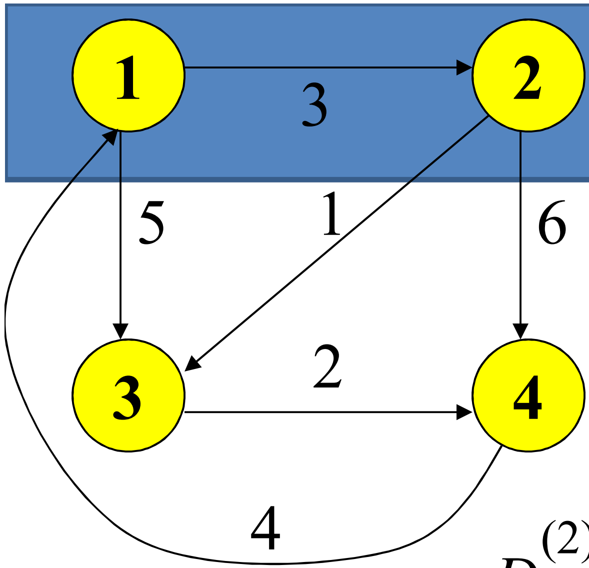
Shortest path from 3 to 2: $3 \rightarrow 4 \rightarrow 1 \rightarrow 2$

Length = 9

$$D^{(2)} \begin{pmatrix} 0 & 3 & 4 & 9 \\ \infty & 0 & 1 & 6 \\ \infty & \infty & 0 & 2 \\ 4 & 7 & 8 & 0 \end{pmatrix} \qquad P^{(2)} \begin{pmatrix} \text{Nil} & 1 & 2 & 2 \\ \text{Nil} & \text{Nil} & 2 & 2 \\ \text{Nil} & \text{Nil} & \text{Nil} & 3 \\ 4 & 1 & 2 & \text{Nil} \end{pmatrix}$$

$$D^{(3)} \begin{pmatrix} 0 & 3 & 4 & 6 \\ \infty & 0 & 1 & 3 \\ \infty & \infty & 0 & 2 \\ 4 & 7 & 8 & 0 \end{pmatrix} \qquad P^{(3)} \begin{pmatrix} \text{Nil} & 1 & 2 & 3 \\ \text{Nil} & \text{Nil} & 2 & 3 \\ \text{Nil} & \text{Nil} & \text{Nil} & 3 \\ 4 & 1 & 2 & \text{Nil} \end{pmatrix}$$

$$D^{(4)} \begin{pmatrix} 0 & 3 & 4 & 6 \\ 7 & 0 & 1 & 3 \\ 6 & 9 & 0 & 2 \\ 4 & 7 & 8 & 0 \end{pmatrix} \qquad P^{(4)} \begin{pmatrix} \text{Nil} & 1 & 2 & 3 \\ 4 & \text{Nil} & 2 & 3 \\ 4 & 1 & \text{Nil} & 3 \\ 4 & 1 & 2 & \text{Nil} \end{pmatrix}$$



$$d_{ij}^{(k)} \leftarrow \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)})$$

Shortest path from 1 to 4: $1 \to 2 \to 3 \to 4$

Length = 6

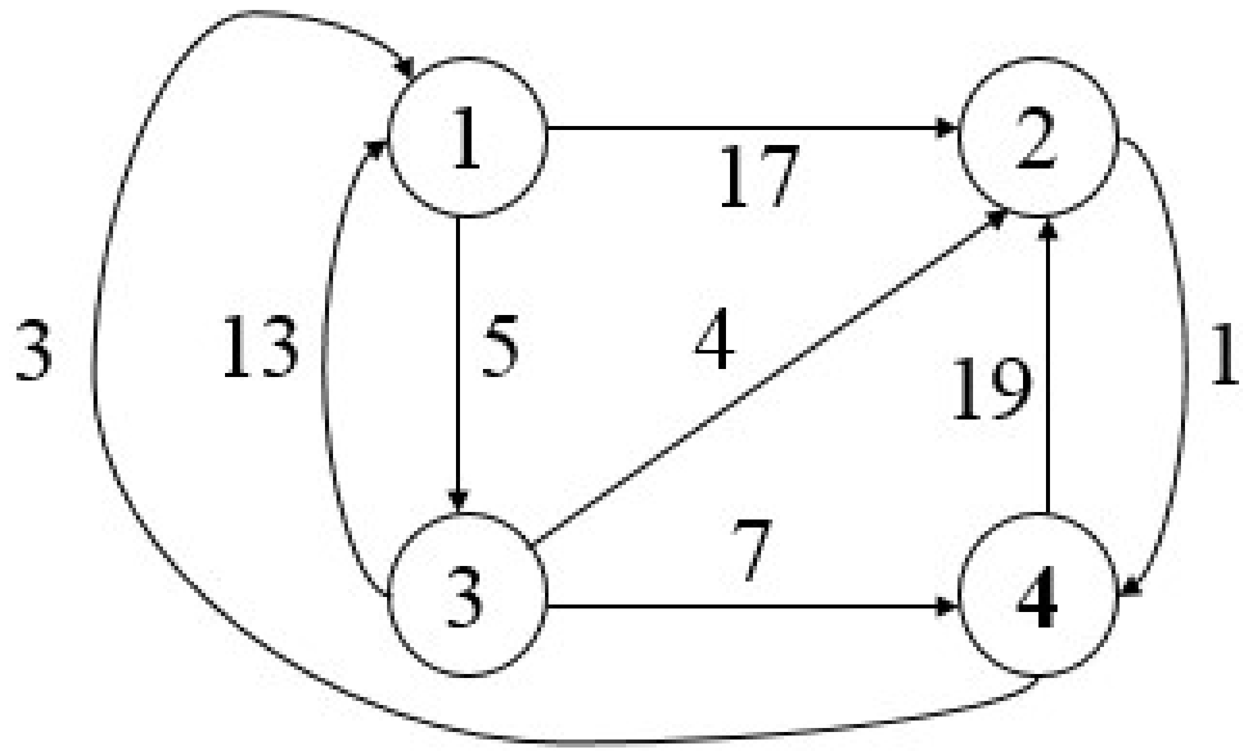
$$D^{(2)} \begin{pmatrix} 0 & 3 & 4 & 9 \\ \infty & 0 & 1 & 6 \\ \infty & \infty & 0 & 2 \\ 4 & 7 & 8 & 0 \end{pmatrix} \qquad P^{(2)} \begin{pmatrix} \text{Nil} & 1 & 2 & 2 \\ \text{Nil} & \text{Nil} & 2 & 2 \\ \text{Nil} & \text{Nil} & \text{Nil} & 3 \\ 4 & 1 & 2 & \text{Nil} \end{pmatrix}$$

$$D^{(3)} \begin{pmatrix} 0 & 3 & 4 & 6 \\ \infty & 0 & 1 & 3 \\ \infty & \infty & 0 & 2 \\ 4 & 7 & 8 & 0 \end{pmatrix} \qquad P^{(3)} \begin{pmatrix} \text{Nil} & 1 & 2 & 3 \\ \text{Nil} & \text{Nil} & 2 & 3 \\ \text{Nil} & \text{Nil} & \text{Nil} & 3 \\ 4 & 1 & 2 & \text{Nil} \end{pmatrix}$$

$$D^{(4)} \begin{pmatrix} 0 & 3 & 4 & 6 \\ 7 & 0 & 1 & 3 \\ 6 & 9 & 0 & 2 \\ 4 & 7 & 8 & 0 \end{pmatrix} \qquad P^{(4)} \begin{pmatrix} \text{Nil} & 1 & 2 & 3 \\ 4 & \text{Nil} & 2 & 3 \\ 4 & 1 & \text{Nil} & 3 \\ 4 & 1 & 2 & \text{Nil} \end{pmatrix}$$

Example:

Apply Floyd-Warshall algorithm to find the shortest paths between all pairs of vertices of the graph.



Shortest path variants