

# Lecture 8: Graph Algorithms (II)

Dr. Tsung-Wei Huang

Department of Electrical and Computer Engineering  
University of Utah, Salt Lake City, UT



# Graph Traversal

---

## ❑ Depth First Search (DFS)

### ❑ Stack

- ❑ The algorithm starts at the root node (selecting some arbitrary node as the root node in the case of a graph) and explores as far as possible along each branch before backtracking.

## ❑ Breadth First Search (BFS)

### ❑ Queue

- ❑ The algorithm starts at the root node, and explores all of the neighbor nodes at the present depth prior to moving on to the nodes at the next depth level.

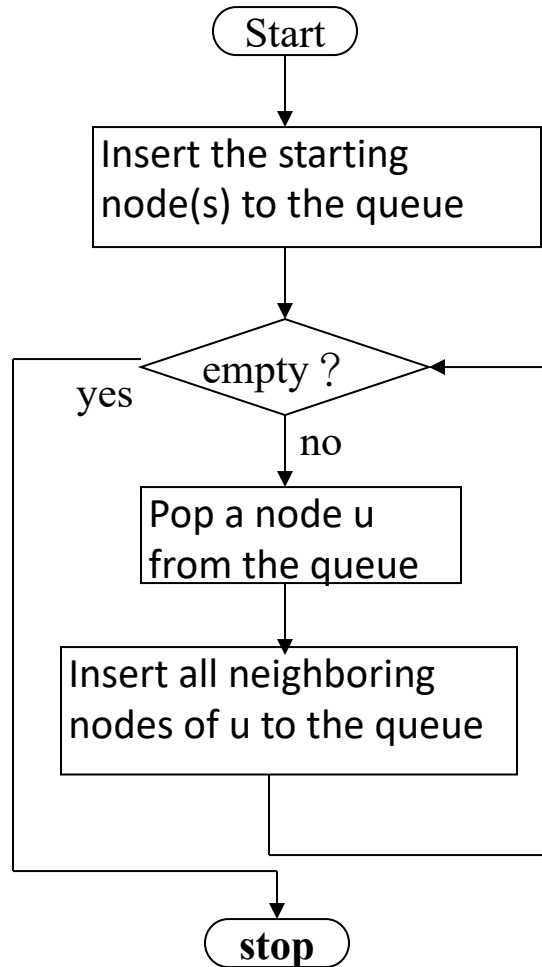
# DFS Algorithm

---

```
procedure DFS(G, v) is  
  label v as discovered  
  for all directed edges from v to w that are in G.adjacentEdges(v) do  
    if vertex w is not labeled as discovered then  
      recursively call DFS(G, w)
```

# BFS Algorithm

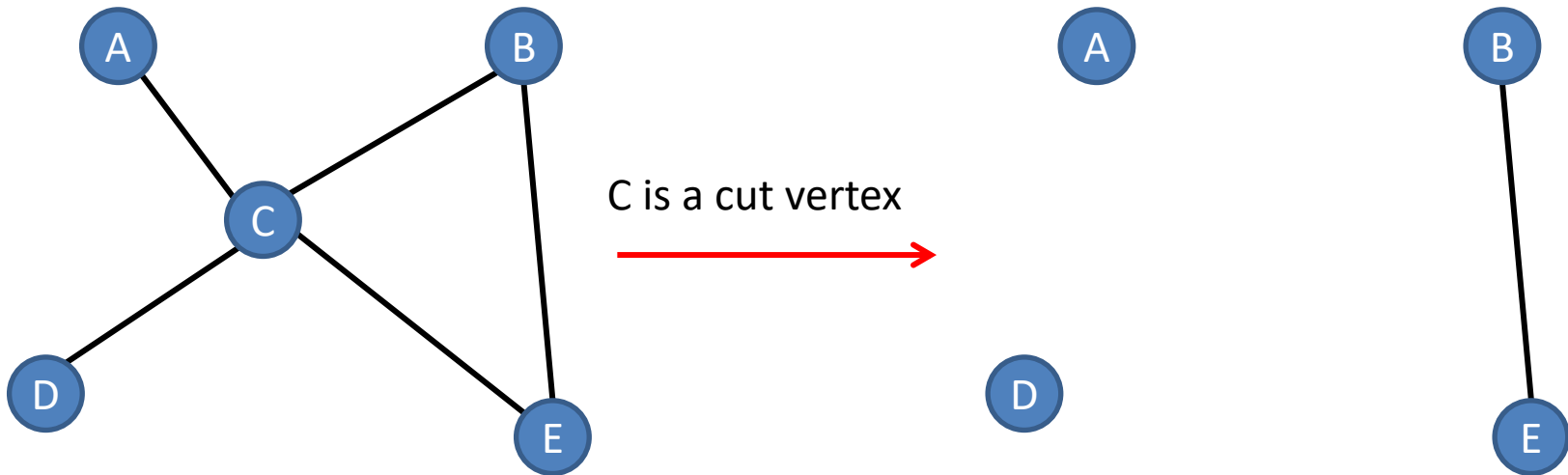
---



# Cut Vertex (Articulation Point)

## ❑ Formal definition

- ❑ A cut vertex or articulation point is a vertex in a graph such that removal of the vertex causes an increase in the number of connected components.
- ❑ If the graph was connected before the removal of the vertex, it will be disconnected afterwards.



# How to Find Cut Vertices?

---

## ☐ Brute force?

### ☐ Enumerate all vertices $O(N)$

- Try remove a vertex from the graph
- Perform union-and-find algorithms to find the number of sets
- If the number of disjoint sets increases, the vertex is a cut

### ☐ Union-and-Find takes $O(N\log N)$

### ☐ Total time complexity is $O(N^2\log N)$

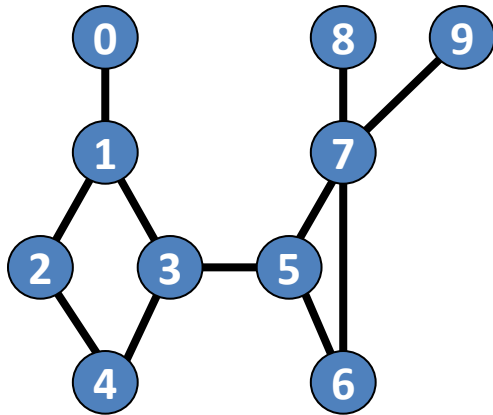
# DFS can Do this for us

---

## ☐ We have two edge type during DFS

☐ Forward edge  $u \rightarrow v$ ,  $v$  is not visited

☐ Backward edge  $u \rightarrow v$ ,  $v$  is visited



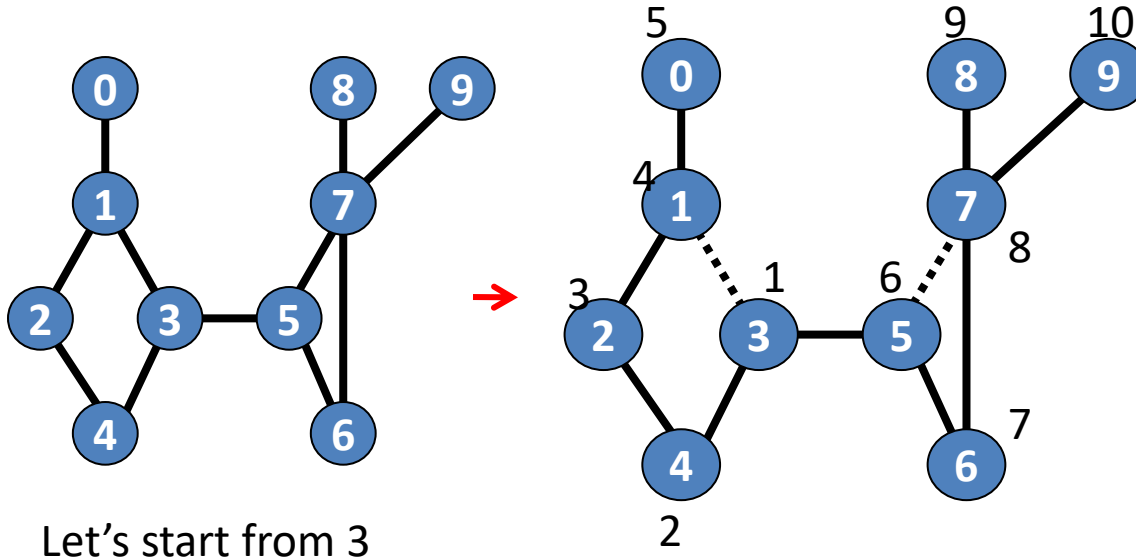
Let's start from 3

# DFS can Do this for us

## □ We have two edge type during DFS

□ Forward edge  $u \rightarrow v$ ,  $v$  is not visited

□ Backward edge  $u \rightarrow v$ ,  $v$  is visited



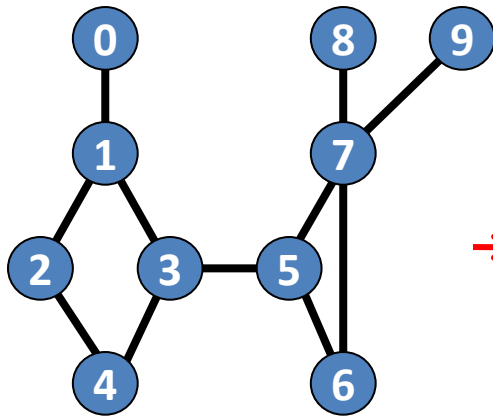
Label the order of traversal  
in a linear array "dfn"



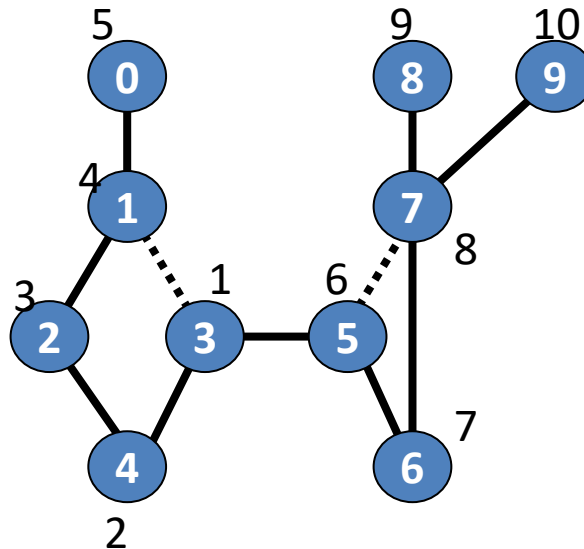
# DFS can Do this for us

## □ We have two edge type during DFS

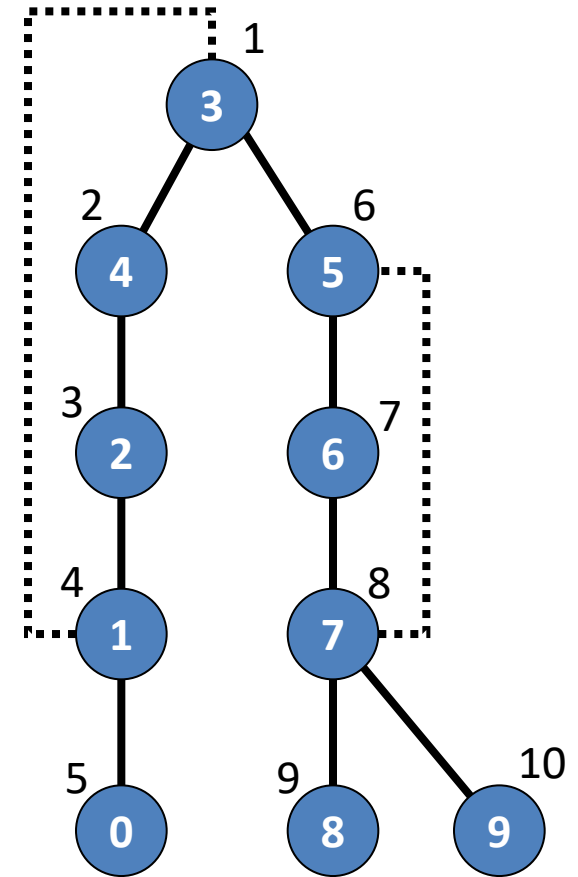
- Forward edge  $u \rightarrow v$ ,  $v$  is not visited
- Backward edge  $u \rightarrow v$ ,  $v$  is visited



Let's start from 3



Label the order of traversal  
in a linear array "dfn"



DFS gives us a tree order of  
vertices

# Cut Vertex Property

---

## ❑ Observation

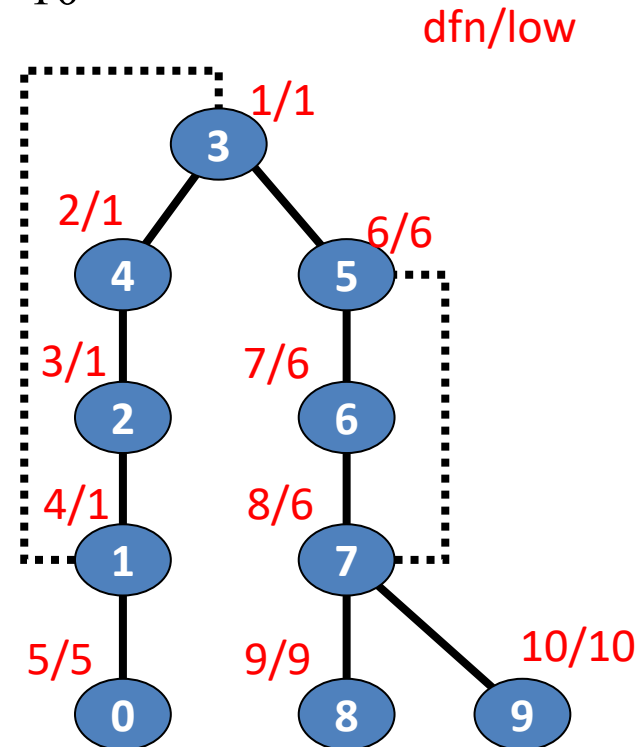
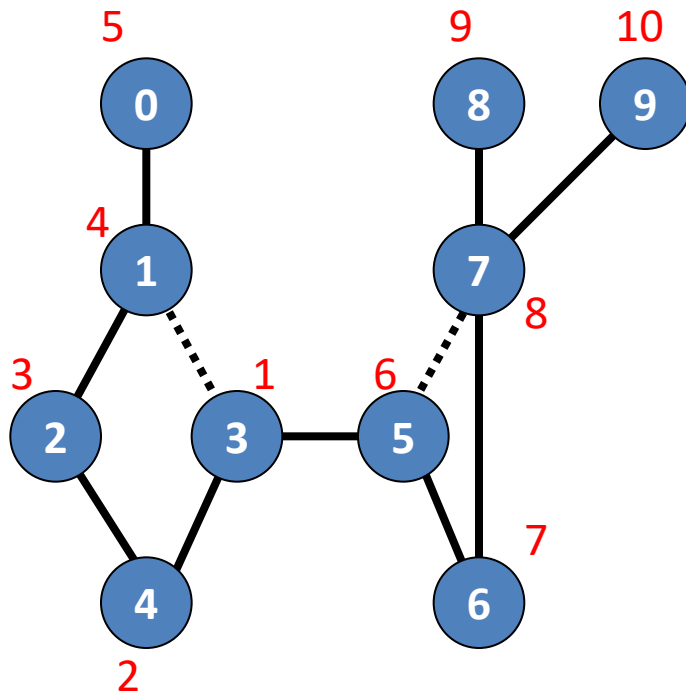
- ❑ If root has two children, => the root is a cut vertex
- ❑ If a vertex  $u$  has a child  $v$  such that  $v$  can't go back to  $u$ 's parent =>  $u$  is a cut vertex
  - Assume no duplicate edges between vertices

## ❑ Let's quantify this

- ❑  $\text{low}(u)$ : the minimum dfn value  $u$  can reach
  - $\min\{\text{low}(u), \text{low}(v)\}$ , foreach edge  $u \rightarrow v$

# Example of $dfn[i]$ and $low[i]$

<i>Vertex</i>	0	1	2	3	4	5	6	7	8	9
<i>dfn</i>	5	4	3	1	2	6	7	8	9	10
<i>low</i>	5	1	1	1	1	6	6	6	9	10



# Algorithm

```
void dfs(int u) {
    visited[u]=true;
    low[u]=dfn[u]=_____ ;
    int child=0;
    for(int i=0; i<adj[u].size(); i++) {
        int v=adj[u][i];
        if(visited[v]==false) {
            child++;
            parent[v]=u;
            dfs(v);
            low[u]=min(low[u], low[v]);
            if _____
                cut[u]=true;
            else if(v!=parent[u]) // backward edge
                _____
        }
    }
}
```

Initialization:

times=0, parent[i]=-1, cut[i]=0, visited[i]=0

## □ Observation

- If root has two children, => the root is a cut vertex
- If a vertex  $u$  has a child  $v$  such that  $v$  can't go back to  $u$ 's parent =>  $u$  is a cut vertex
  - Assume no duplicate edges between vertices

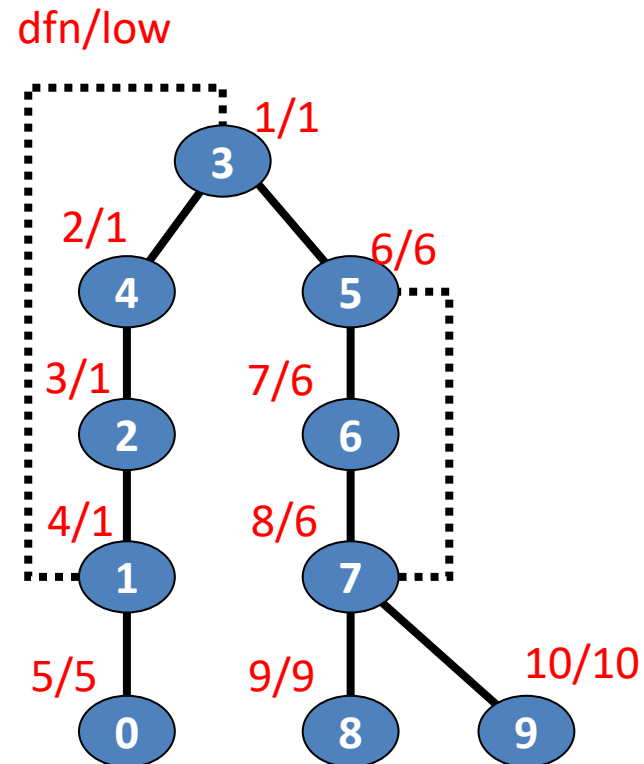
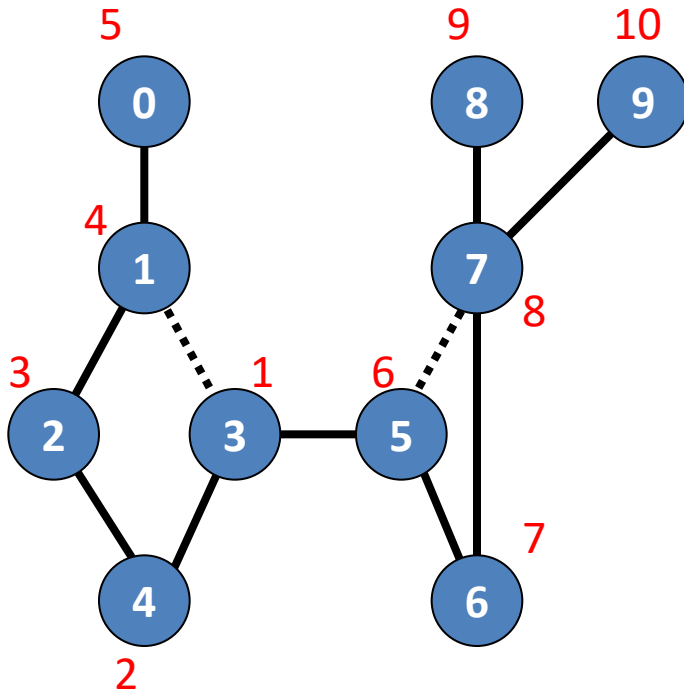
## □ Let's quantify this

- $low(u)$ : the minimum dfn value  $u$  can reach
  - $\min\{low(u), low(v)\}$ , foreach edge  $u \rightarrow v$

# Cut Edge

## □ Observation

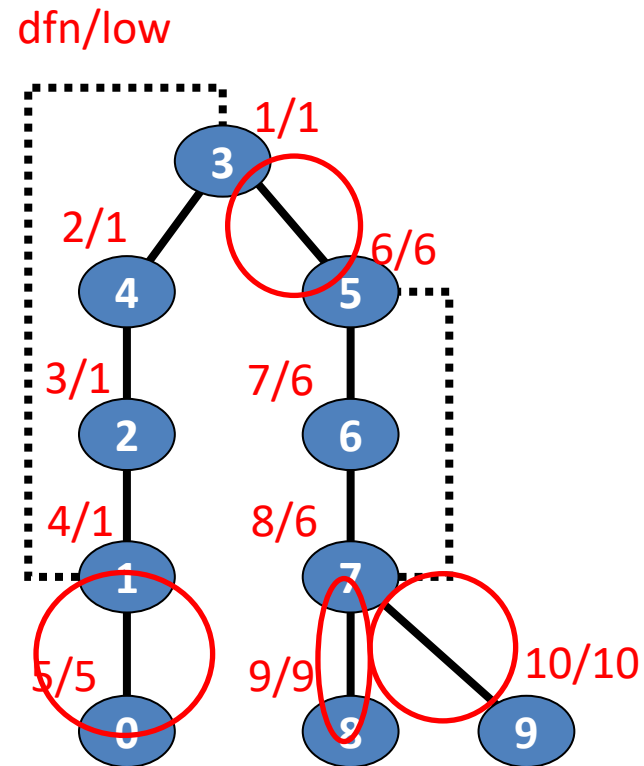
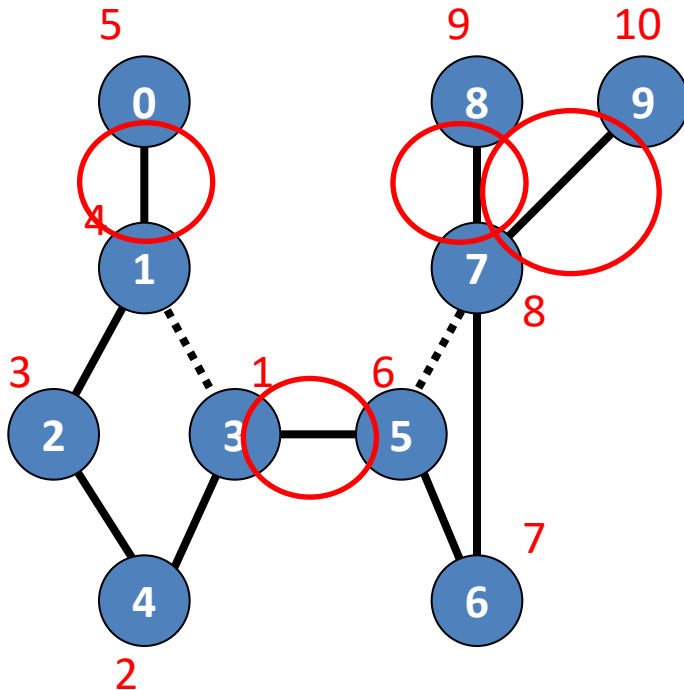
□ For each edge  $u \rightarrow v$ , if  $\text{low}[v] > \text{dfn}[u] \Rightarrow$  **cut edge**



# Cut Edge

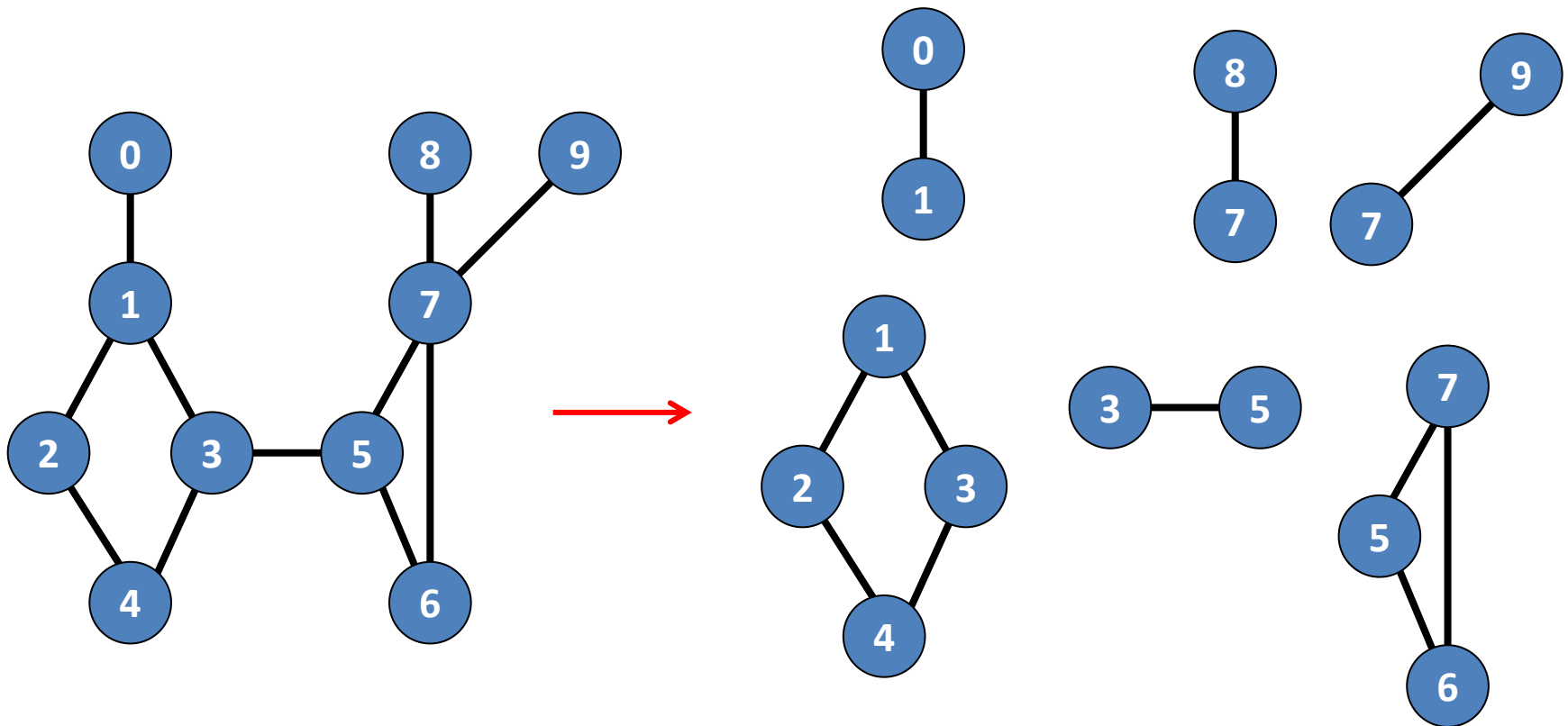
## □ Observation

□ For each edge  $u \rightarrow v$ , if  $\text{low}[v] > \text{dfn}[u] \Rightarrow$  **cut edge**



# Bi-connected Components

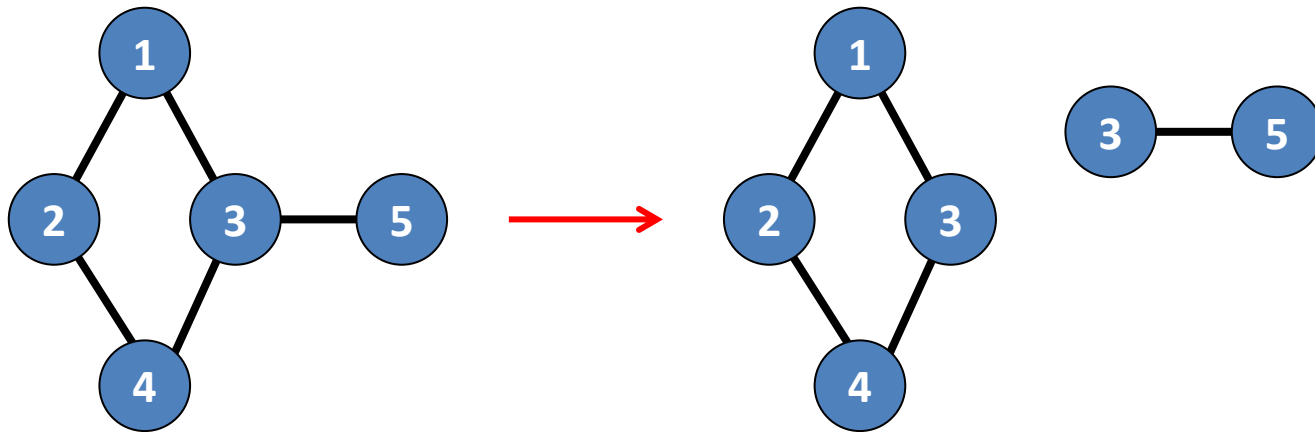
□ A connected graph with NO cut vertices



six bi-connected components

# Let's Use a Simple Example first ...

- Apparently, we have two bi-connected components
  - Cut vertices exist in the boundary between components
    - otherwise, it's not connected by contradiction

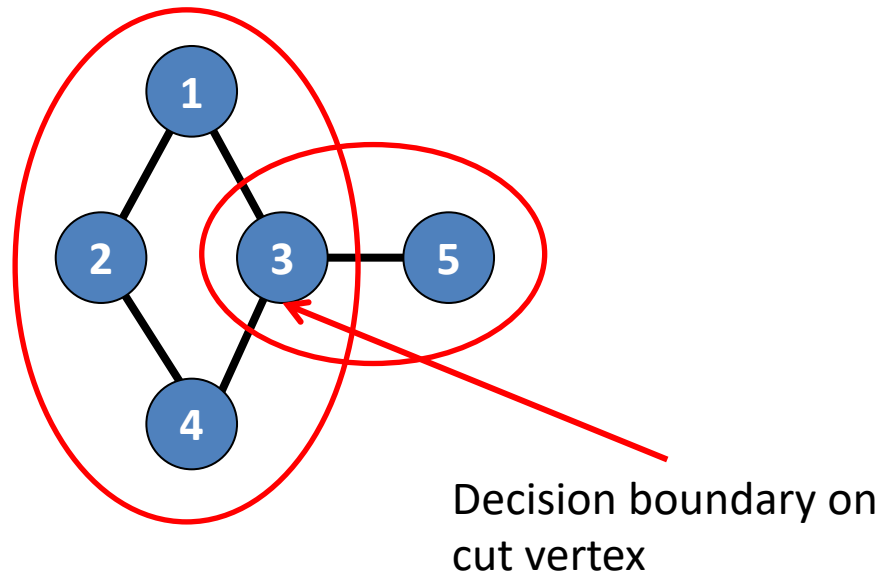


two bi-connected components



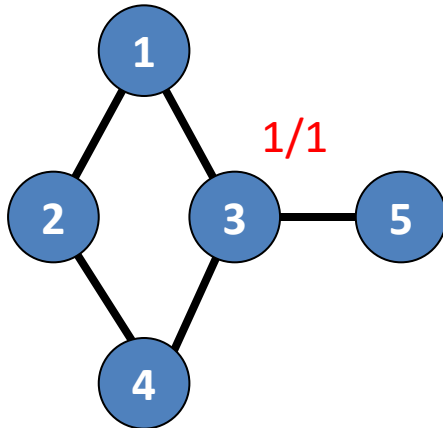
# Algorithms

- ❑ **Maintain a stack of traversed edges**
  - ❑ This records the “trace” of our traversal
- ❑ **When we find a cut vertex**
  - ❑ Pop all edges from the stack until the decision boundary



# Example

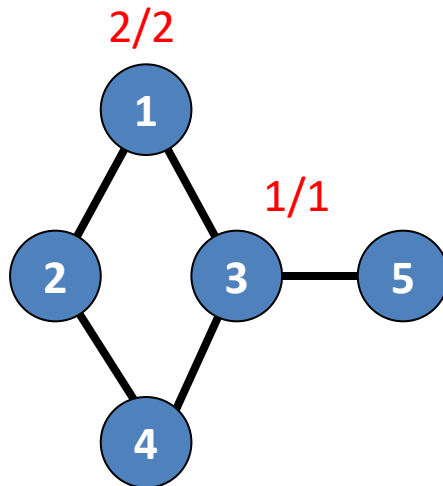
dfn/low



-1→3					
------	--	--	--	--	--

# Example

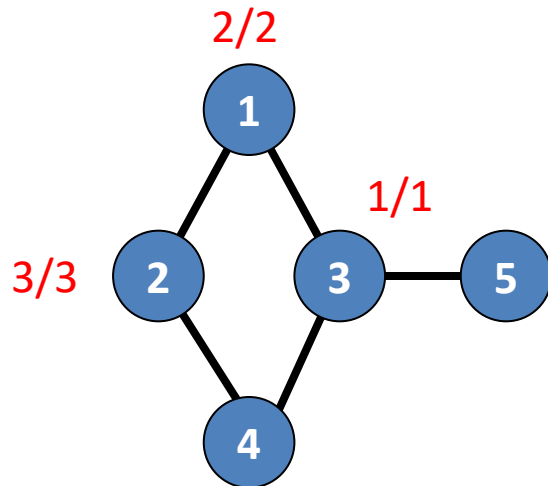
dfn/low



-1→3	3→1				
------	-----	--	--	--	--

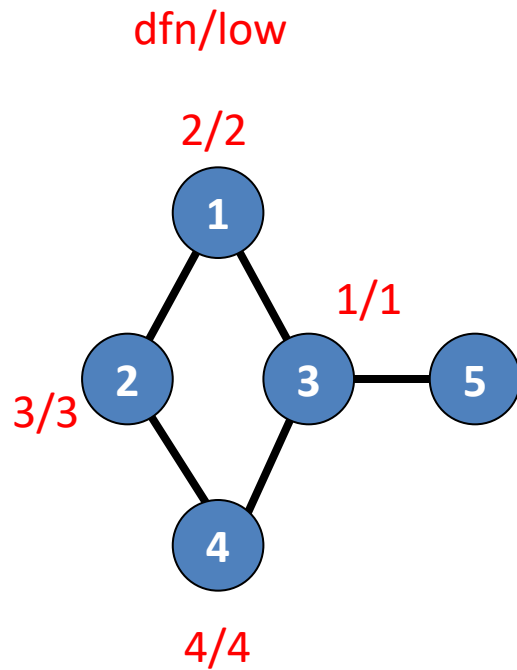
# Example

dfn/low



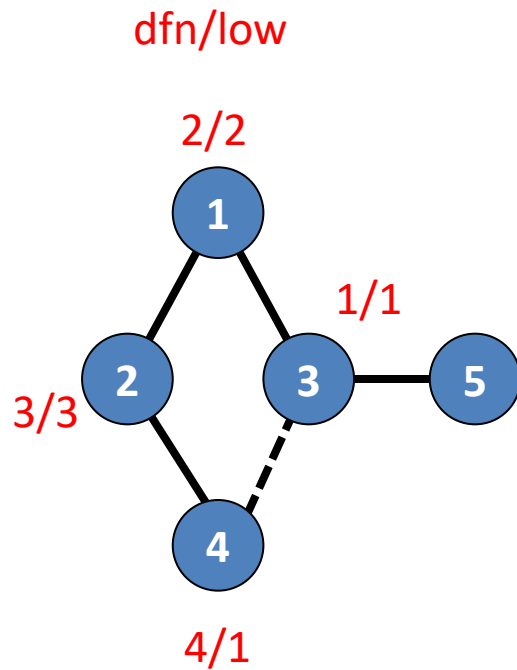
-1→3	3→1	1→2			
------	-----	-----	--	--	--

# Example



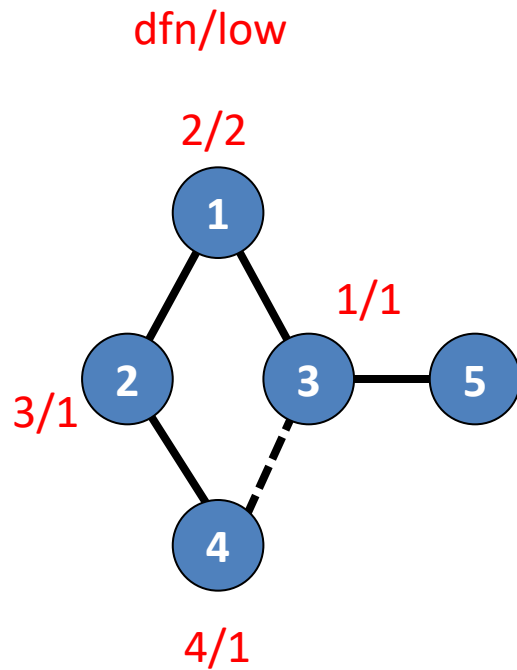
-1→3	3→1	1→2	2→4		
------	-----	-----	-----	--	--

# Example



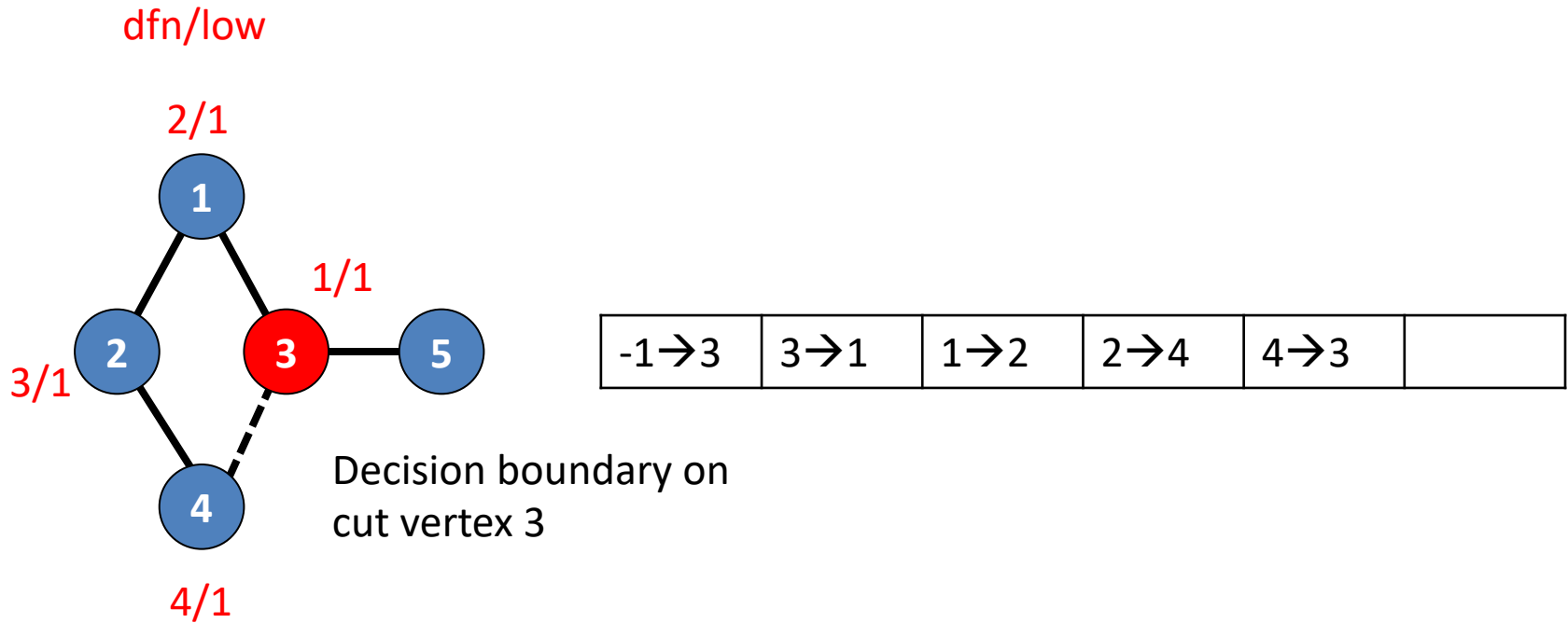
-1→3	3→1	1→2	2→4	4→3	
------	-----	-----	-----	-----	--

# Example



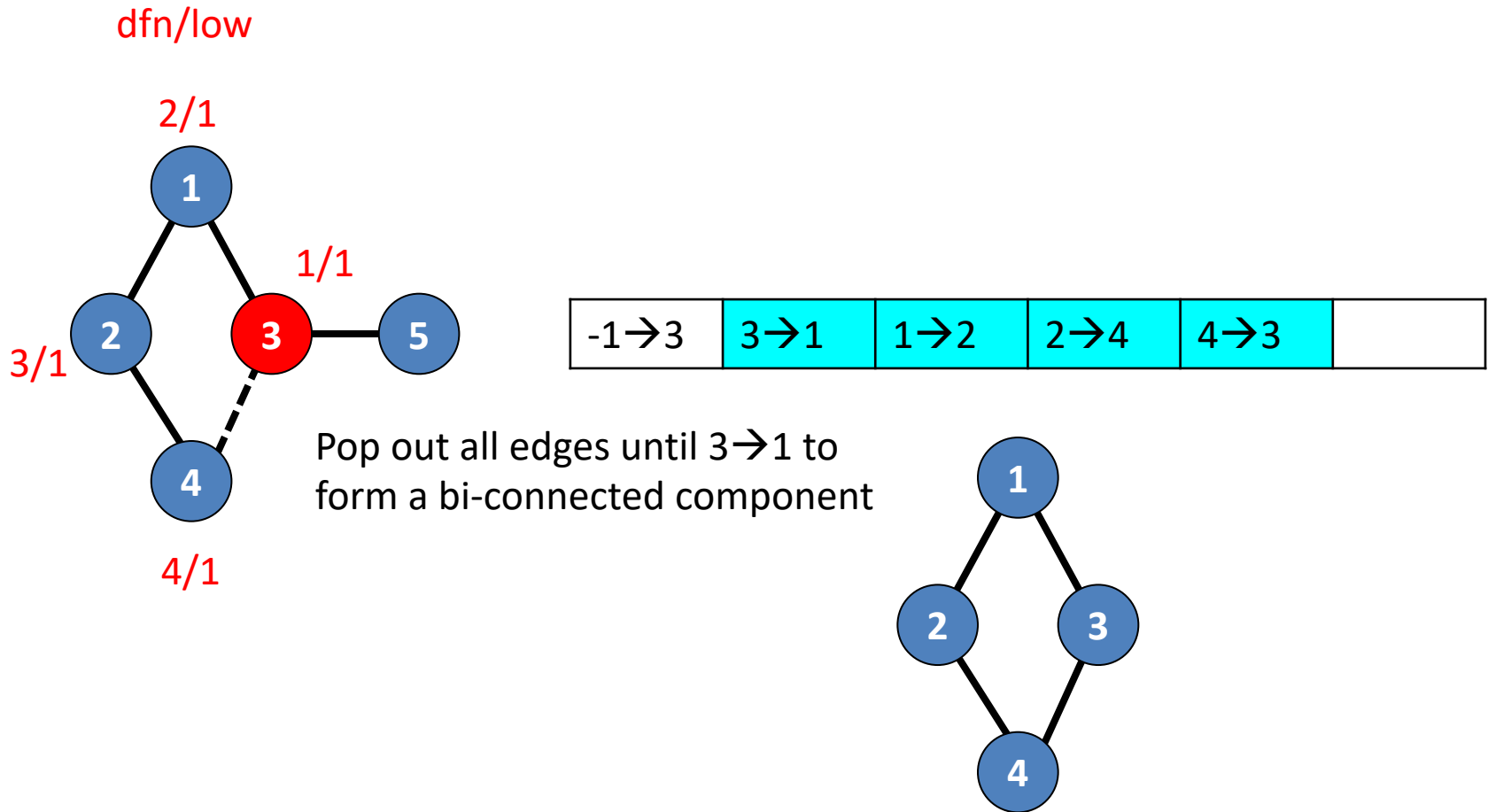
-1→3	3→1	1→2	2→4	4→3	
------	-----	-----	-----	-----	--

# Example





# Example



# Implementation Details

```
for (ptr = graph[u]; ptr; ptr = ptr→link) {
    w = ptr→vertex;
    if (v != w && dfn[w] < dfn[u])
        push(u,w); /* add edge to stack */
    if (dfn[w] < 0) { /* w has not been visited */
        bicon(w,u);
        low[u] = MIN2(low[u],low[w]);
        if (low[w] >= dfn[u]) {
            printf("New biconnected component: ");
            do { /* delete edge from stack */
                pop(&x, &y);
                printf(" <%d,%d>",x,y);
            } while (!(x == u) && (y == w));
            printf("\n");
        }
    }
    else if (w != v) low[u] = MIN2(low[u],dfn[w]);
}
```

**Cut vertex found!**

**Backward edge**



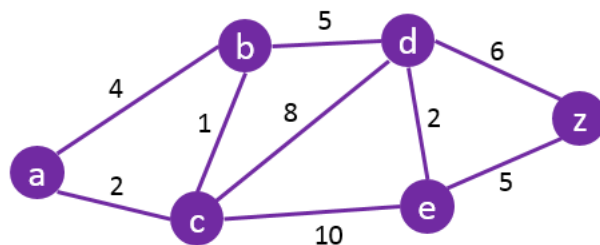
# Shortest Path Algorithms

## ❑ Single source all destinations

- ❑ Given a weighted directed graph  $G$ , find the minimum-weight path from a given source vertex  $s$  to all other vertices
- ❑ “Shortest-path” = minimum summation weight
  - Weight of path is sum of edges

## ❑ Tremendous applications

- ❑ Map: what is the shortest path from SLC to CA?
- ❑ Circuit design: what is the minimum interconnect?



What is the shortest path from a to z?

# Shortest Path Property

---

## ❑ *Optimal substructure*

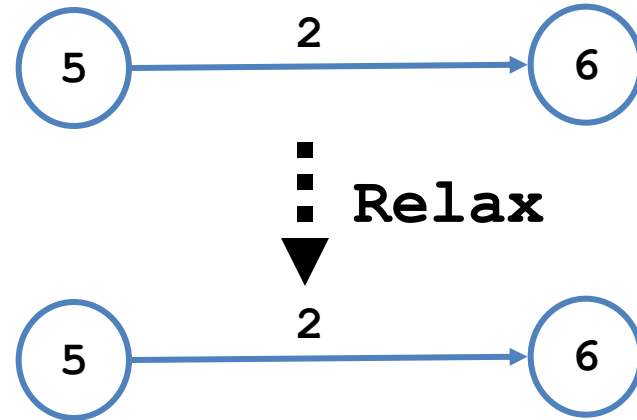
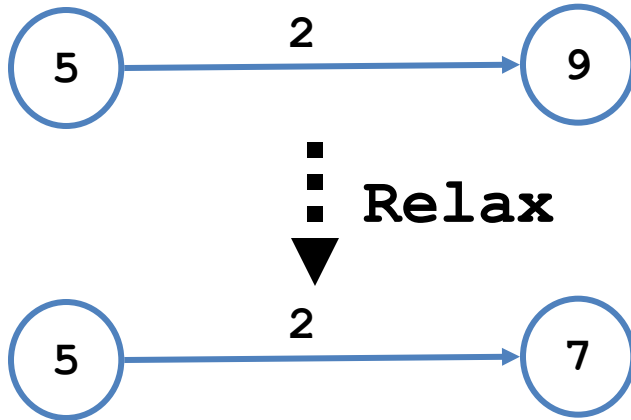
- ❑ The shortest path consists of shortest subpaths
- ❑ Easy to prove by contradiction
- ❑ **Let  $\delta(u,v)$  be the the shortest path from u to v:**
  - ❑ Shortest paths satisfy the *triangle inequality*
    - $\delta(u,v) \leq \delta(u,x) + \delta(x,v)$
- ❑ **Shortest paths may not exist if graph contains:**
  - ❑ Non-reachable targets
  - ❑ Negative feedback loops/cycles

# Algorithm

## ❑ Key technique: *relaxation*

- ❑ Maintain upper bound  $d[v]$  on  $\delta(s,v)$ :

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



# Bellman Ford

```
BellmanFord()  
  for each  $v \in V$   
     $d[v] = \infty$ ;  
 $d[s] = 0$ ;  
  for  $i=1$  to  $|V|-1$   
    for each edge  $(u,v) \in E$   
      Relax( $u,v, w(u,v)$ );
```

} Initialize  $d[]$ , which will converge to shortest-path value  $\delta$

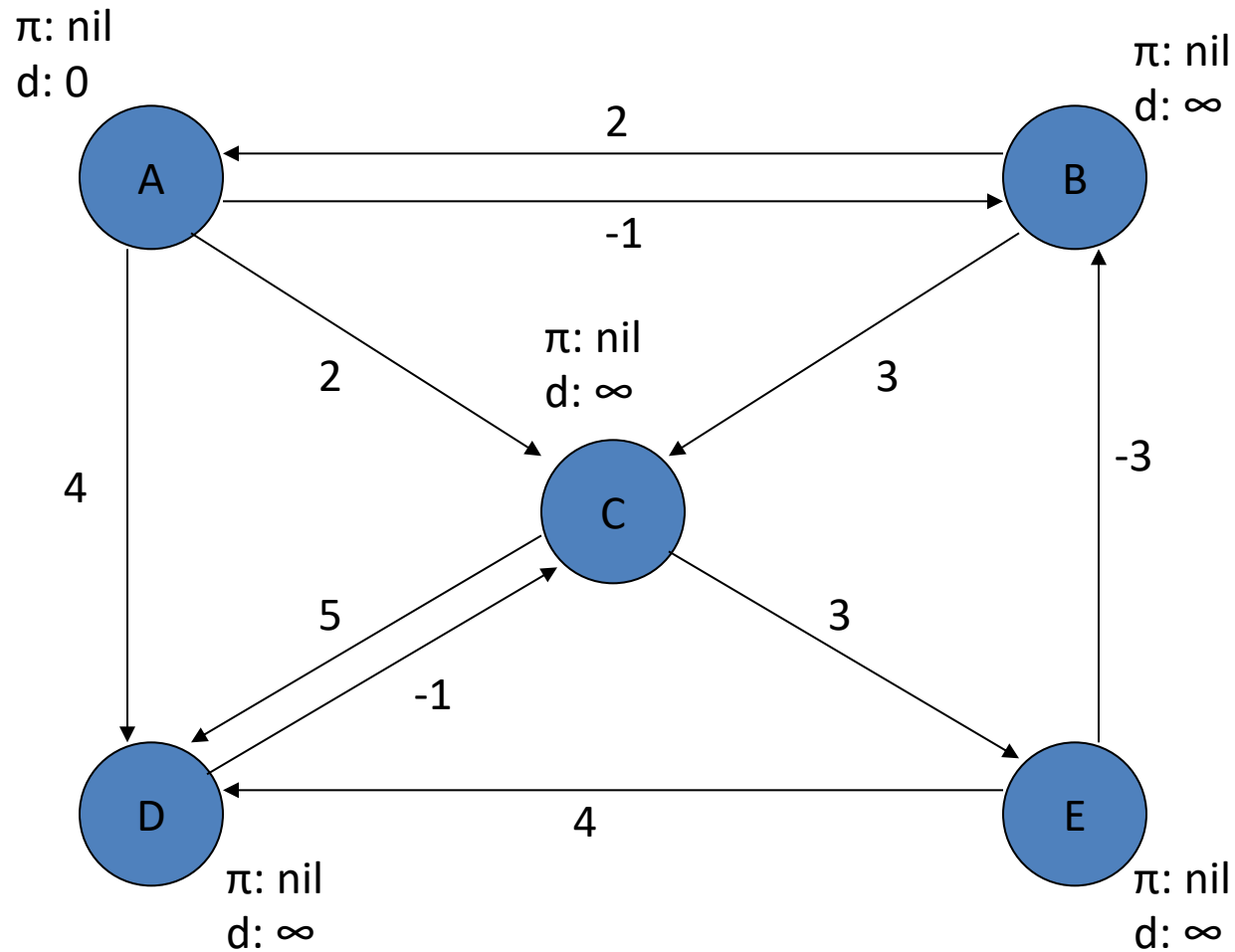
} Relaxation:  
Make  $|V|-1$  passes, relax each edge

Relax( $u,v,w$ ): if  $(d[v] > d[u]+w)$  then  $d[v]=d[u]+w$

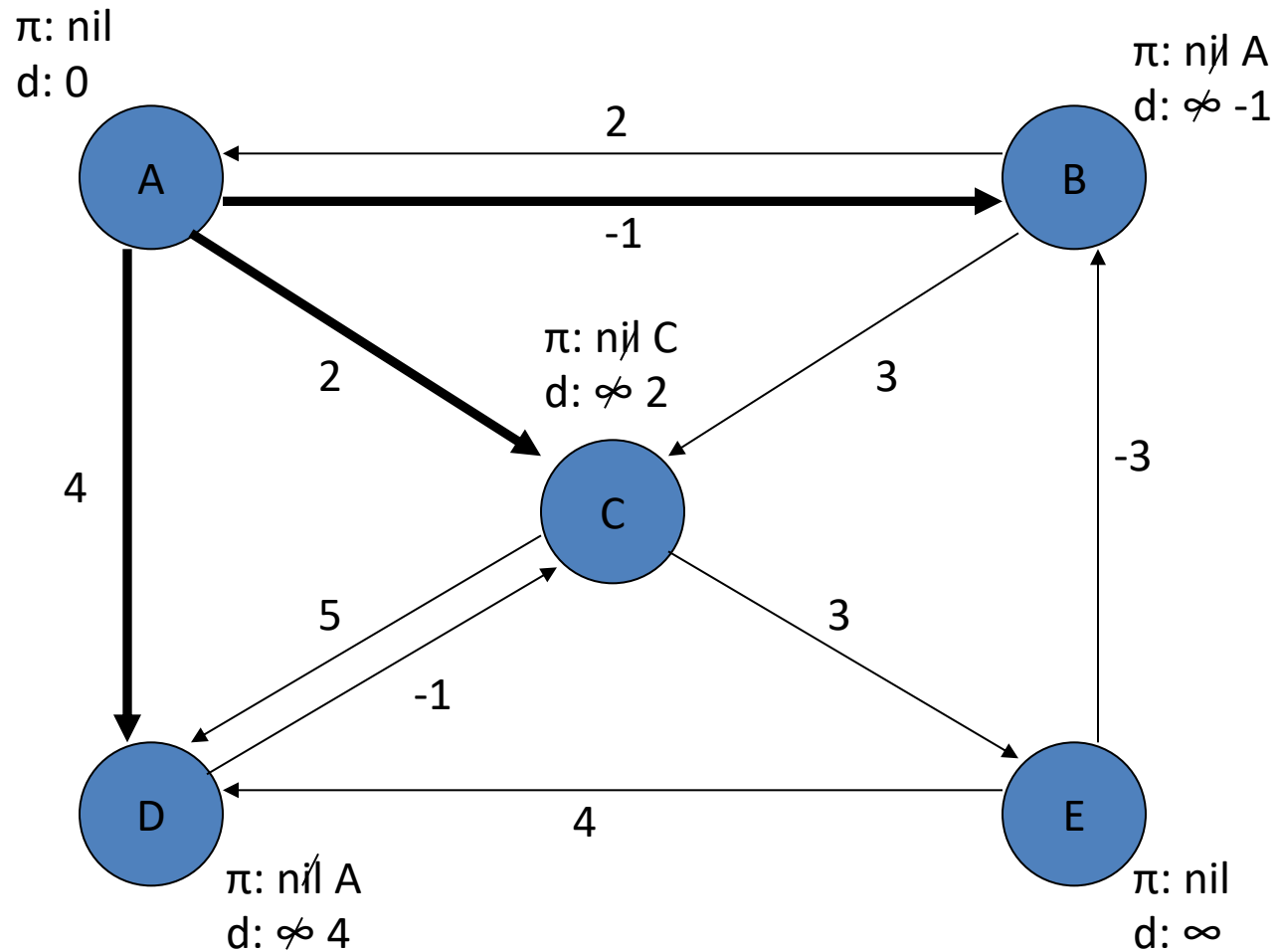
□ The simplest shortest path algorithm

# Bellman Ford

---

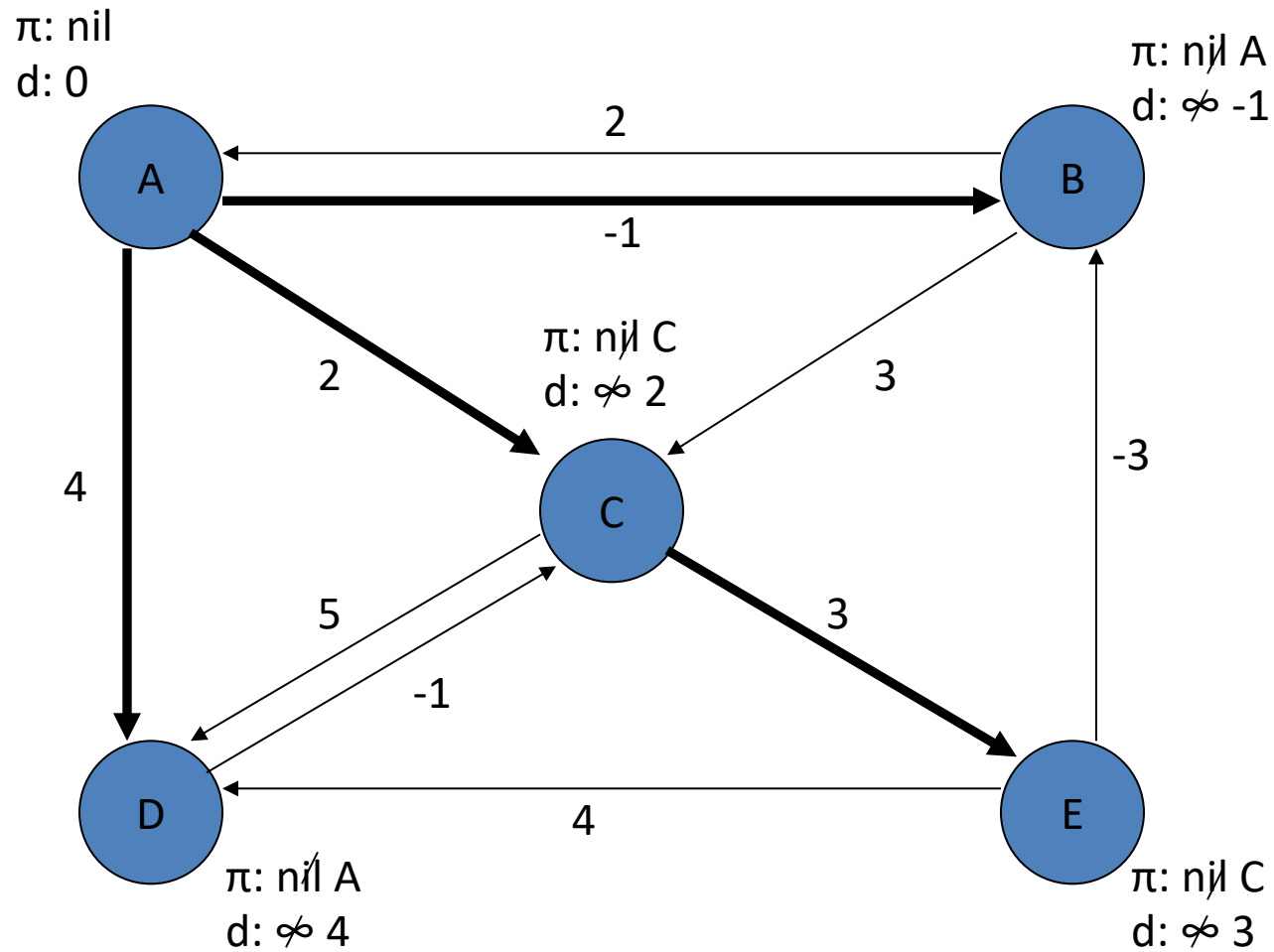


# Bellman Ford





# Bellman Ford



# Complexity

---

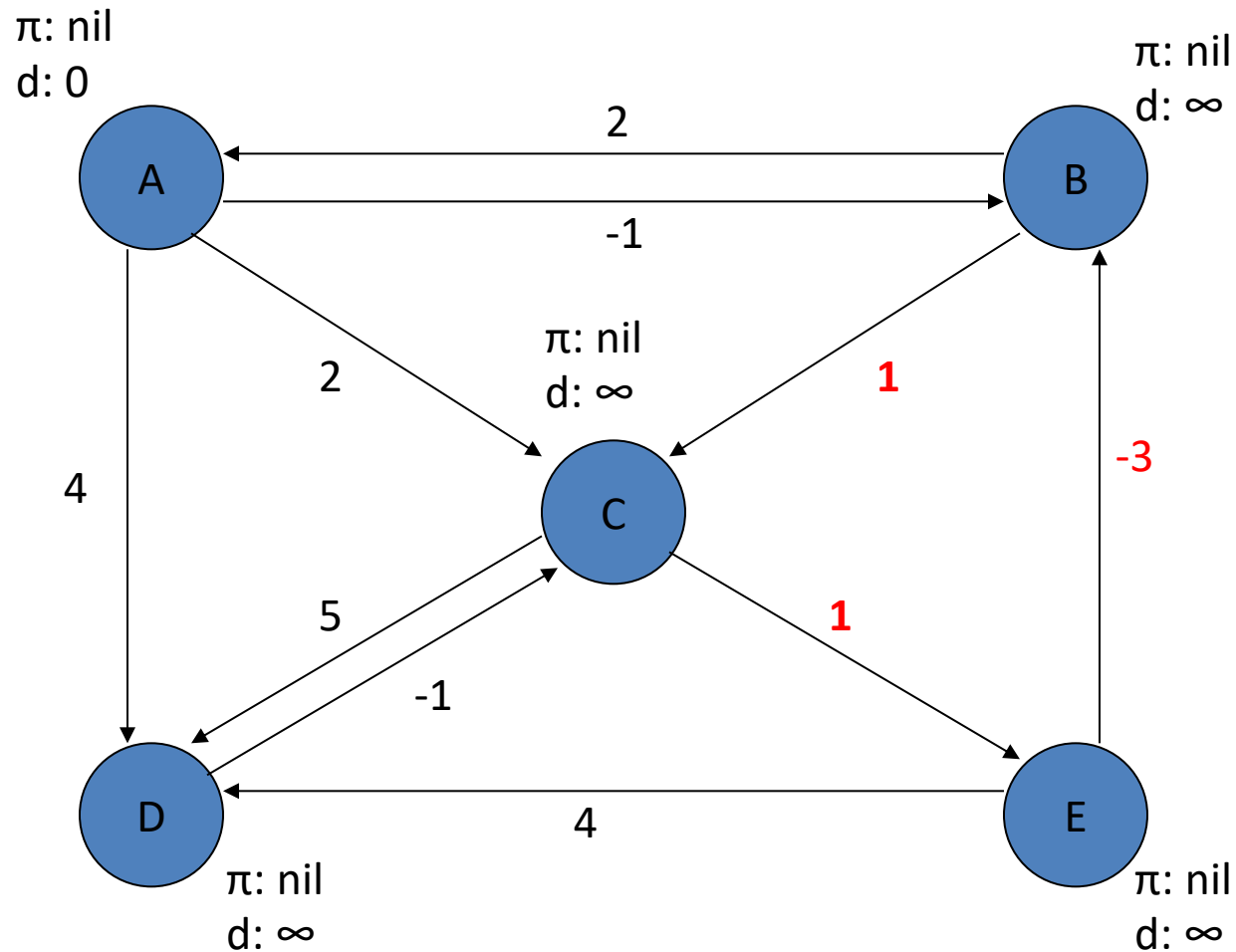
- ❑ **Running time:  $O(VE)$**

- ❑ Not so good for large dense graphs

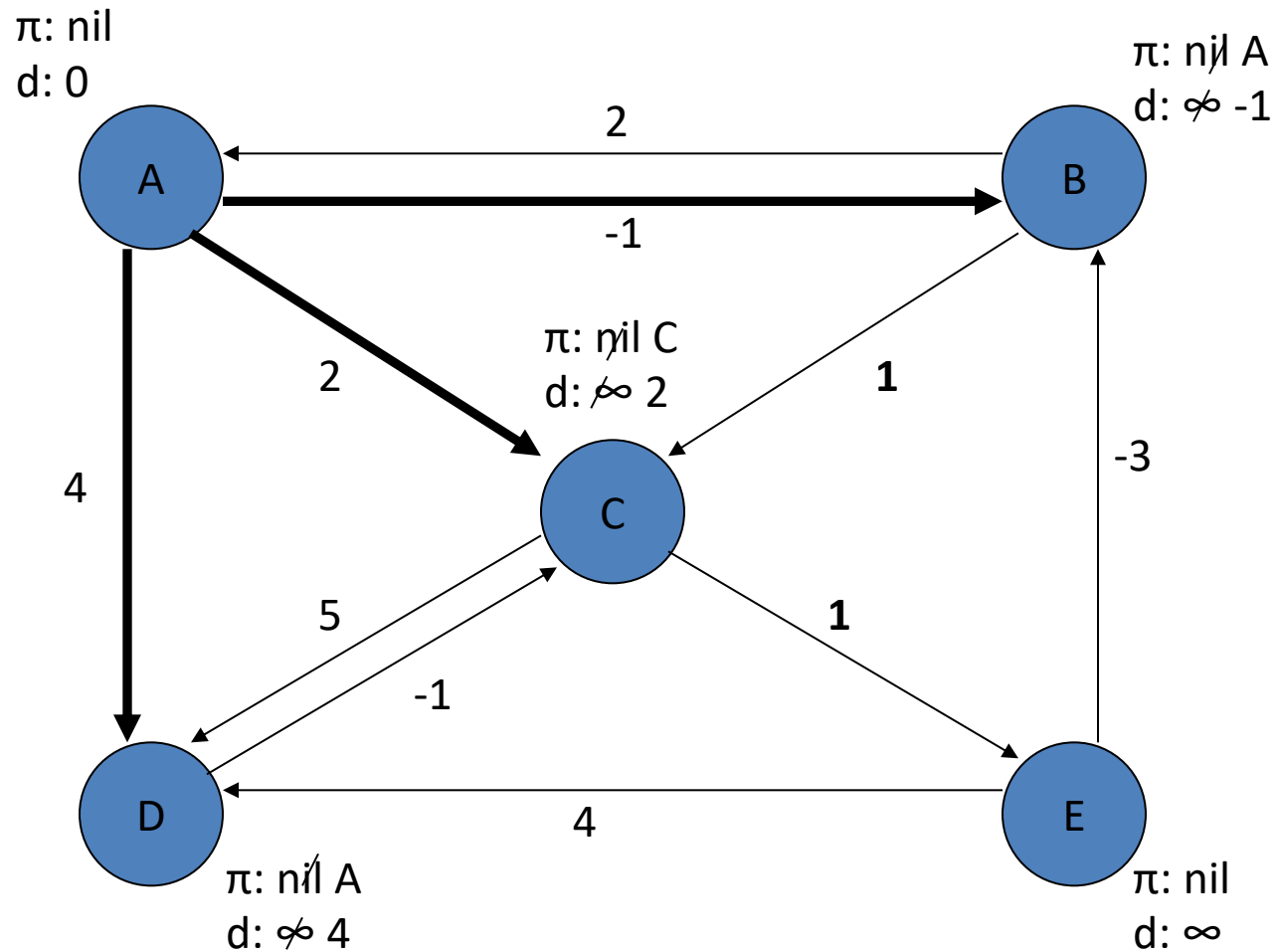
- ❑ But a very practical algorithm in many ways

- ❑ **What about graph with negative cycles ...?**

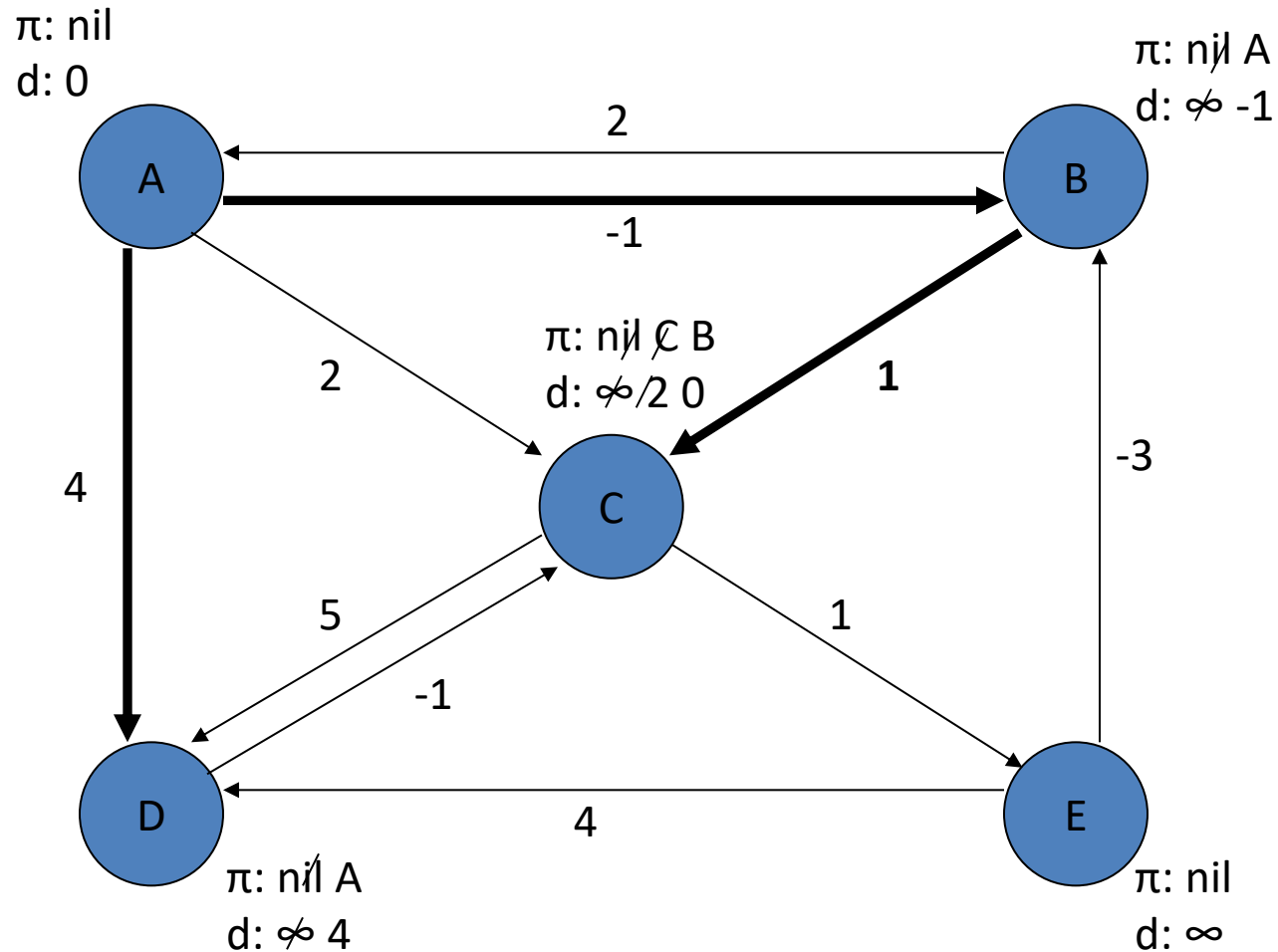
# Bellman Ford



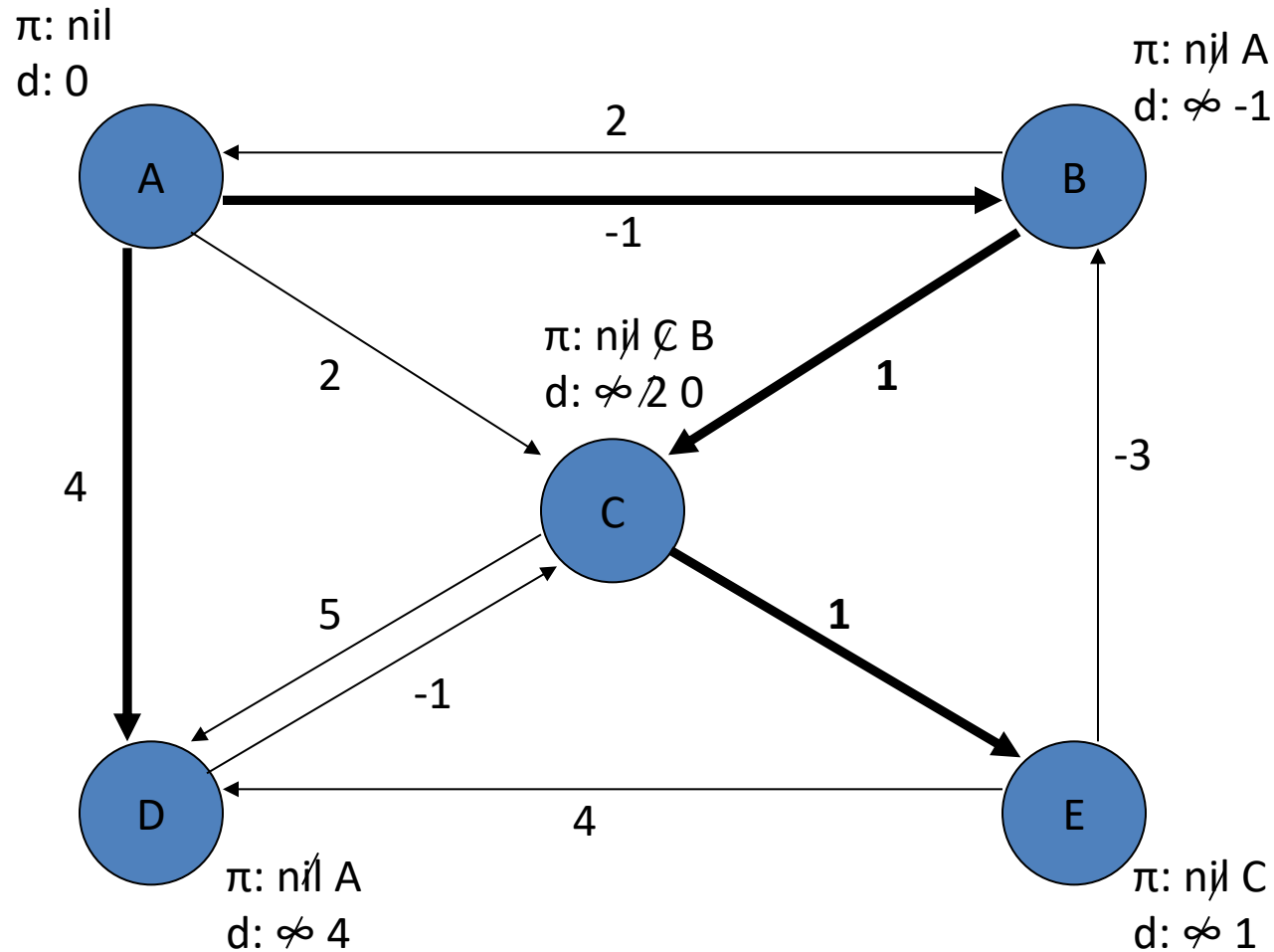
# Bellman Ford



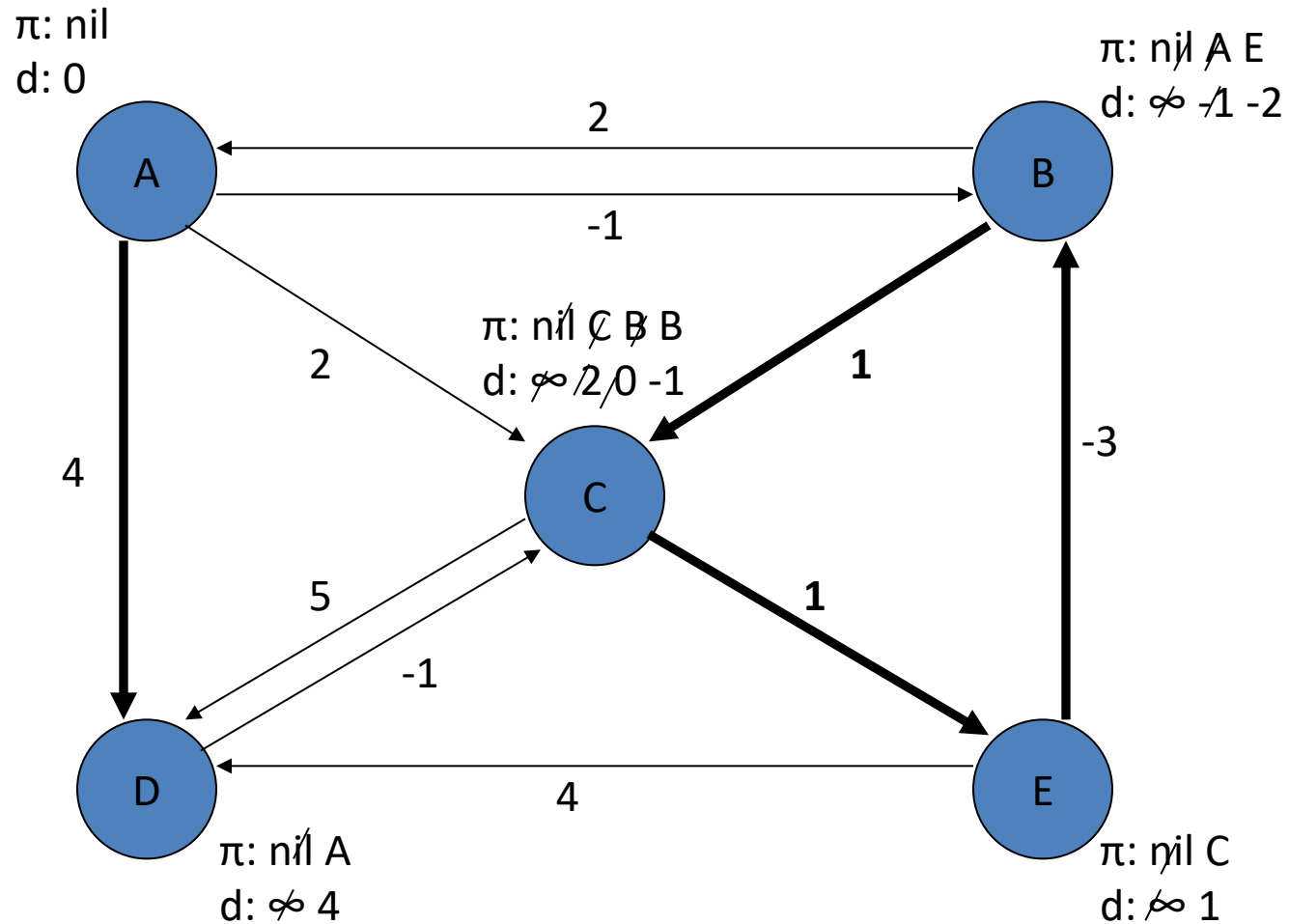
# Bellman Ford



# Bellman Ford



# Bellman Ford



# Bellman Ford

```
BellmanFord()  
  for each  $v \in V$   
     $d[v] = \infty$ ;  
   $d[s] = 0$ ;  
  for  $i=1$  to  $|V|-1$   
    for each edge  $(u,v) \in E$   
      Relax( $u,v, w(u,v)$ );  
  
  for each edge  $(u,v) \in E$   
    if ( $d[v] > d[u] + w(u,v)$ )  
      return "no solution";
```

Initialize  $d[]$ , which will converge to shortest-path value  $\delta$

Relaxation:  
Make  $|V|-1$  passes, relax each edge

Test for solution:  
have we converged yet?  
ie,  $\exists$  negative cycle?

Relax( $u,v,w$ ): if ( $d[v] > d[u]+w$ ) then  $d[v]=d[u]+w$



# Bellman-Ford Variant: SPFA

---

## ❑ Shortest path faster algorithm (SPFA)

### ❑ Modified Bellman-Ford using either BFS pattern

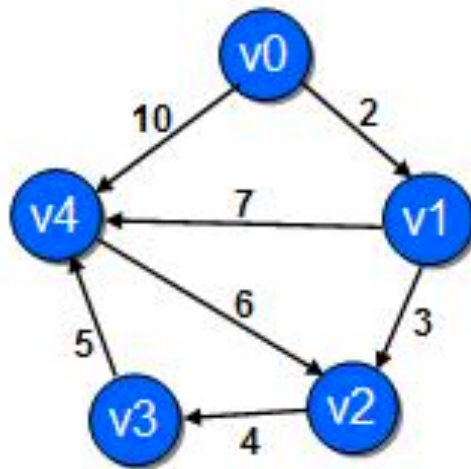
- Maintain the “frontier” vertices to relax

### ❑ Practical performance is excellent

```
input G,v
for each u ∈ V(G)
    let dist[u] = ∞
let dist[v] = 0
let Q be an initially empty queue
push(Q,v)
while not empty(Q)
    let u = pop(Q)
    for each (u,w) ∈ E(G)
        if dist[w] > dist[u]+wt(u,w)
            dist[w] = dist[u]+wt(u,w)
            if w is not in Q
                push(Q,w)
```

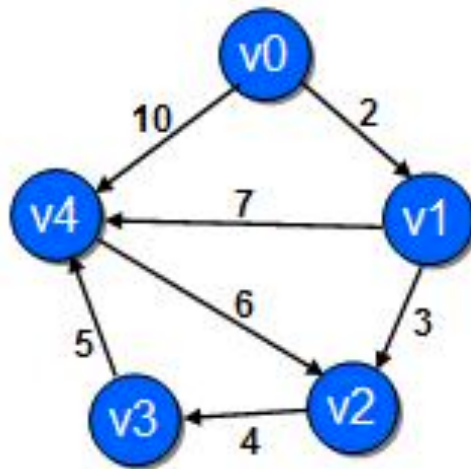
# Example

---



Let's start from v0

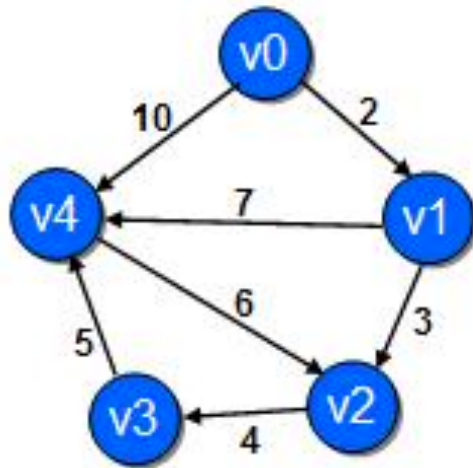
# Example



q	v0				
	v0	v1	v2	v3	v4
dis	0	$\infty$	$\infty$	$\infty$	$\infty$

Insert v0 into the queue

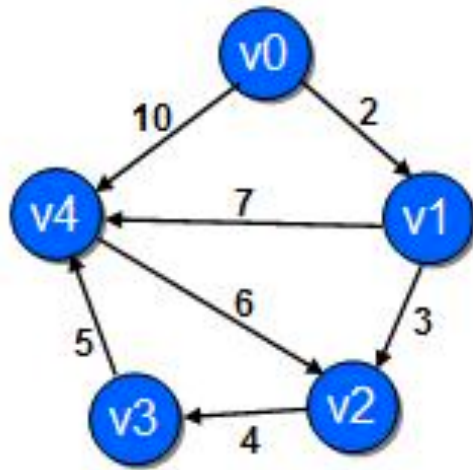
# Example



q	v1	v4			
	v0	v1	v2	v3	v4
dis	0	2	$\infty$	$\infty$	10

Relax v1 and v4 and insert them to the queue

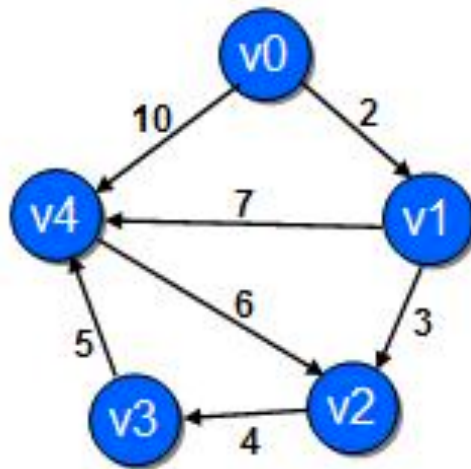
# Example



q	v4	v2			
	v0	v1	v2	v3	v4
dis	0	2	5	$\infty$	9

Relax v2 and v4 and insert them to the queue

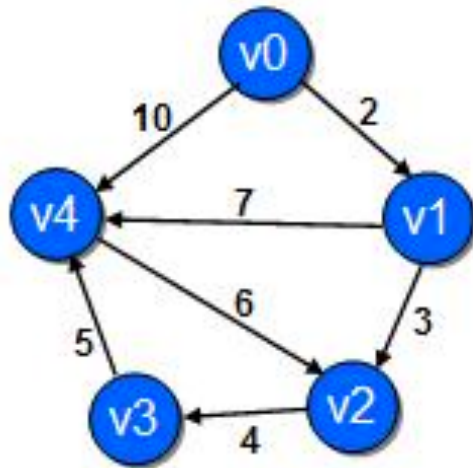
# Example



q	v2				
	v0	v1	v2	v3	v4
dis	0	2	5	$\infty$	9

Nothing to relax from v4

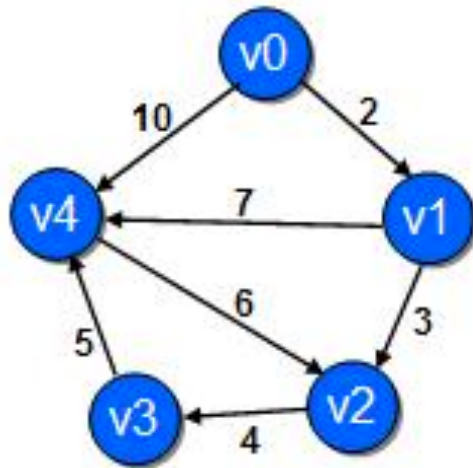
# Example



q	v3				
	v0	v1	v2	v3	v4
dis	0	2	5	9	9

Relax v3 and insert it to the queue

# Example



q					
	v0	v1	v2	v3	v4
dis	0	2	5	9	9

Nothing to relax from v3, done



# What about Negative Cycle in SPFA?

---

- ☐ If a vertex is inserted into the queue N times
  - ☐ Negative cycle found!
- ☐ What is the time complexity of SPFA?