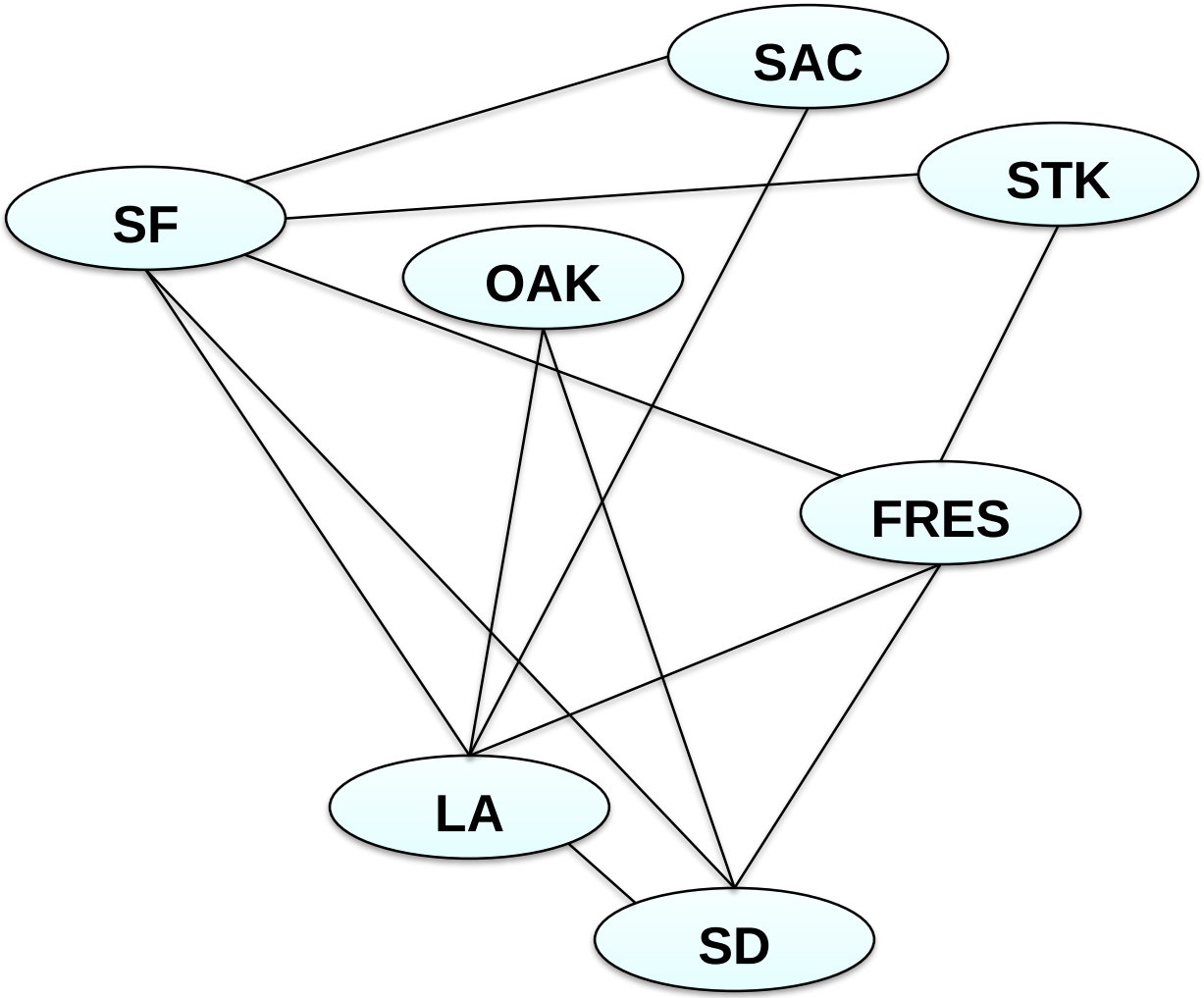
HANDONG GLOBAL UNIVERSITY

Chapter 22
Elementary Graph Algorithms

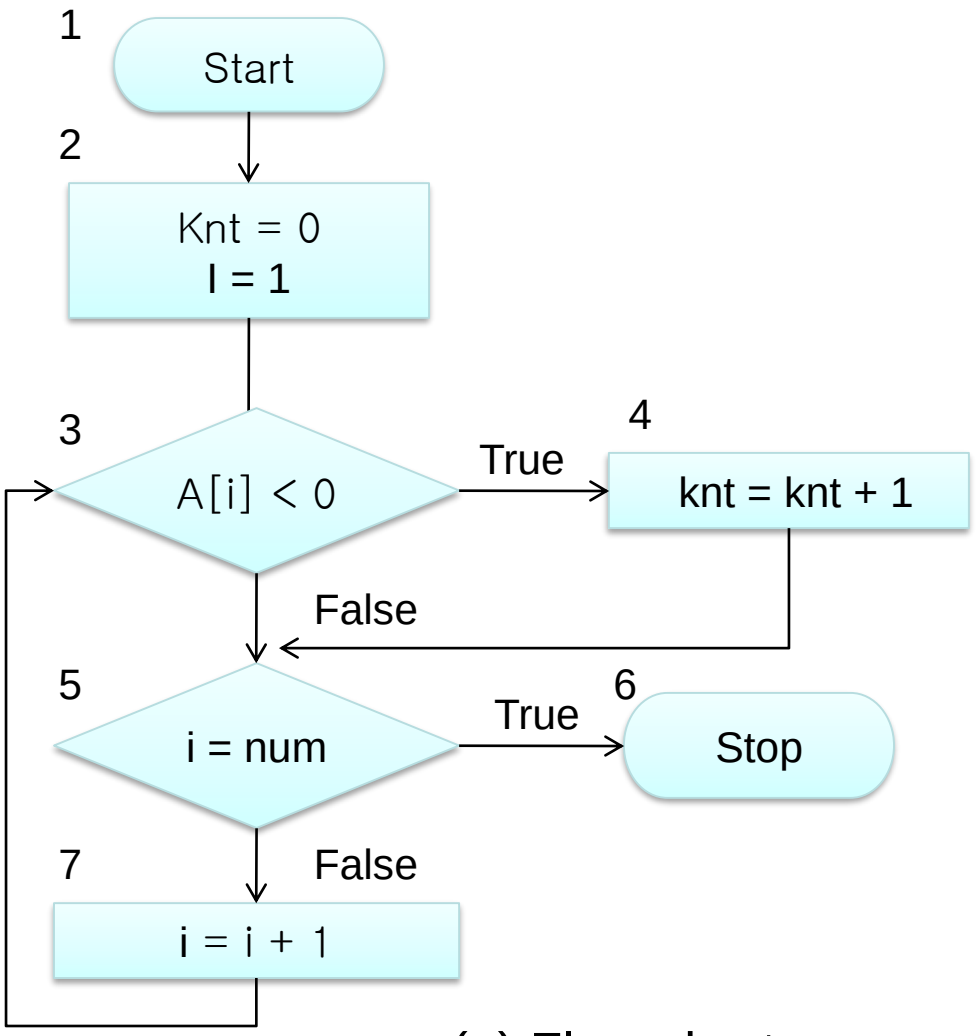
Algorithm Analysis

School of CSEE

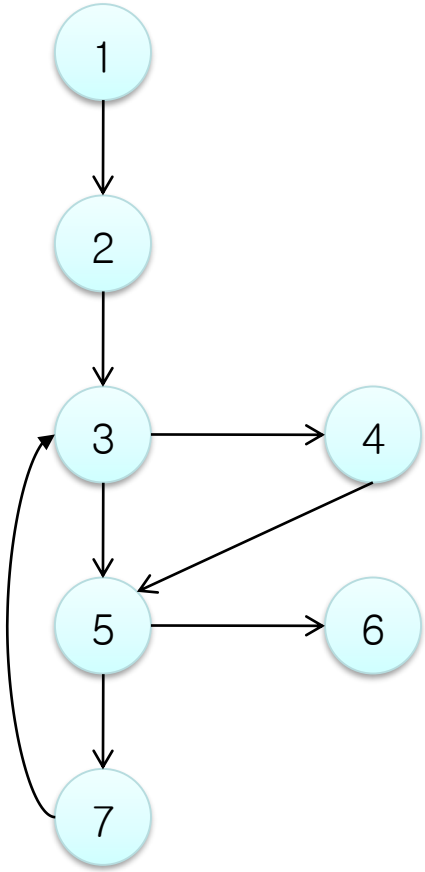
Problems: Airline Routes



Problems: Flowcharts



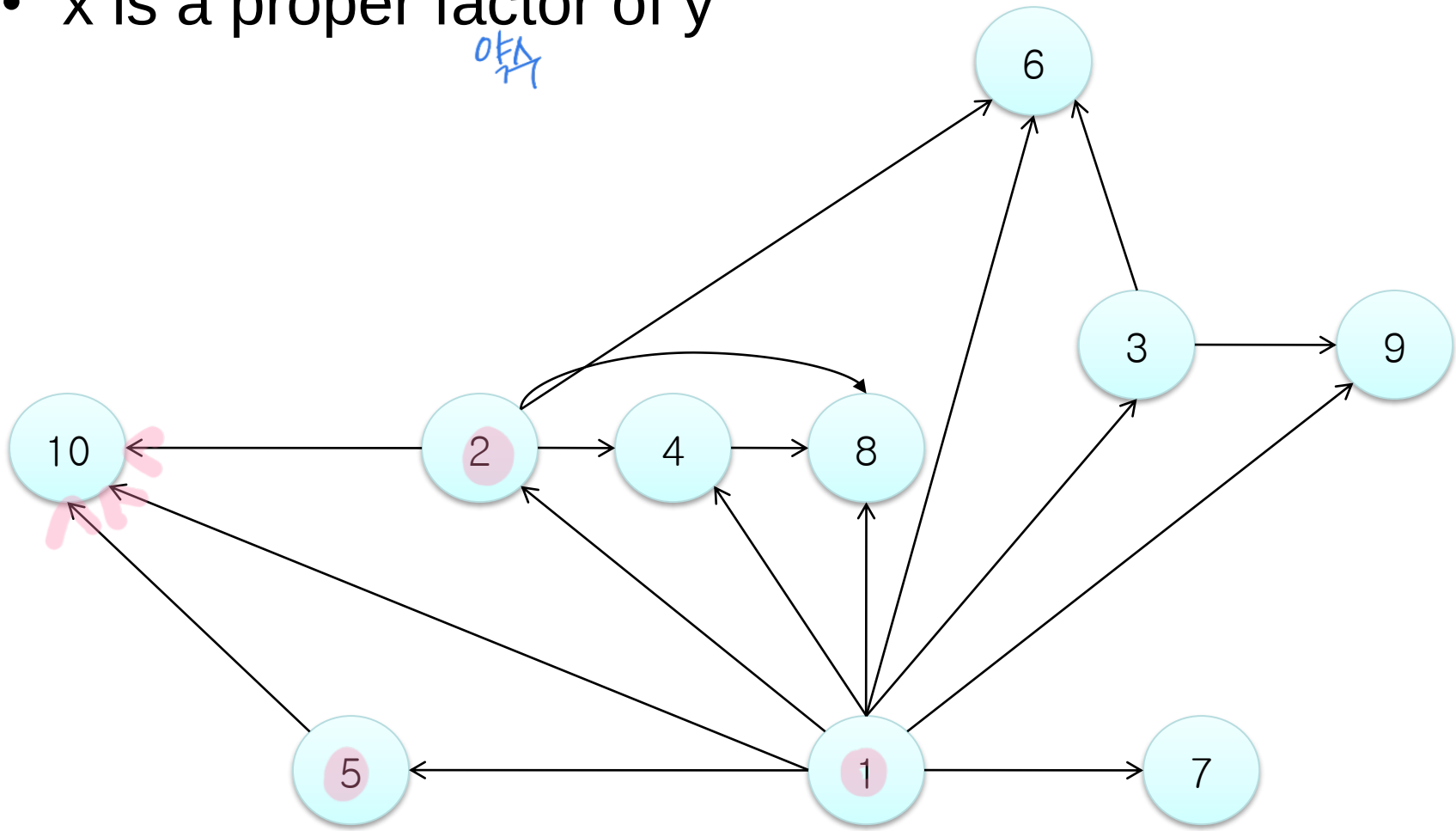
(a) Flow chart



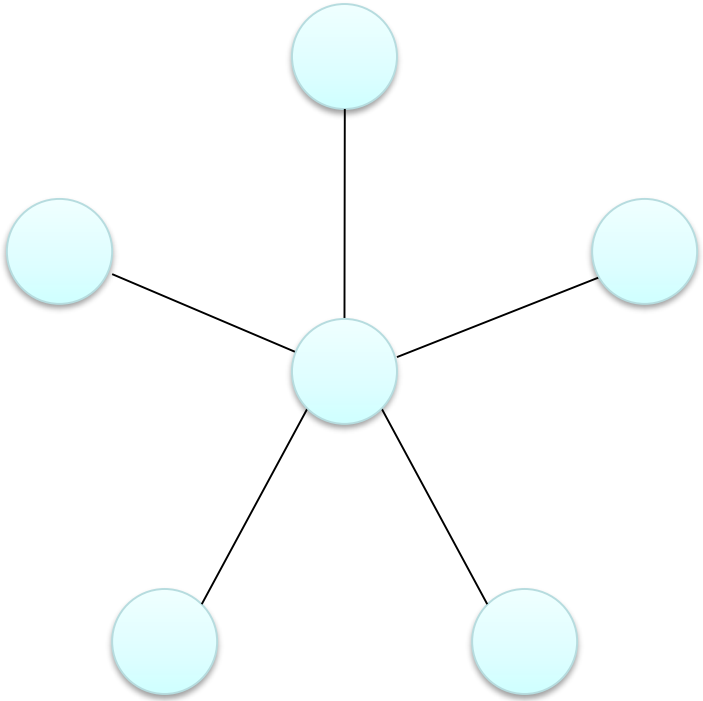
(b) Directed graph

Problems: Binary relation

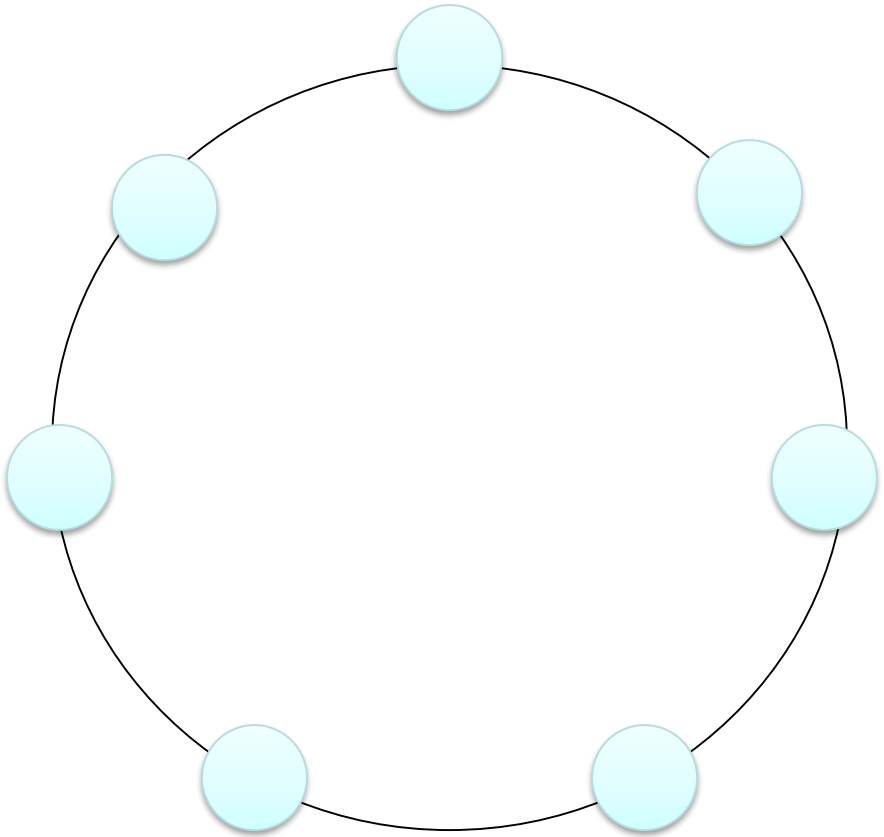
- x is a proper factor of y



Problems: Computer Networks



(a) A star network



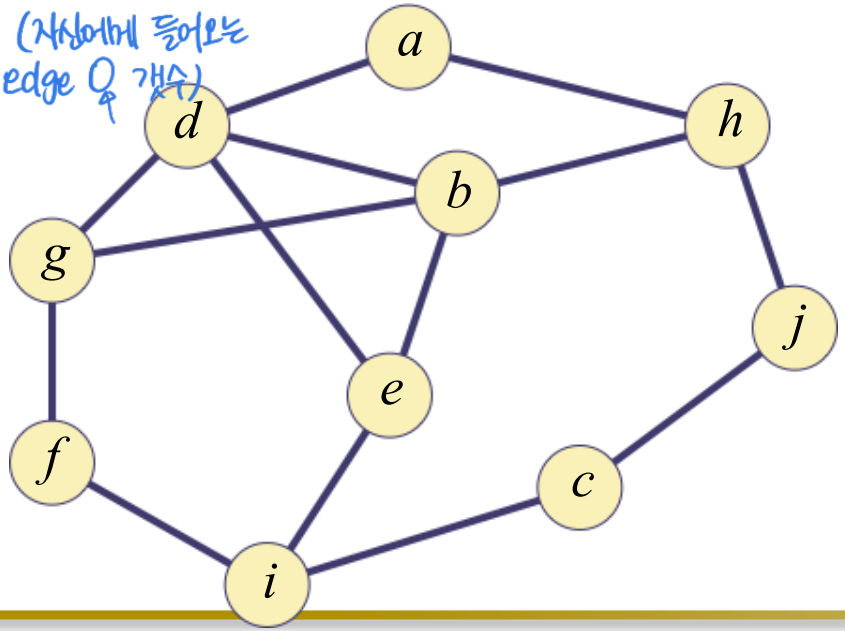
(b) A ring network

Questions…

1. What is the cheapest way to fly from Pohang to New York?
2. Which route involves the least flying time?
3. If one City's airport is closed by bad weather, can you still fly between every other pair of cities?
4. If one computer in a network goes down, can a messages be sent between every other pair of computers in the network?
5. Does a flow chart have any loop?
6. How should wires be attached to various electrical outlets so that all are connected together using the least amount of wire?

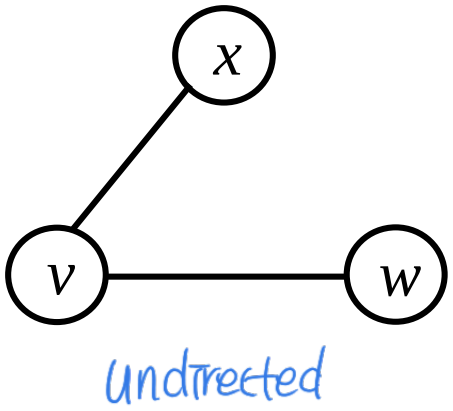
Terms You Should Know or Learn Now

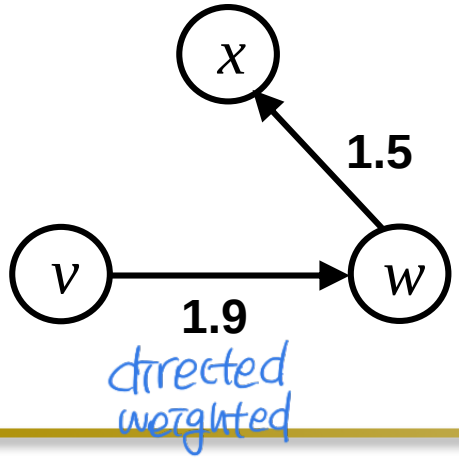
- Vertex (plural *vertices*) or Node
- Edge (sometimes referred to as an *arc*)
  - Note the meaning of *incident*
- Degree of a vertex: how many adjacent vertices
  - Digraph: in-degree (num. of incoming edges) vs. out-degree

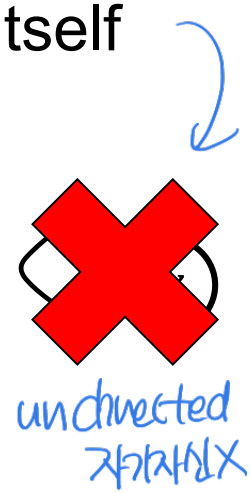


Terms You Should Know or Learn Now

- Graphs can be:
  - Directed or undirected
  - Weighted or not weighted
    - weights can be reals, integers, etc.
    - weight also known as: cost, length, distance, capacity,…
- Undirected graphs:
  - Normally an edge can't connect a vertex to itself

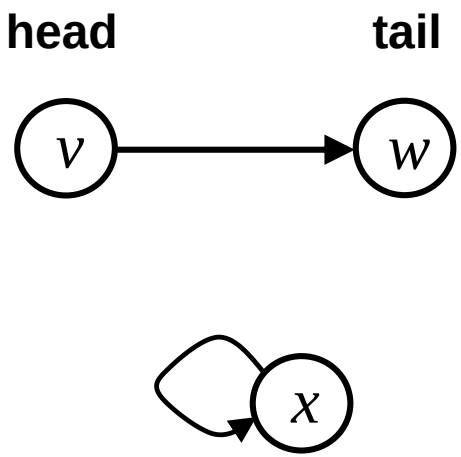


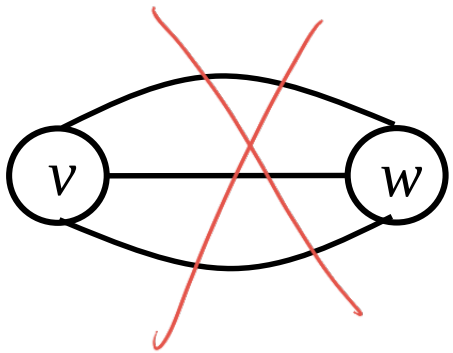




Terms You Should Know or Learn Now

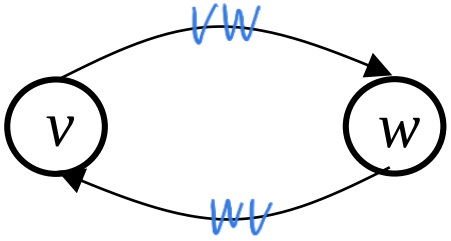
- A directed graph (also known as a *digraph*)
  - "Originating" node is the *head*, the target the *tail*
  - An edge may (or may not) connect a vertex to itself
- A graph may not have multiple occurrences of the same edge.





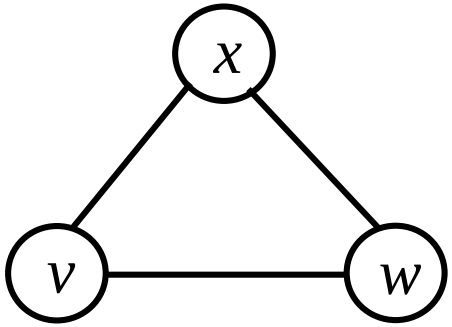
Terms You Should Know or Learn Now

- Symmetric digraph
  - For every edge *vw*, there is also the reverse edge *wv*.



- Complete graphs
  - There is an edge with each pair of vertices.

모든 vertex (node) 간 edge 有



Terms You Should Know or Learn Now

- Size of graph? Two measures:
  - Number of nodes. Usually n , v
  - Number of edges: Usually m , e
- Dense graph: many edges
  - Undirected:

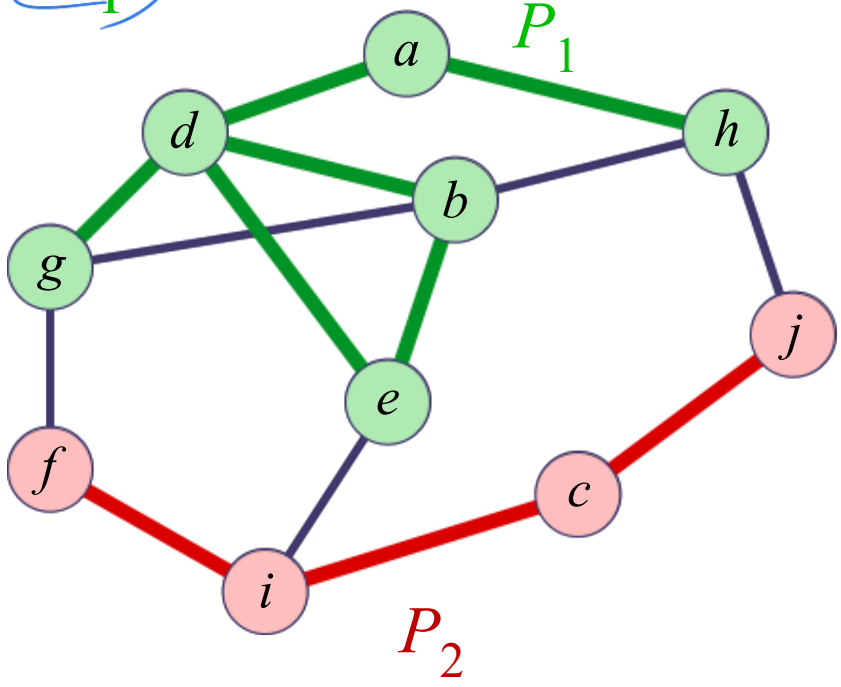
    Each node connects to all others, so the graph with $m = n(n-1)/2$ is called a *complete graph.*
  - Directed: $m = n(n-1)$
- Sparse graph: fewer edges
  - Could be zero edges…

Paths and Cycles

- A *path* is a sequence of vertices $P = (v_0, v_1, \ldots, v_k)$ such that, for $1 \le i \le k$, edge $(v_{i-1}, v_i) \in E$.

- Path P is *simple* if no vertex appears more than once in P.

$P_1 = (g, d, e, b, d, a, h)$ is not simple.

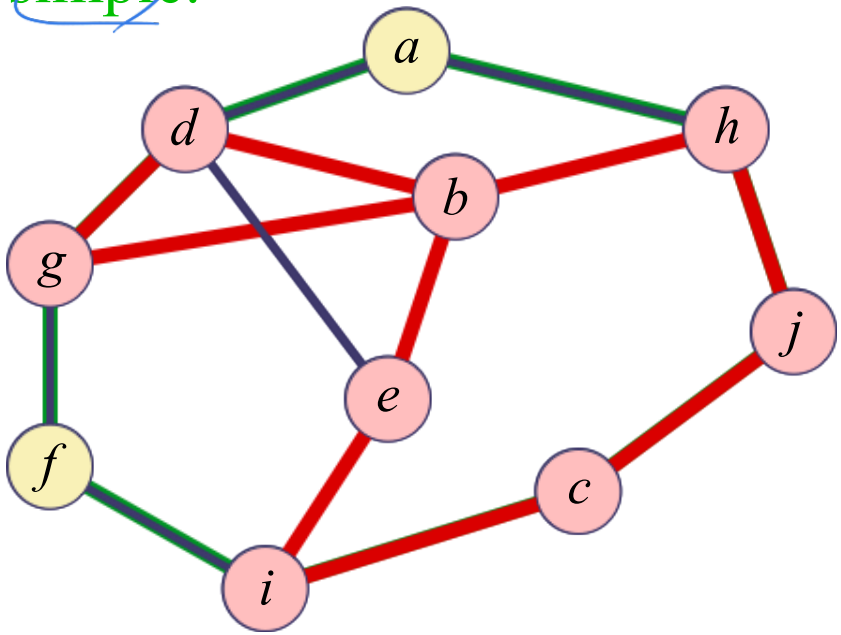


$P_2 = (f, i, c, j)$ is simple.

Paths and Cycles

- A *cycle* is a sequence of vertices $C = (v_0, v_1, \ldots, v_{k-1})$ such that, for $0 \le i < k$, edge $(v_i, v_{(i+1) \bmod k}) \in E$.
- Cycle C is *simple* if the path (v_0, v_1, v_{k-1}) is simple.

$C_1 = (a, h, j, c, i, f, g, d)$ is simple.



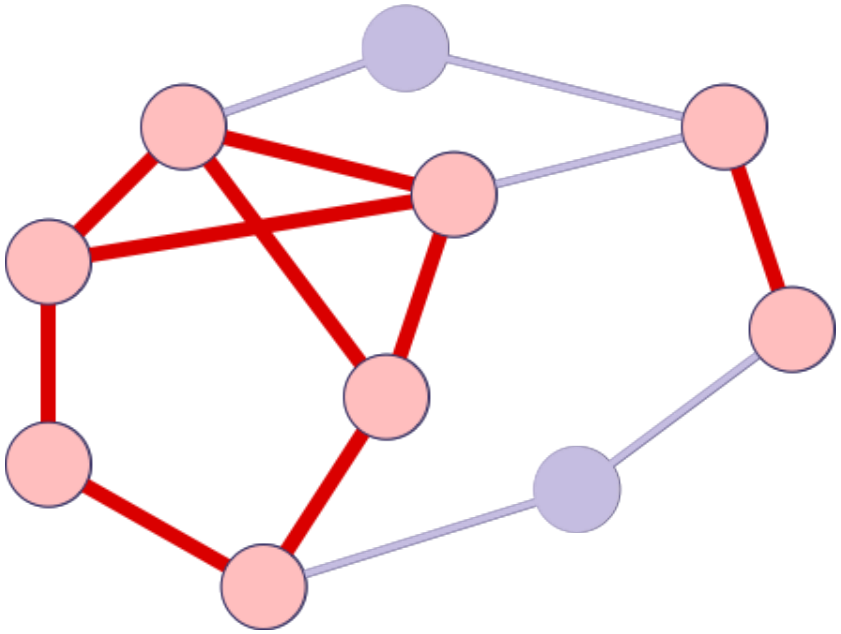
$C_2 = (g, d, b, h, j, c, i, e, b)$ is not simple.

Subgraphs

- A graph $H = (W, F)$ is a *subgraph* of a graph $G = (V, E)$
- if $W \subseteq V$ and $F \subseteq E$.

(subset)

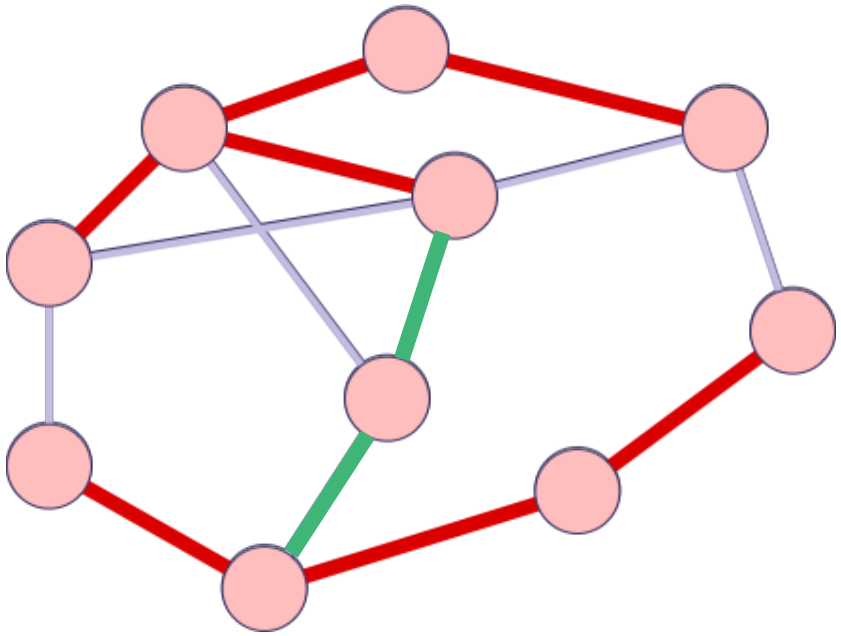
둘 다 subset 이어야 함



Spanning Graphs

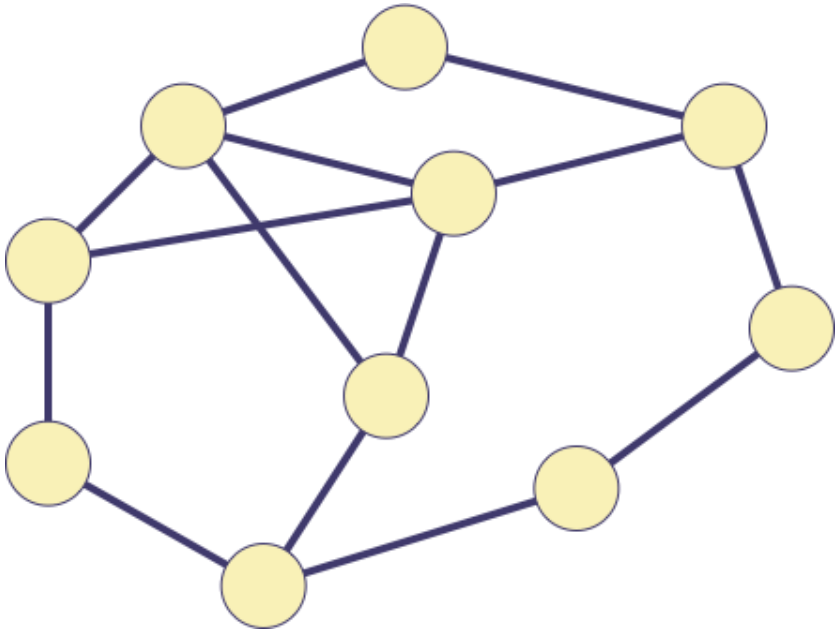
(줄기 + 뼈강)

- A *spanning graph* of *G* is a subgraph of *G* that contains all vertices of *G*.

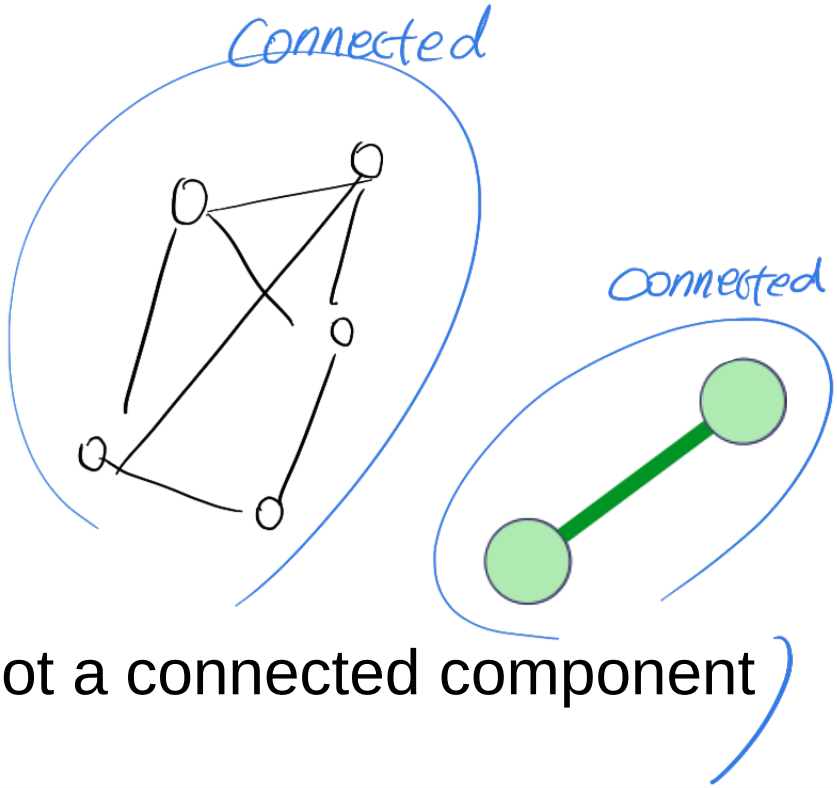


Connectivity : Undirected graph

A graph *G* is ***connected*** if there is a path between any two vertices in *G*.



The ***connected components*** of a graph are its maximal connected subgraphs.

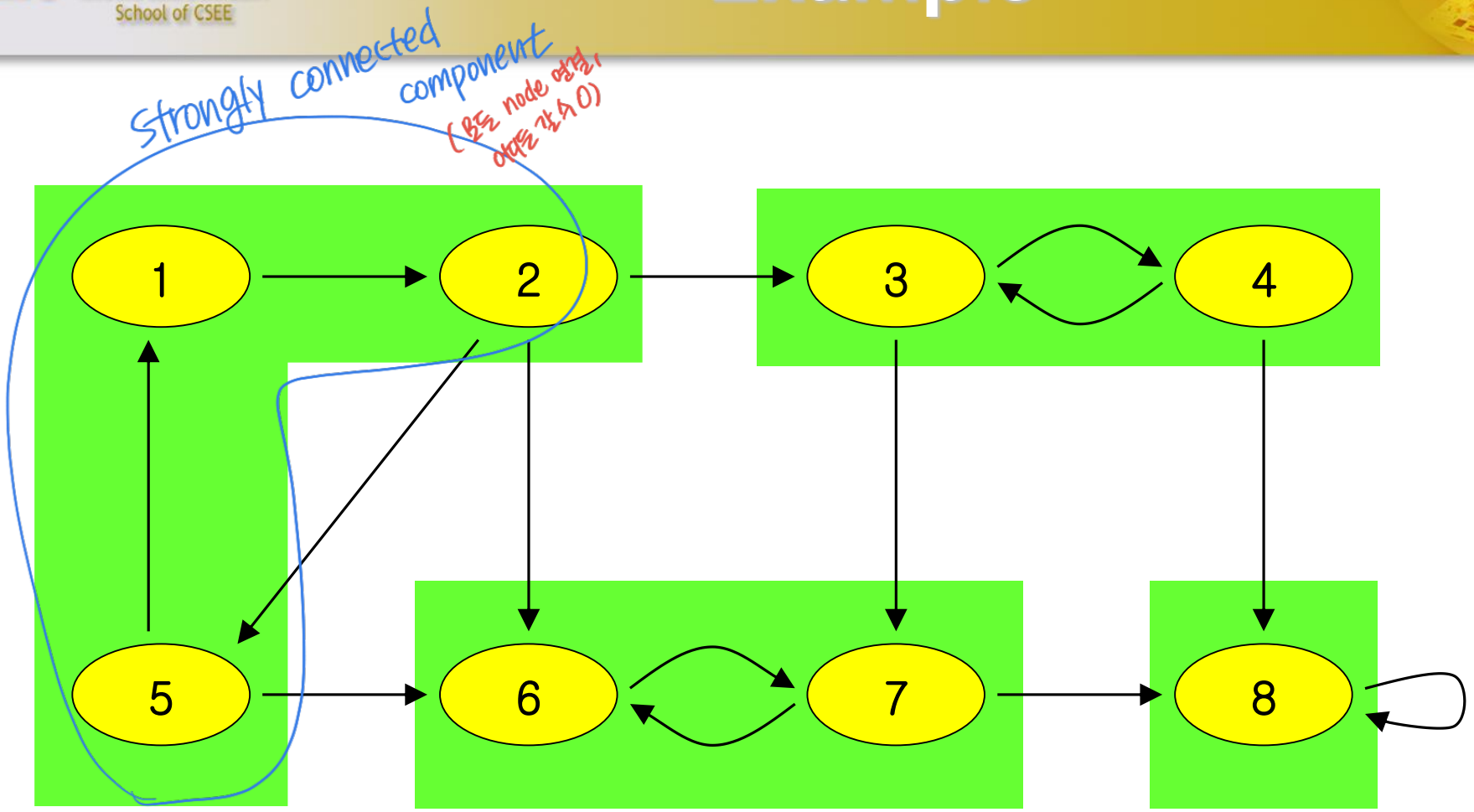


And more terminology

- A *strongly connected digraph:*
  - direction affects this!
  - node *u* may be reachable from *v*, but not *v* from *u*

    ➔ not strongly connected
  - Strongly connected means both directions
- Acyclic: no-cycles (방향과 무관)
- Directed acyclic graph: a DAG

Example

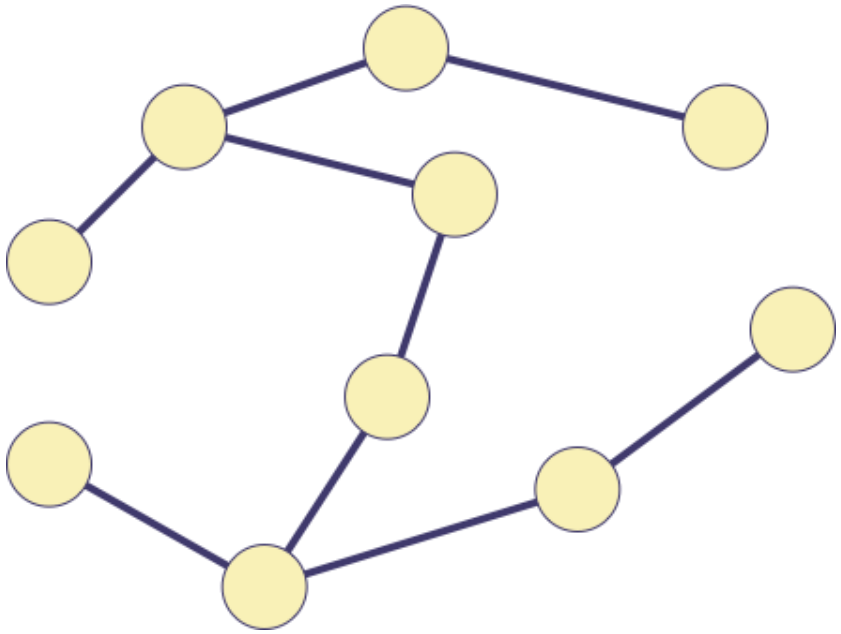


Green areas are the Strongly Connected Components.

= 4개!

Trees

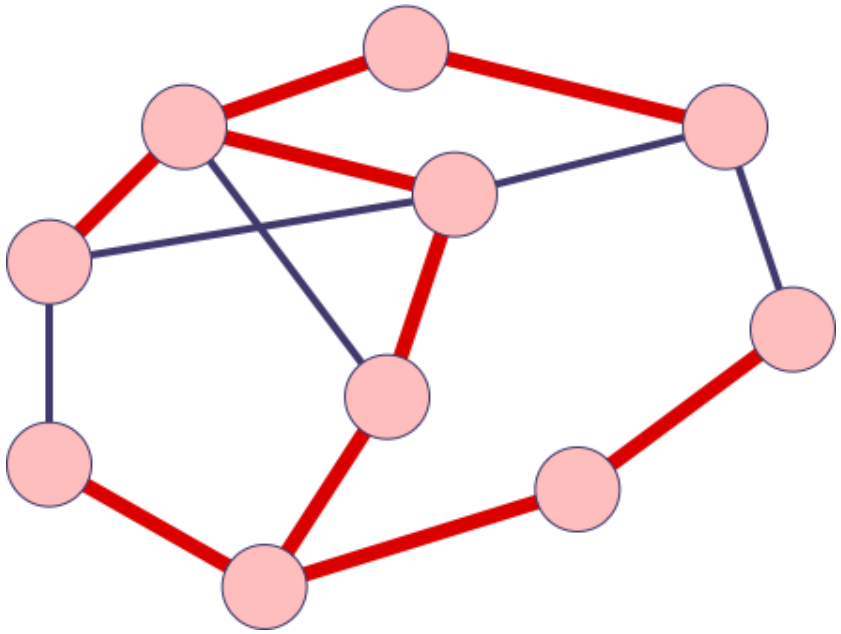
A ***tree*** is a graph that is connected and contains no cycles.



Spanning Trees

A ***spanning tree*** of a graph is a spanning graph that is a tree.

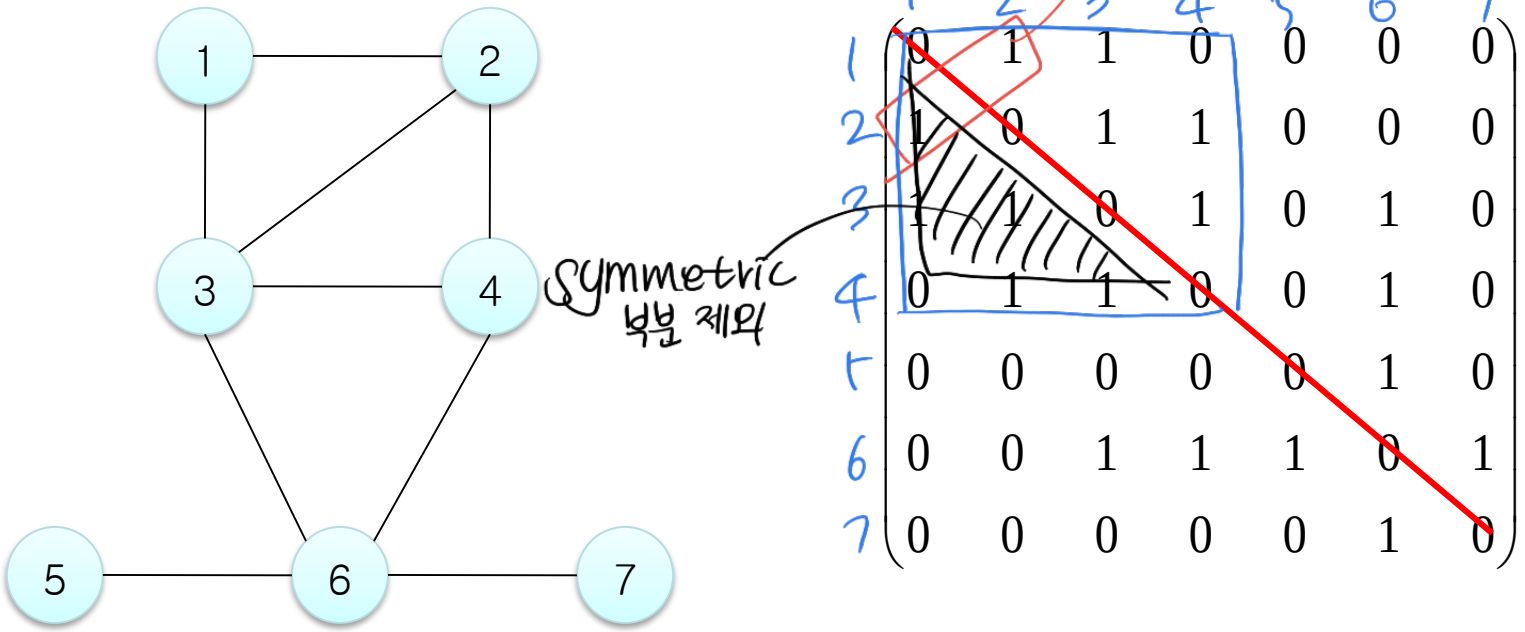
cycle X



Graph Representations

Adjacency Matrix Representation

- Let G = (V,E), n = |V|, m = |E|, V = {v1, v2, …, vn)
- G can be represented by an n × n matrix



(a) An undirected graph (b) Its adjacency matrix

Graphs: Adjacency Matrix

- Q: *How much storage does the adjacency matrix require?*

  A: $O(V^2)$

- Q: *What is the minimum amount of storage needed by an adjacency matrix representation of an undirected graph with 4 vertices?* 3비트 storage? 4X4=16? X

  A: 6 bits

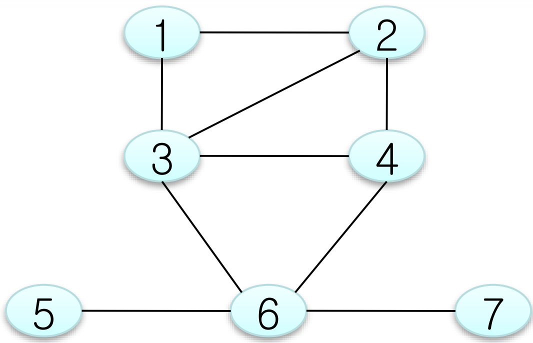
  – Undirected graph → matrix is symmetric

  – No self-loops → don't need diagonal

Graphs: Adjacency Matrix

- Preferred when the graph is dense.
  - Usually too much storage for large graphs
  - But can be very efficient for small graphs
- Most large interesting graphs are sparse
  - For this reason the *adjacency list* is often a more appropriate representation

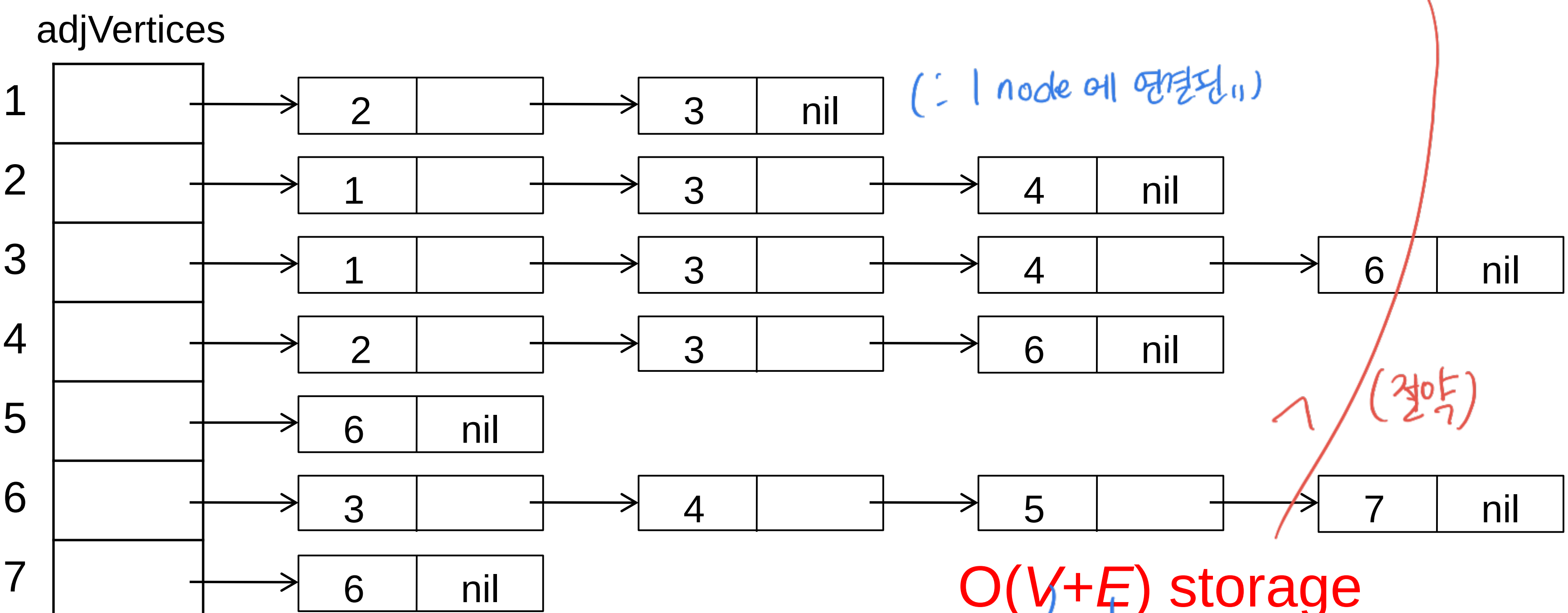
Array of Adjacency Lists Representation



(a) An undirected graph

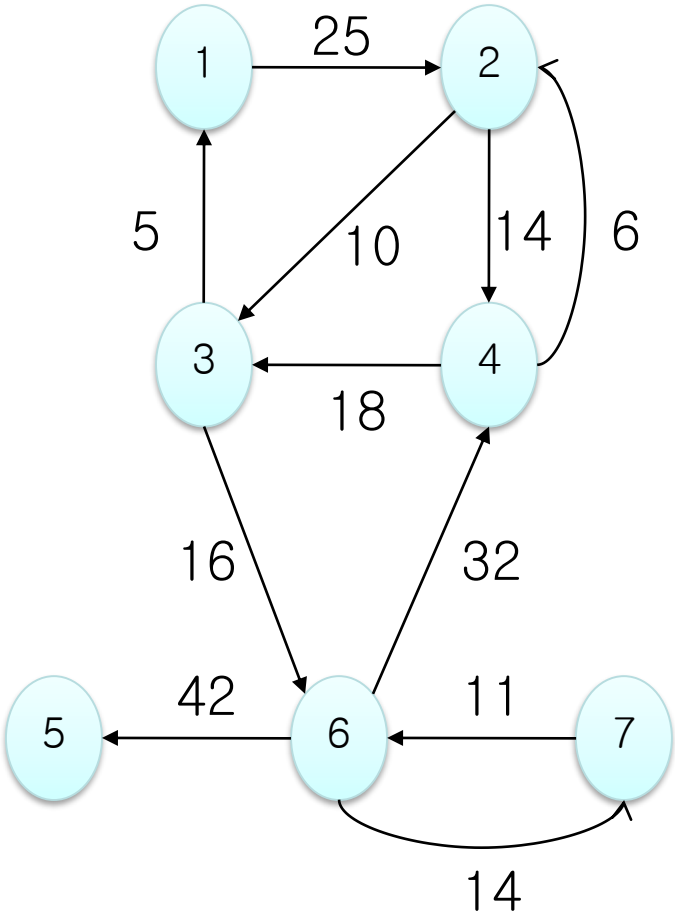
$$\begin{pmatrix} 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix}$$

(b) Its adjacency matrix



O(*V*+*E*) storage

Adjacency Matrix for weight digraph



(a) A weighted digraph

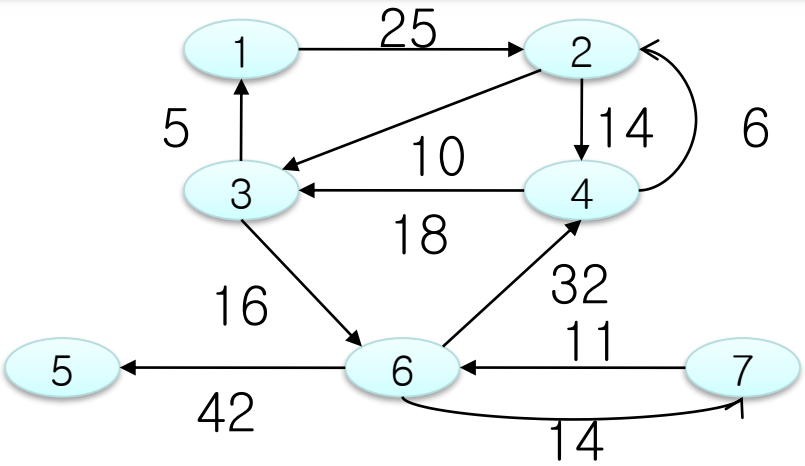
Symmetric X (diagraph)

(2→1, X)

$$\begin{pmatrix} 0 & 25.0 & \infty & \infty & \infty & \infty & \infty \\ \infty & 0 & 10.0 & 14.0 & \infty & \infty & \infty \\ 1 & \infty & 0 & \infty & \infty & 16.0 & \infty \\ \infty & 6.0 & 18.0 & 0 & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & 0 & \infty & \infty \\ \infty & \infty & \infty & 32.0 & 42.0 & 0 & 14.0 \\ \infty & \infty & \infty & \infty & \infty & 11.0 & 0 \end{pmatrix}$$

(b) Its adjacency matrix

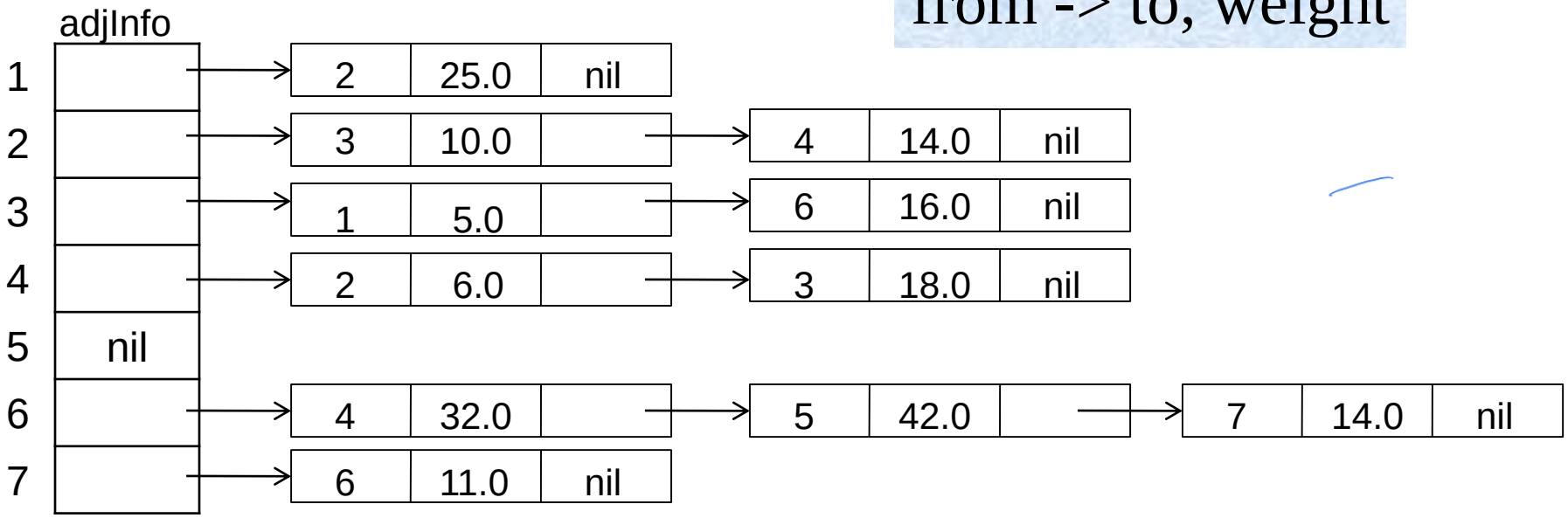
Array of Adjacency Lists Representation



(a) A weighted digraph

$$\begin{pmatrix} 0 & 25.0 & \infty & \infty & \infty & \infty & \infty \\ \infty & 0 & 10.0 & 14.0 & \infty & \infty & \infty \\ 1 & \infty & 0 & \infty & \infty & 16.0 & \infty \\ \infty & 6.0 & 18.0 & 0 & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & 0 & \infty & \infty \\ \infty & \infty & \infty & 32.0 & 42.0 & 0 & 14.0 \\ \infty & \infty & \infty & \infty & \infty & 11.0 & 0 \end{pmatrix}$$

from -> to, weight



Graph Exploration

- Goal:
  - Visit all the vertices in the graph of *G*
    - Count them
    - Number them
    - …
  - Identify the connected components of *G*
  - Compute a spanning tree of *G*

BFS

Breadth-First Search

- Scan a graph, and turn it into a 'Breadth-first (spanning) tree' traversal, level order
  - Tree with discovered vertices and explored edges. (포함된) (모든 vertex 포함) 무조건 다 포함X
  - One vertex at a time
  - Pick a *source vertex*, *s* to be the root.
  - Find ("discover") its children, then their children, etc. (level by level)
  - For any vertex reachable from *s*, the path in the tree from *s* to *v* corresponds to a "shortest path' from *s* to *v* in *G*.

* 'Discover' the vertex : The vertex is first encountered. 처음 만났을 때
* 'Explore' the edge : The edge is first examined. (explore 되지 않은 edge도 有)

Breadth-First Search

- Again we will associate vertex "colors" to guide the algorithm.
  - White vertices have not been discovered.
    - All vertices start out white.
  - Gray vertices are discovered but not finished. (방문중)
    - They may be adjacent to white vertices.
    - The vertex is added to breadth-first tree.
  - Black vertices are discovered and finished.
    - They are adjacent only to black and gray vertices.

Breadth-First Search

- As soon as the white vertex is discovered
  - Turn its color to 'gray'.
  - Discover all of its white neighbor.
  - After all of its neighbors are discovered, turn its color to black (means it is finished).

- Discover vertices by scanning adjacency list of gray vertices.

Breadth-First Search

```
BFS(G, s)                                   // s: source vretex
  for each vertex u ∈ V – {s}
      do d[u] = ∞;   π[u] = NIL;            // π : predecessor (parent in the tree)
  color[s] = GRAY;  d[s] = 0; π[s] = NIL;   // source vertex 처리
  Q = ∅
  ENQUEUE(Q, s)
  while Q ≠ ∅
     do u = DEQUEUE(Q)
        for each v ∈ Adj[u]
            do if color[v] = WHITE
                  then color[v] = GRAY
                       d[v] = d[u] + 1
                       π[v] = u
                       ENQUEUE(Q, v)
     color[u] = BLACK
```

Analysis

Running time of Breadth-First search:

A. Each node is enqueued once. (white → gray) : $\Theta(V)$

한 번 queue에 들어 감

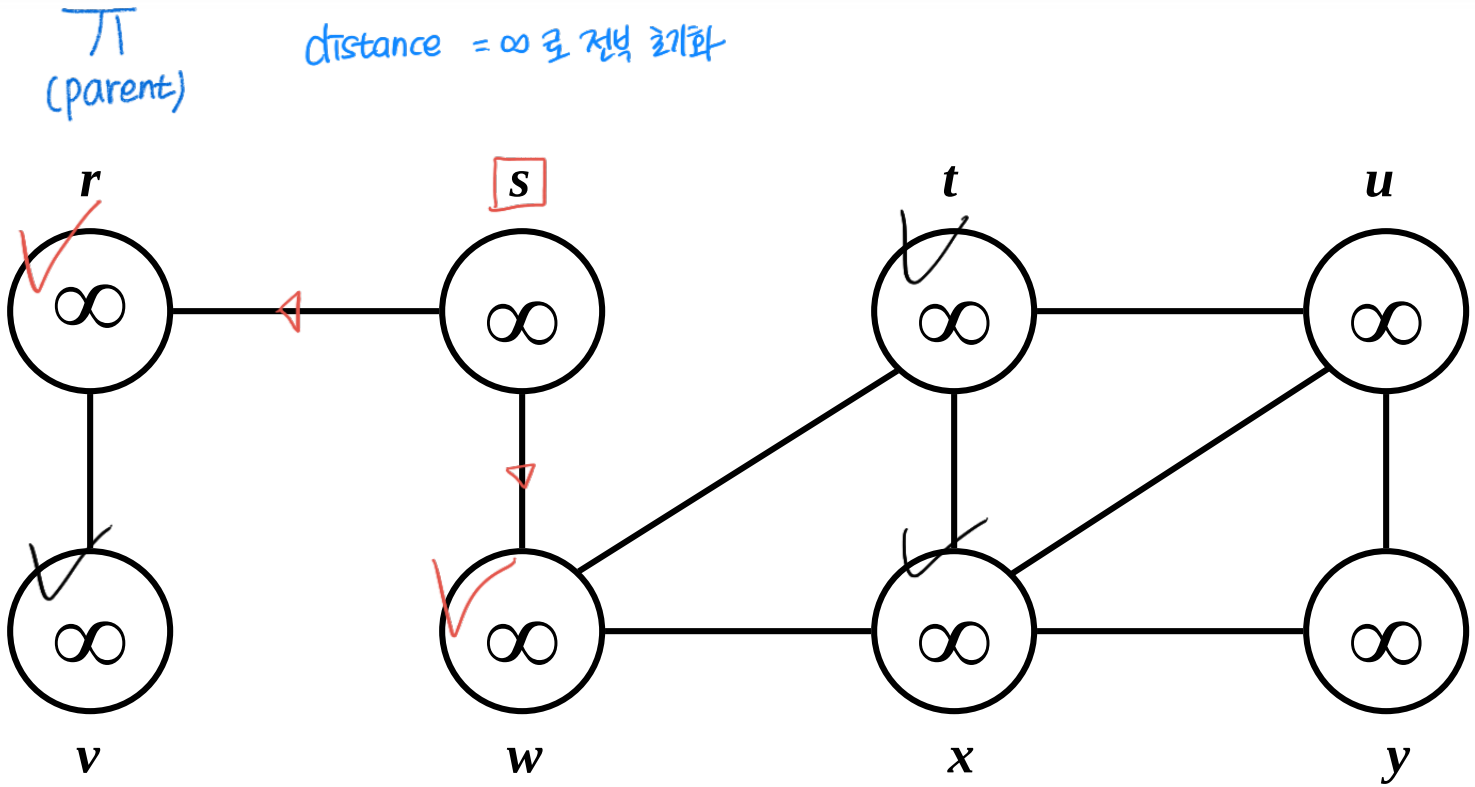
B. Each node is dequeued once. (gray → black) : $\Theta(V)$

C. Each adjacency list is scanned only once. : $\Theta(E)$

Overall running time is $\Theta(V+E)$.

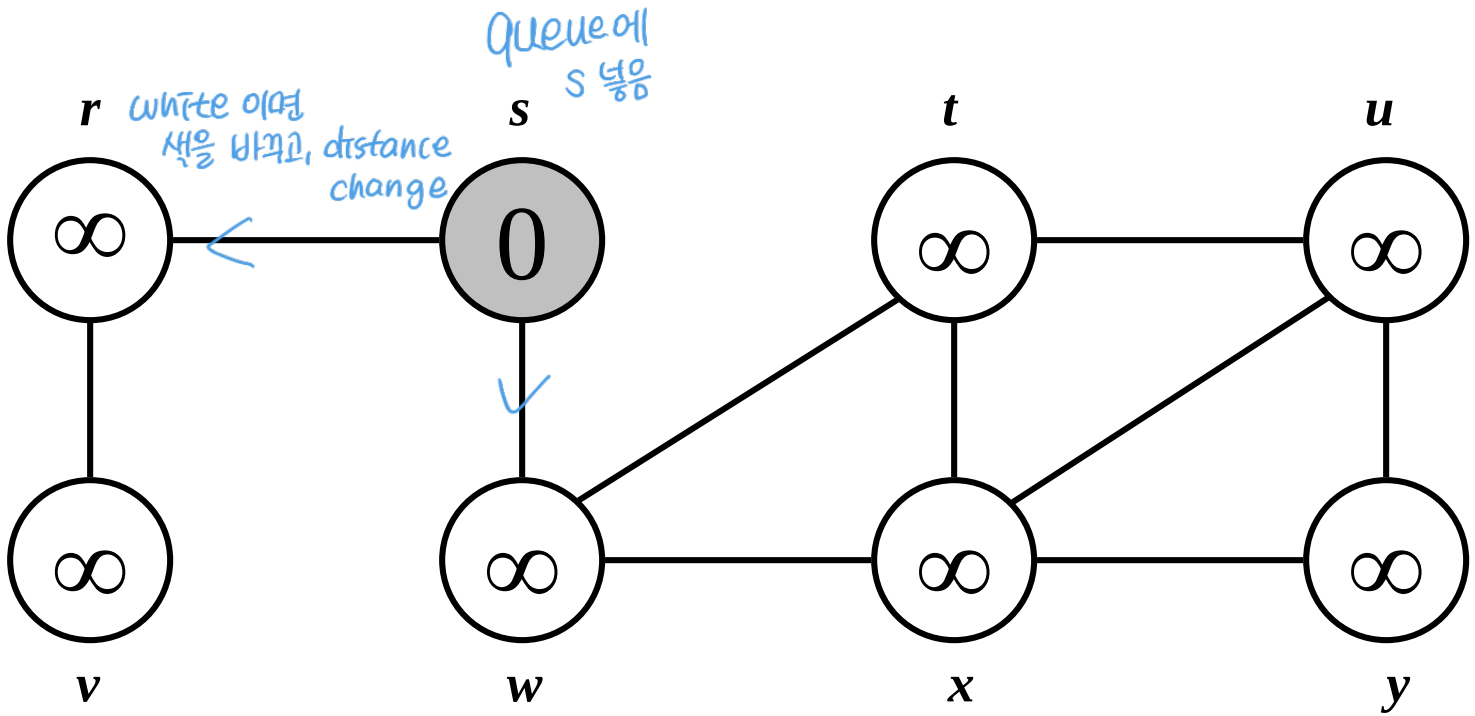
최소 시간

Breadth-First Search: Example



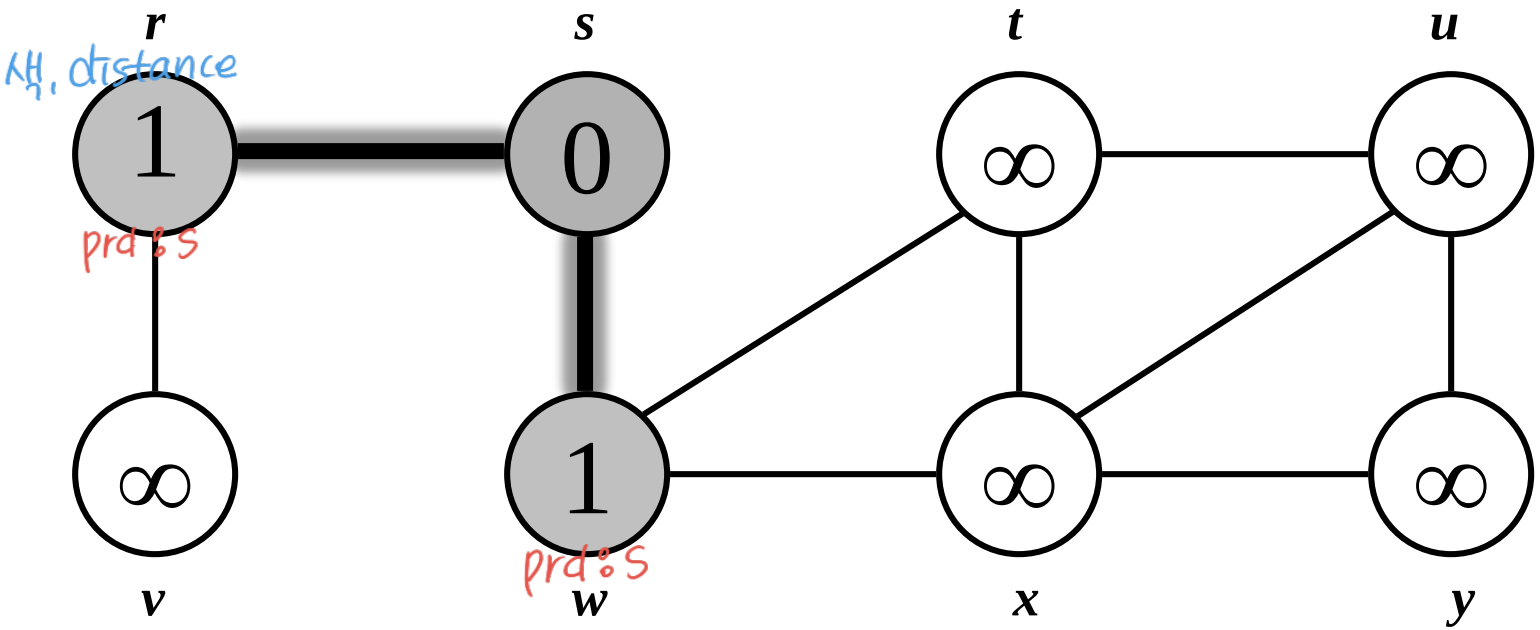
Vertex is an island and edge is an bridge. BFS is visualized as many simultaneous (or nearly simultaneous) explorations from the starting point and spreading out independently.

Breadth-First Search: Example



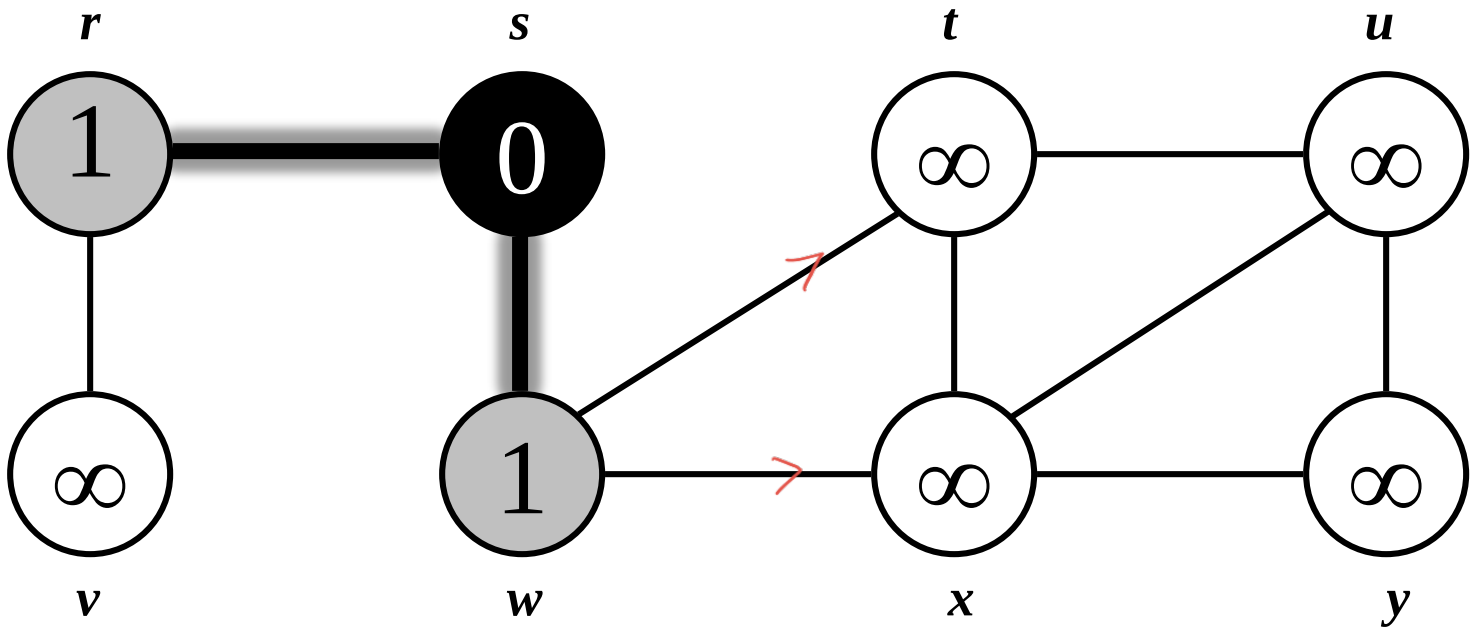
Q: | *s* |

Breadth-First Search: Example



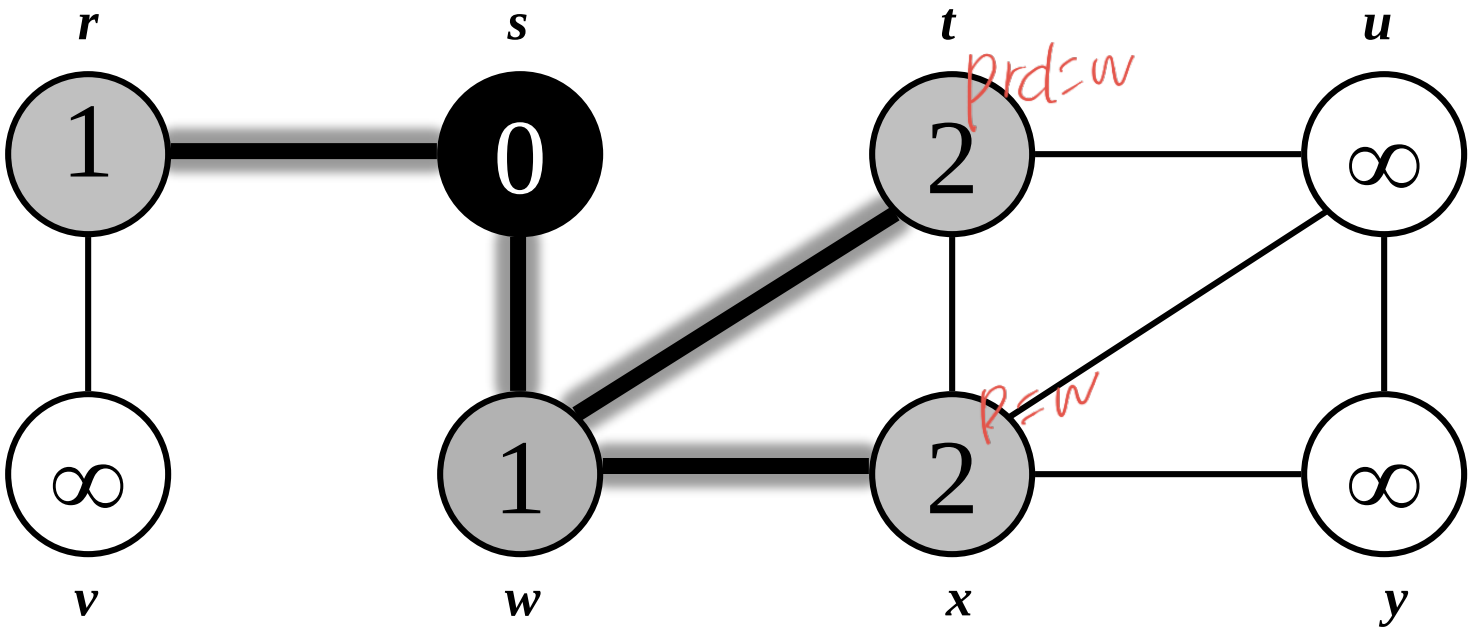
Q: | *w* | *r* |

Breadth-First Search: Example



Q: | *w* | *r* |

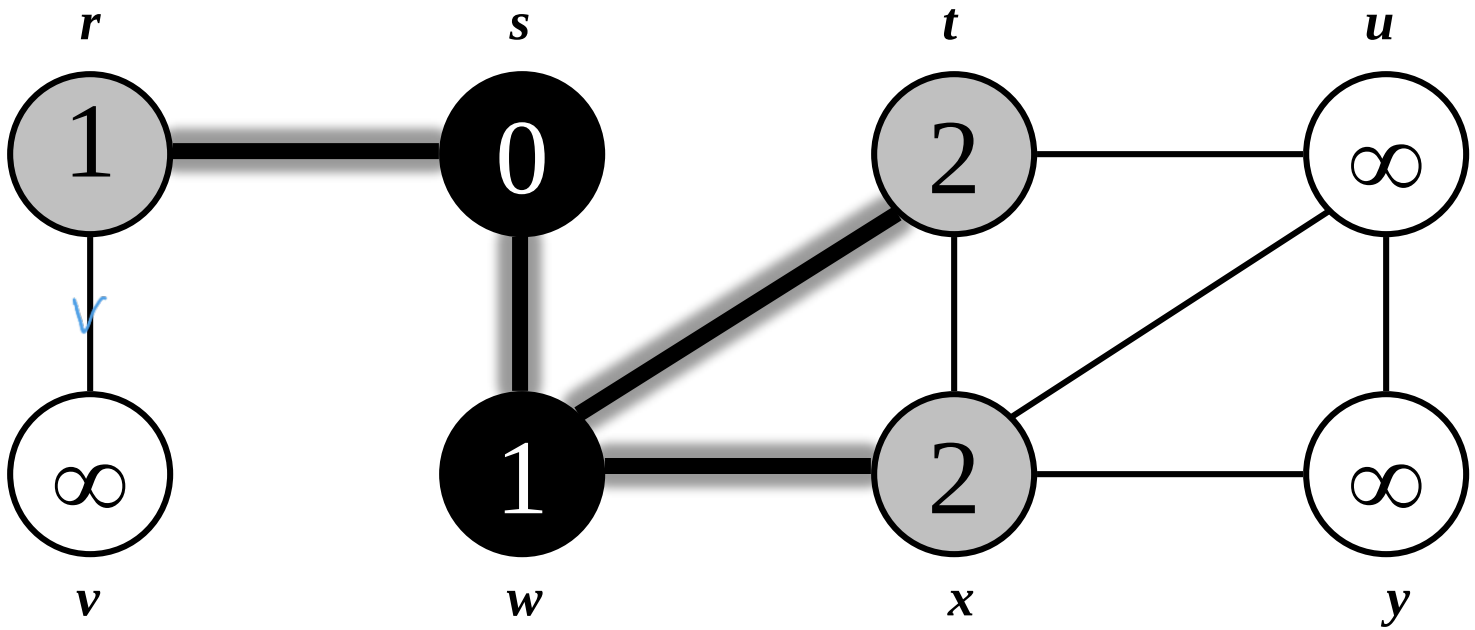
Breadth-First Search: Example



w (Queue에서 w 빼어 냄)

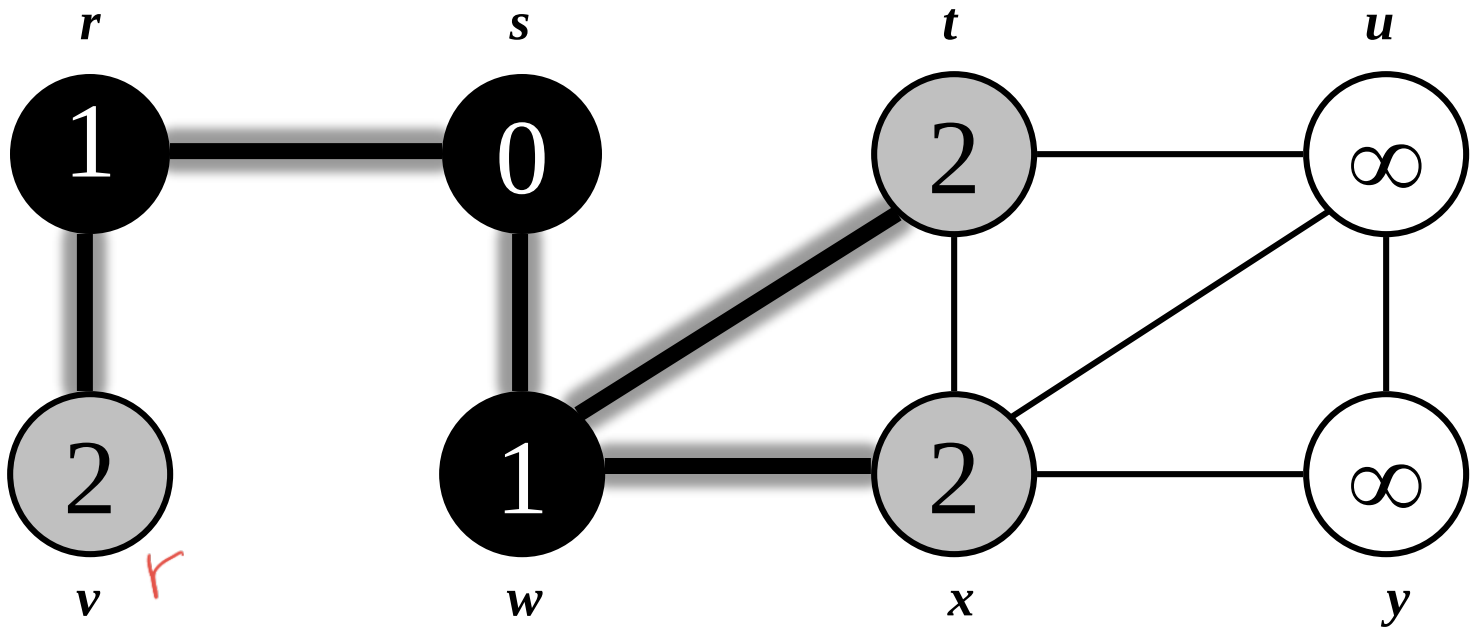
Q: | r | t | x |

Breadth-First Search: Example



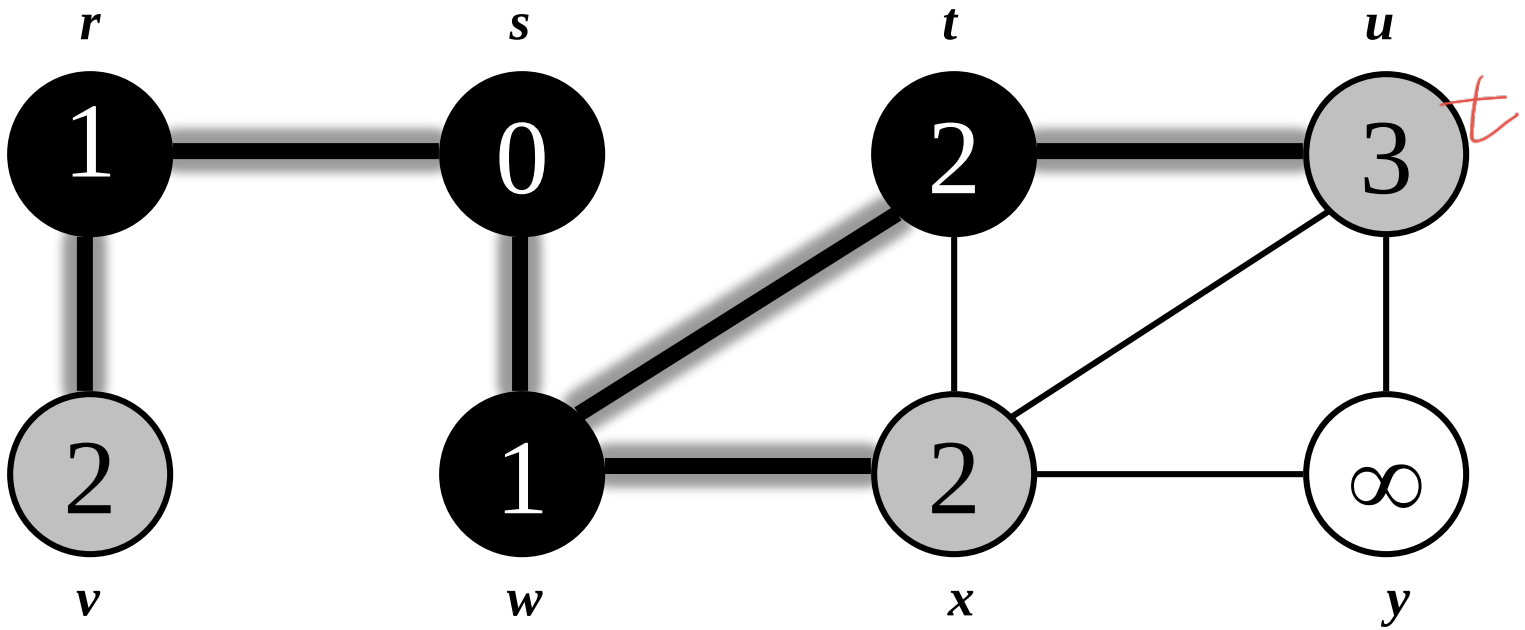
Q: | *r* | *t* | *x* |

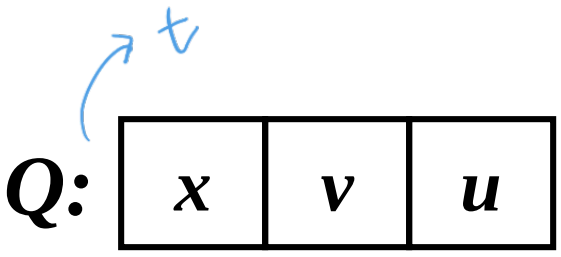
Breadth-First Search: Example



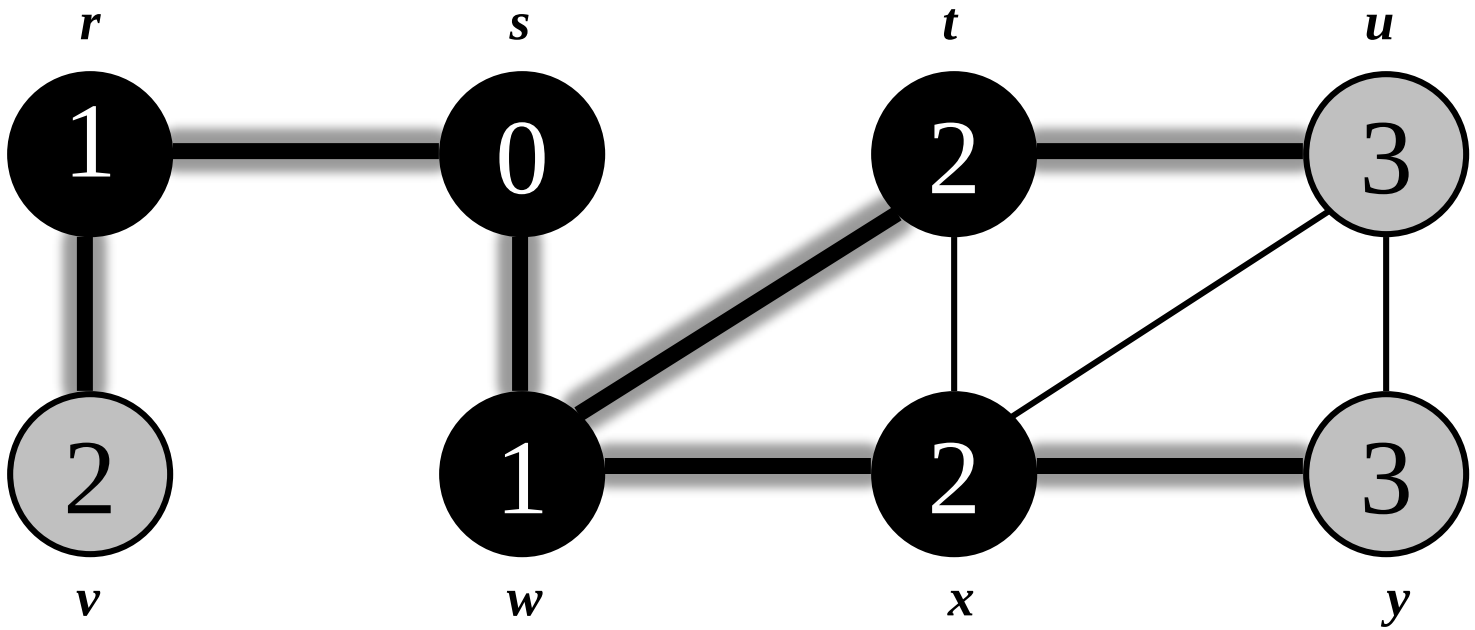
Q: | *t* | *x* | *v* |

Breadth-First Search: Example



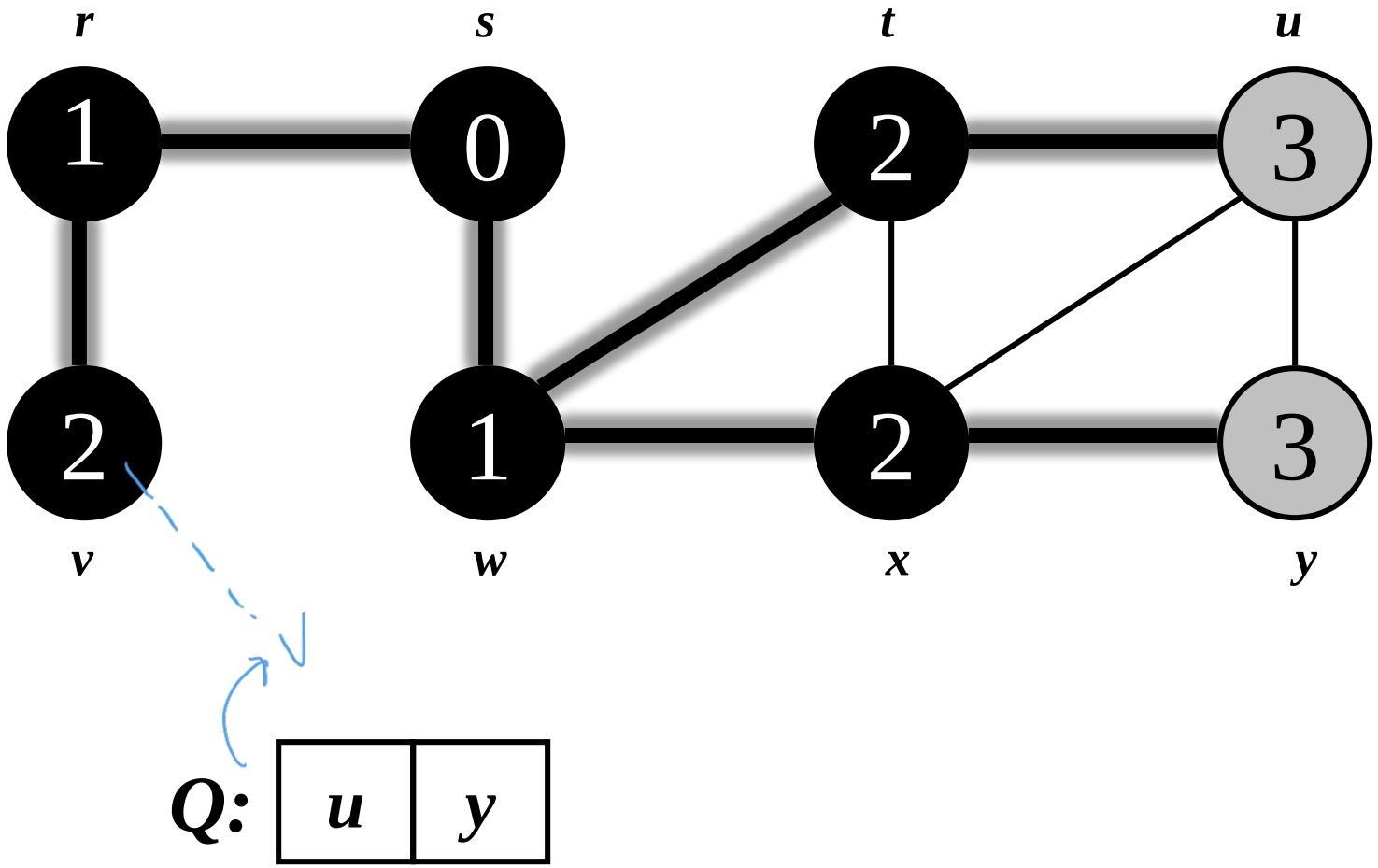


Breadth-First Search: Example

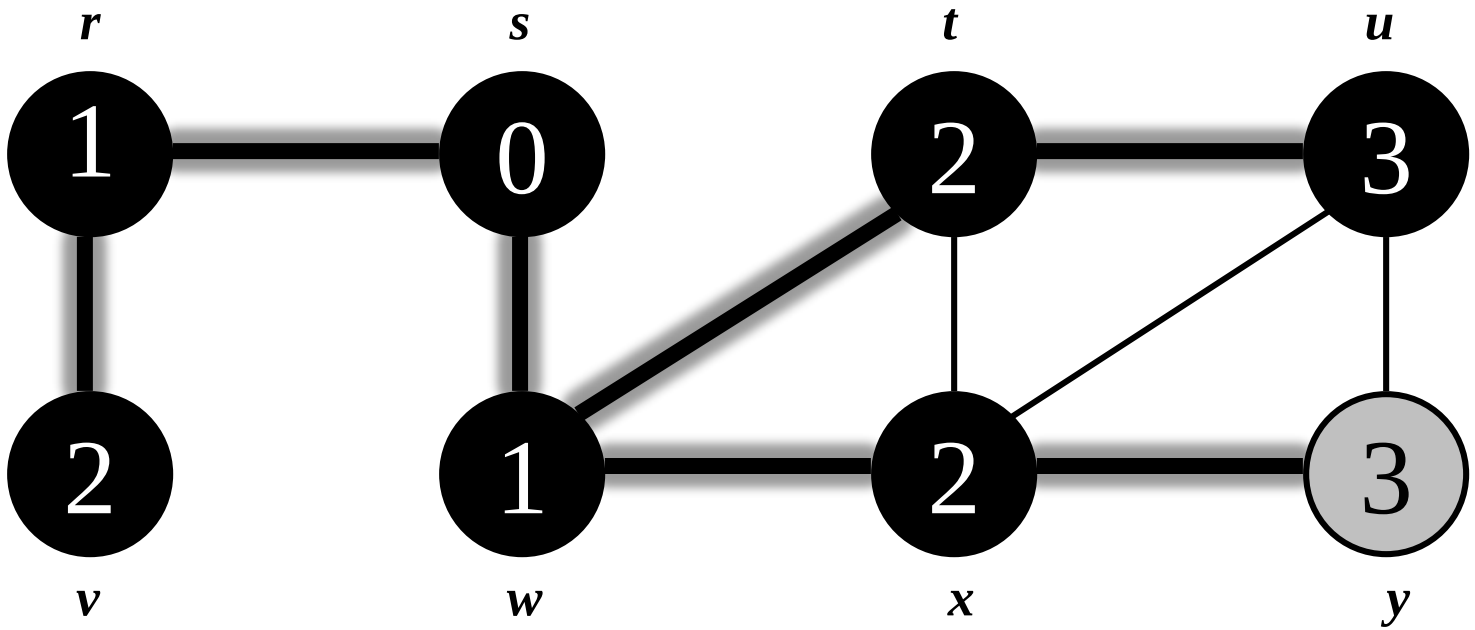


Q: | v | u | y |

Breadth-First Search: Example

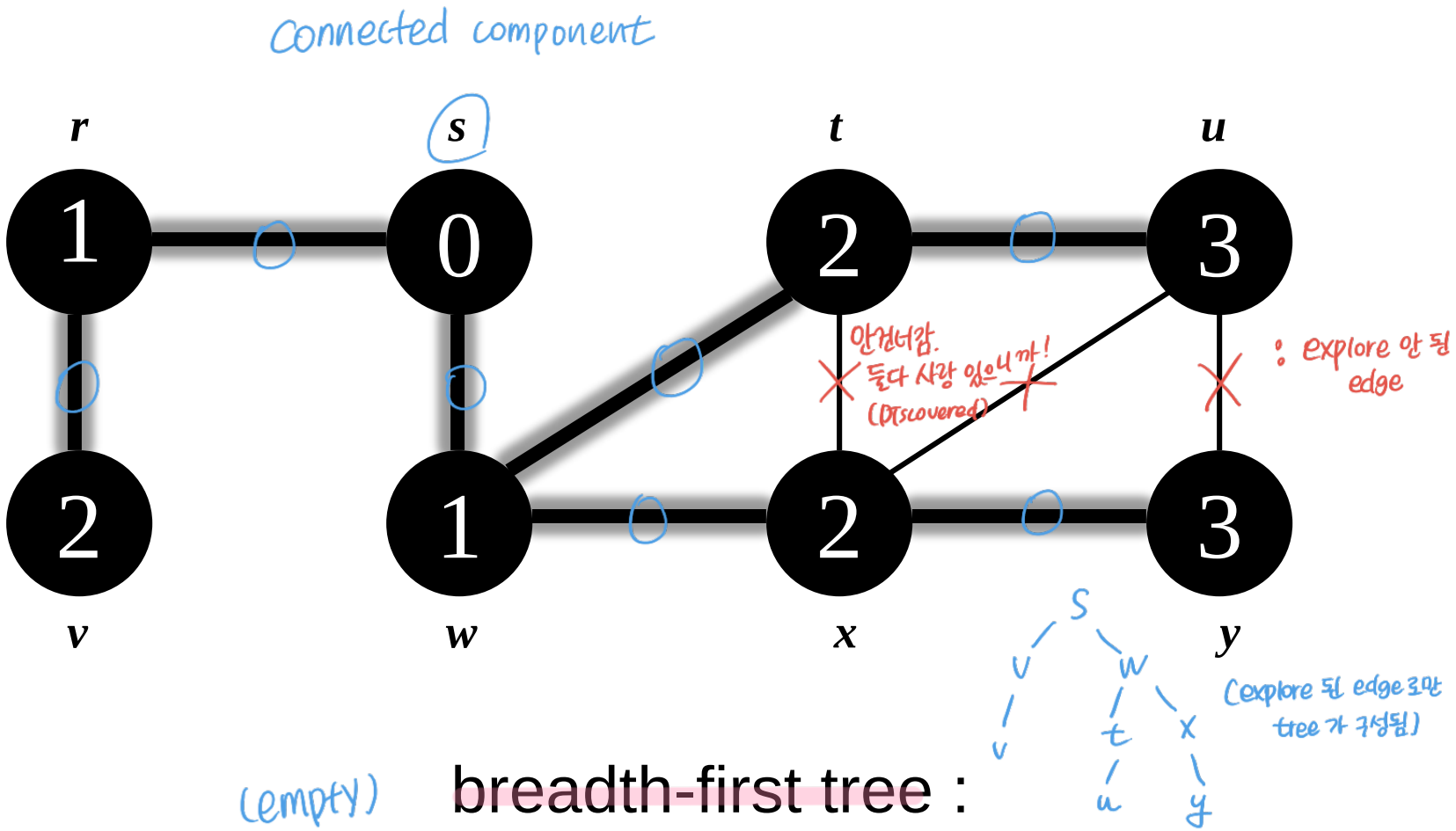


Breadth-First Search: Example



Q: y

Breadth-First Search: Example



breadth-first tree : (empty)

Q: Ø

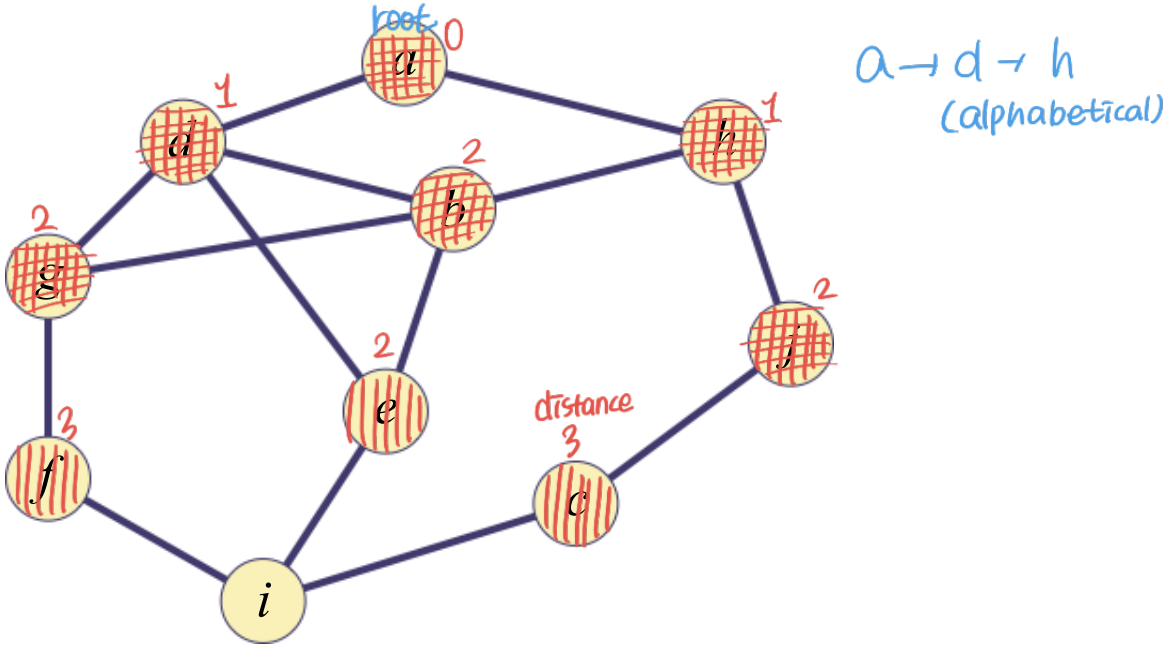
consists of vertices and explored edges

Exercise

For the following graph,

1) Build breadth-first (spanning) tree with vertex '*a*' as root. Assume the vertices are in alphabetical order in the *Adj* array and that each adjacency list is in alphabetical order.

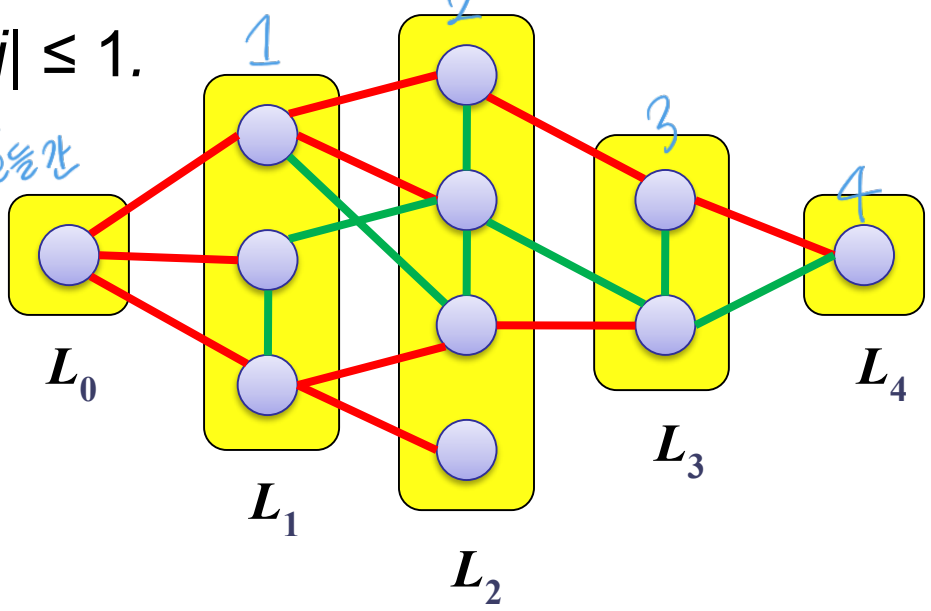
2) What is the minimum distance of vertex *c* from the root?



Properties of Breadth-First Search

Theorem: *Breadth-first search visits the vertices of G by increasing distance from the root. BFS builds breadth-first tree, in which paths to root represent shortest paths in G*

Theorem: *For every edge* (v, w) *in G such that* $v \in L_i$ *and* $w \in L_j$, $|i - j| \leq 1$.



(cf)
BFS
level씩 탐색!

DFS

Depth First Search
깊이 우선 탐색
(내려갔다 올라오고)

Depth-first search

- *Depth-first search* is another strategy for exploring a graph.
  - Explore "deeper" in the graph whenever possible.
  - Pick a *source vertex, s* to be the root.
  - Edges are explored out of the most recently discovered vertex *v* that still has unexplored edges.
  - When all of *v*'s edges have been explored, *backtrack* to the vertex from which *v* was discovered.
  - When one dfs tree is finished, start over from undiscovered vertex as necessary.

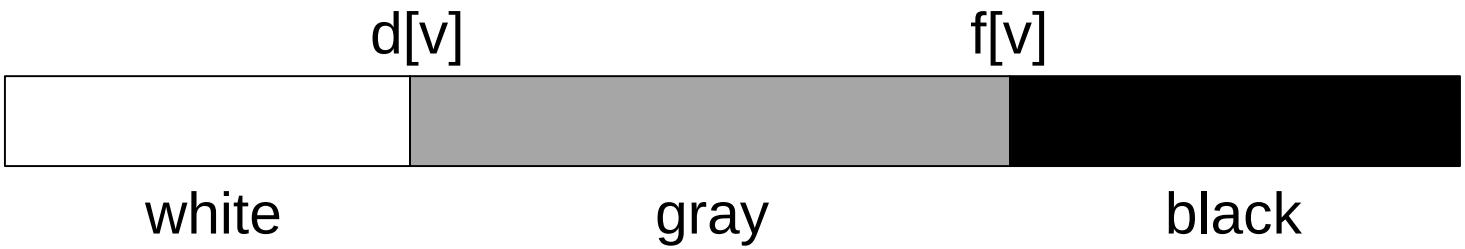
(source) (deeper whenever possible) tree를 이미 만들었는데 undiscovered 있으면, 다시 시작!

Depth-first search

- As soon as we discover a vertex, explore from it. It isn't like BFS, which puts a vertex on a queue so that we explore from it later.
- When the adjacent vertex is already discovered, the edge is not explored (but checked).
- As DFS progresses, every vertex has a color.
  - WHITE : undiscovered
  - GRAY : discovered, but not finished (not done exploring from it.)
  - BLACK : finished (have found everything reachable from it.)

Depth-first search

- Input : $G = (V, E)$, directed or undirected.
- Output : 2 timestamps on each vertex :
  - $d[v]$ = discovery time
  - $f[v]$ = finish time
- Discovery and finish times :
  - Unique integer from 1 to $2|V|$
  - For all v, $d[v] < f[v]$.
  - $1 \le d[v] < f[v] \le 2|V|$



Depth-first search

: stack (when backtrack) ↔ BFS : queue

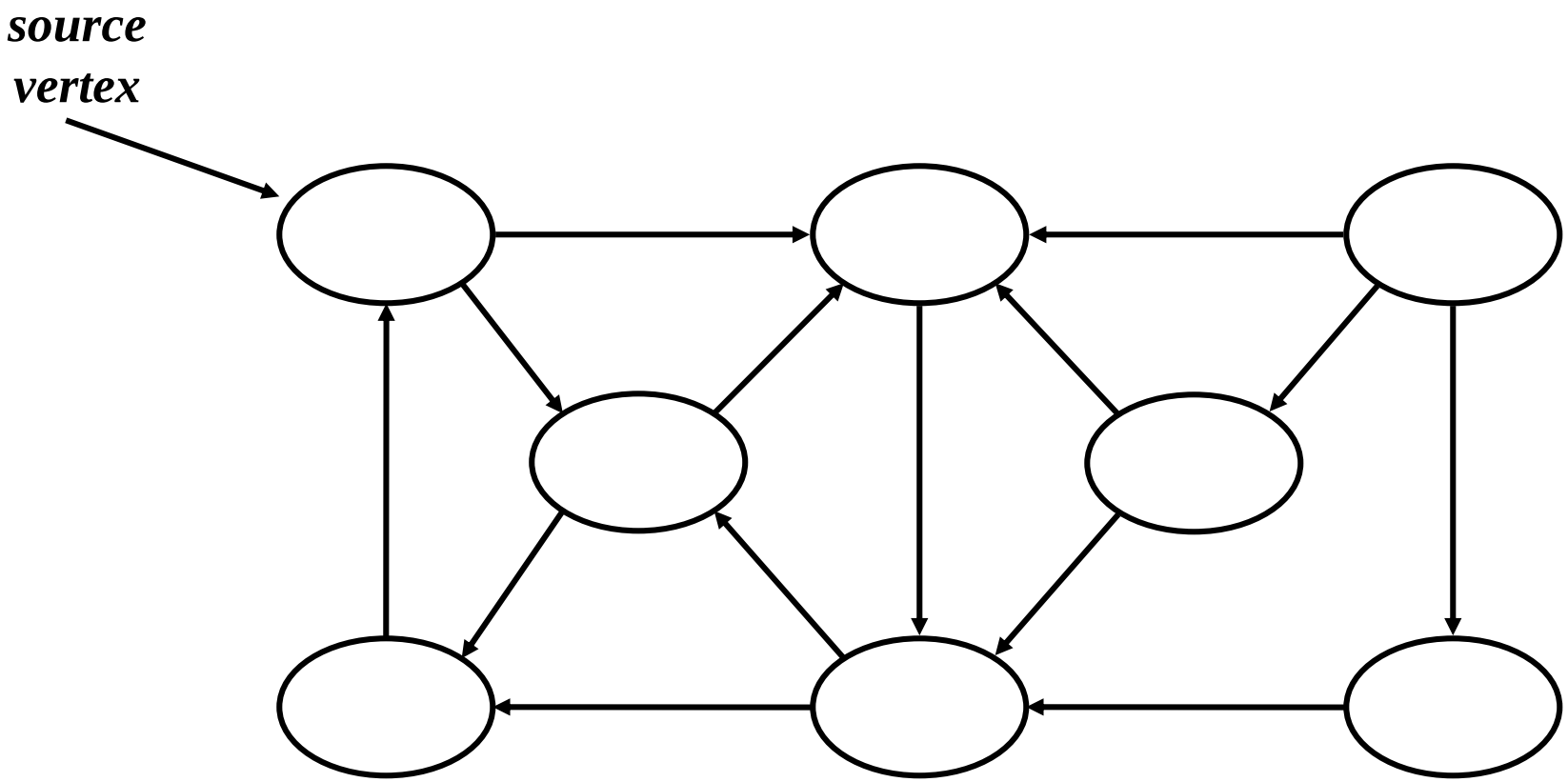
```
DFS(V, E)
for each u ∈ V
   do color[u] = WHITE
time = 0  (global)
for each u ∈ V
     do if color[u] = WHITE
           then DFS-VISIT(u)
```

```
DFS-VISIT(u)        <discovery>
color[u] = GRAY
time = time + 1  (global)
d[u] = time
discovery
for each v ∈ Adj[u]
        do if color[v] = WHITE
              then DFS-VISIT(v)
                   (recursively call)
color[u] = BLACK  다 끝나면 black으로
time = time + 1   <finish>
f[u] = time
finish time
```

- uses a global timestamp *time*

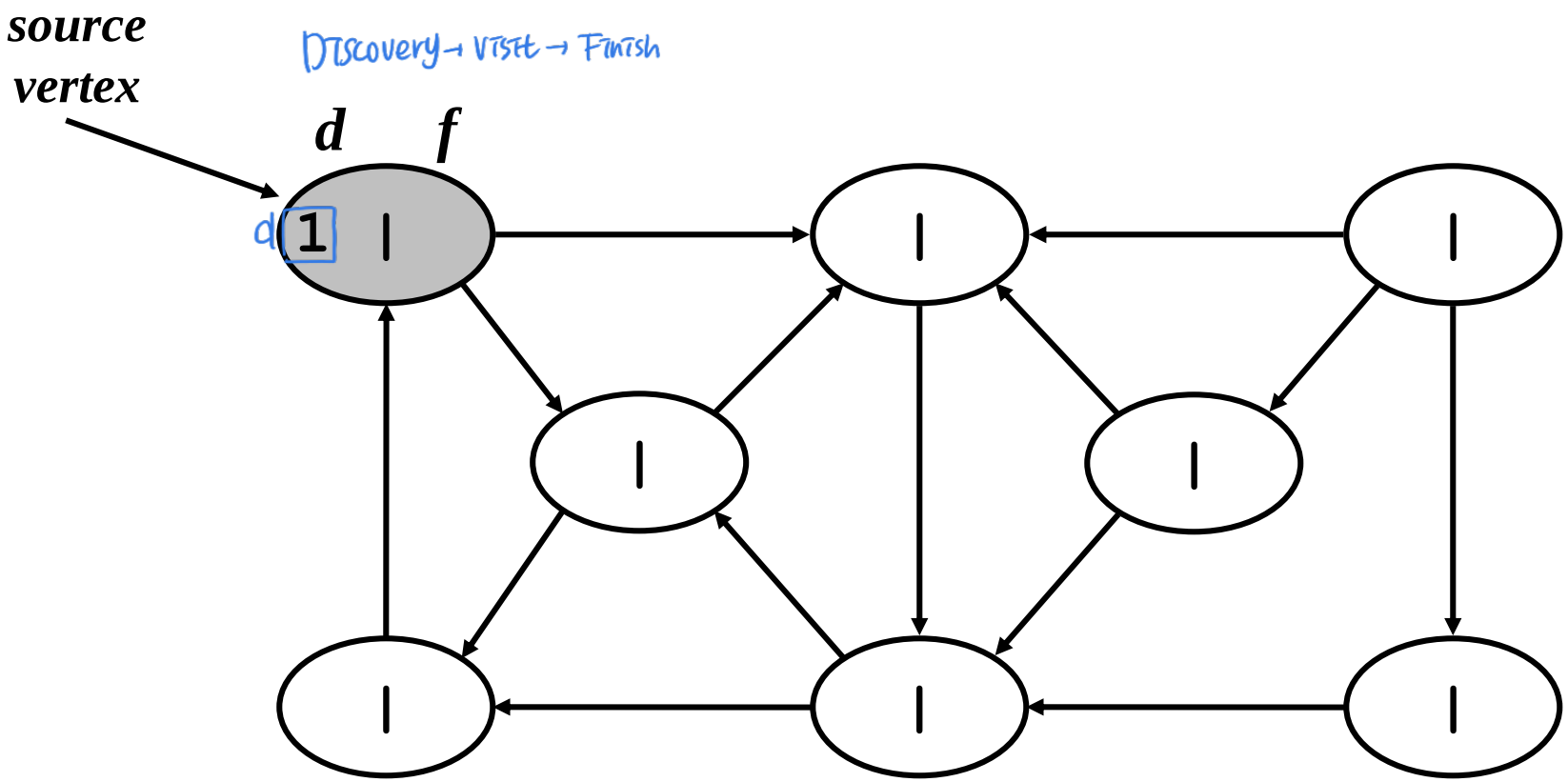
DFS Example

(BFS : 많은 사람, DFS : 한 사람)

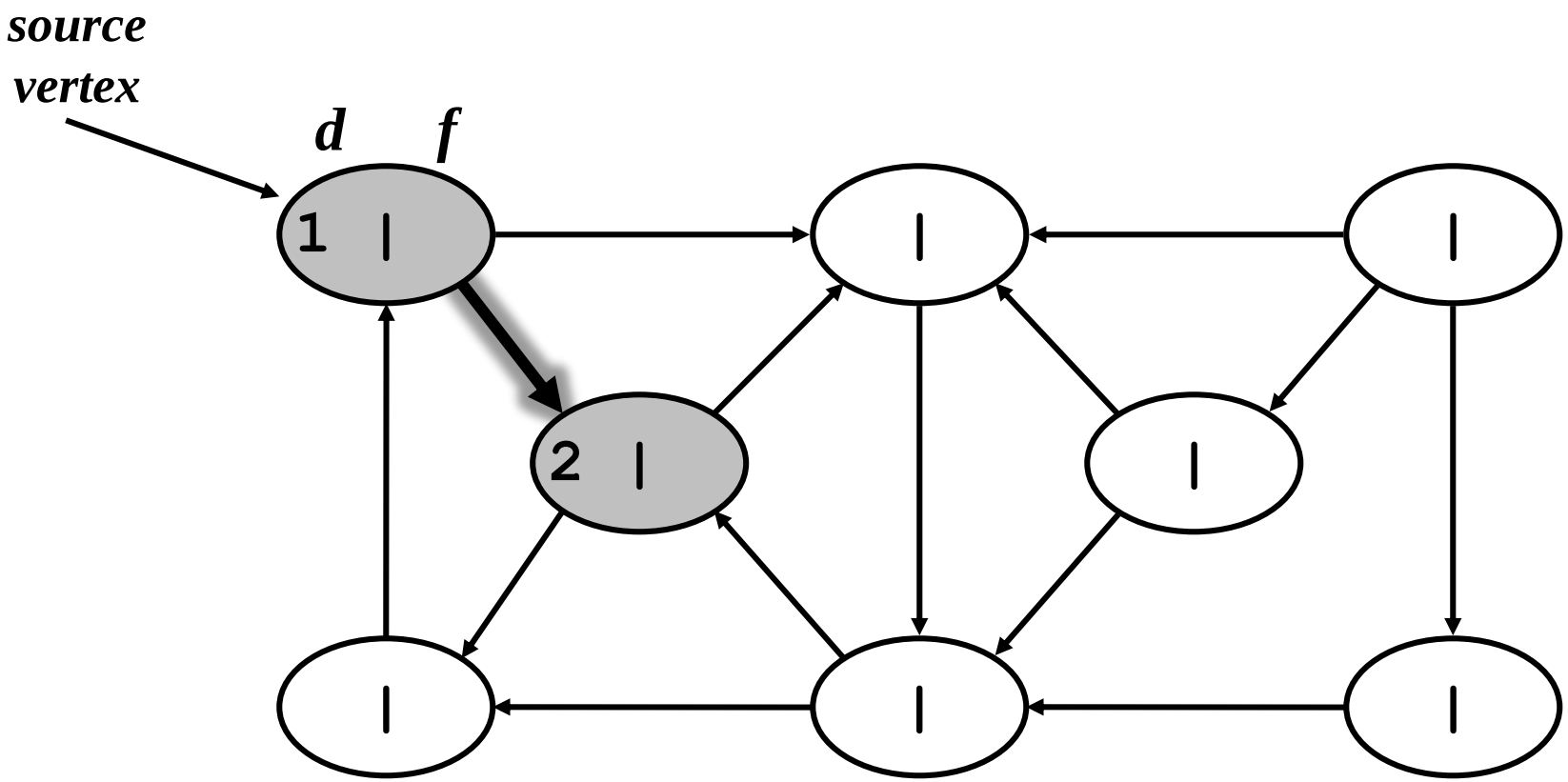


One person explores the islands. If an island is not discovered yet, walk in the direction as arrows. If one walk across the bridge in the reverse direction, one should walk backward – backtracking.

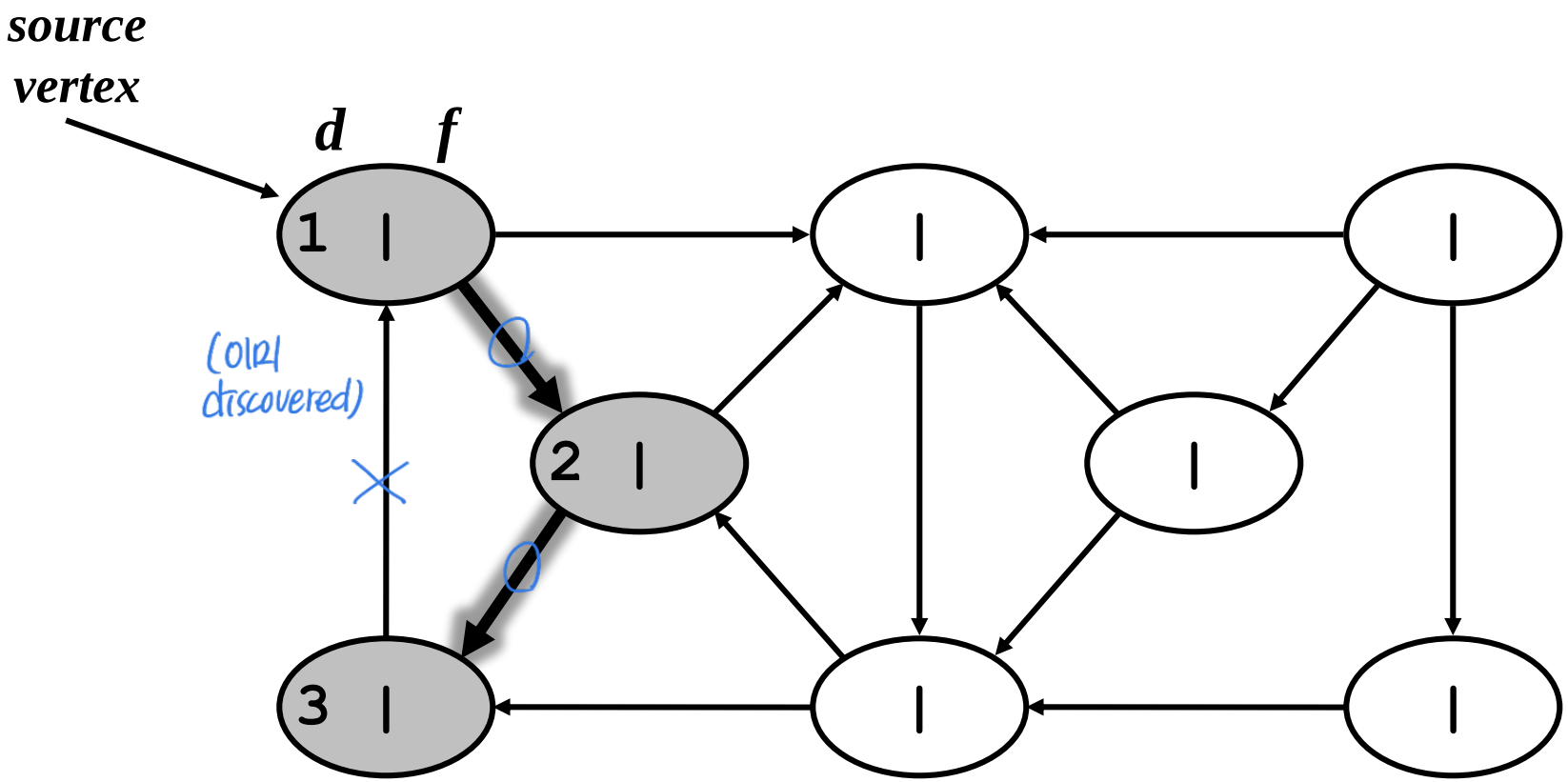
DFS Example



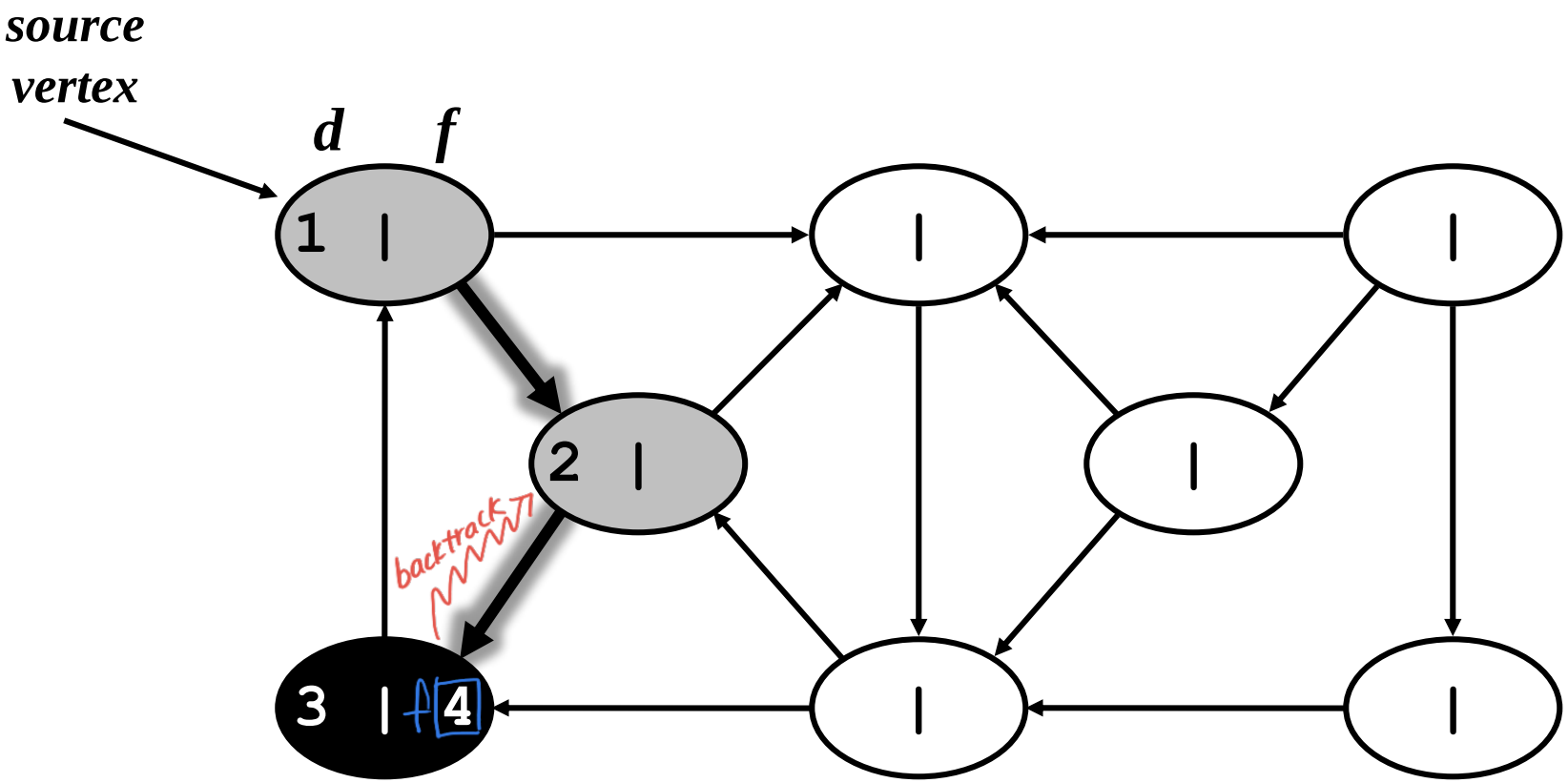
DFS Example



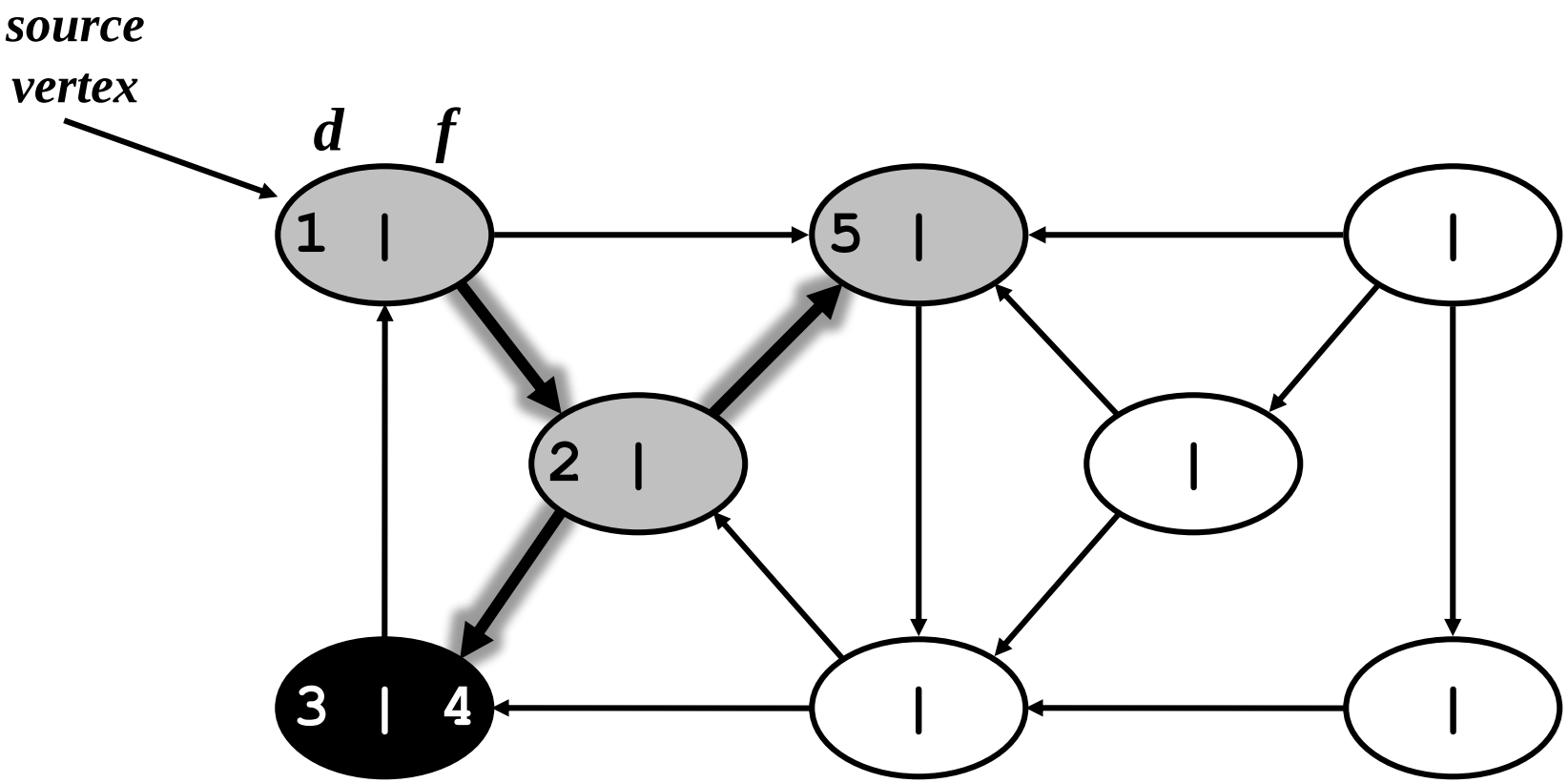
DFS Example



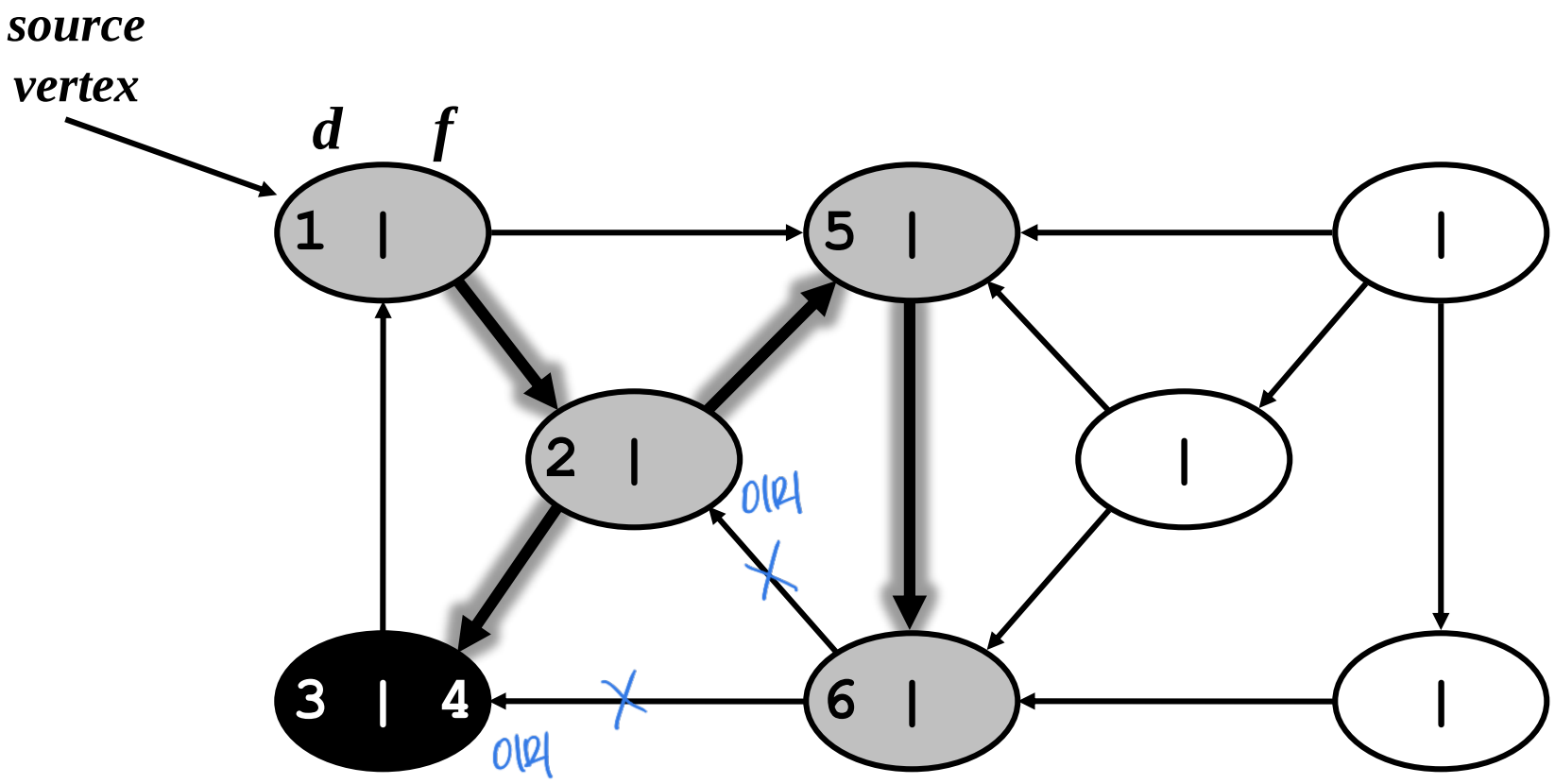
DFS Example



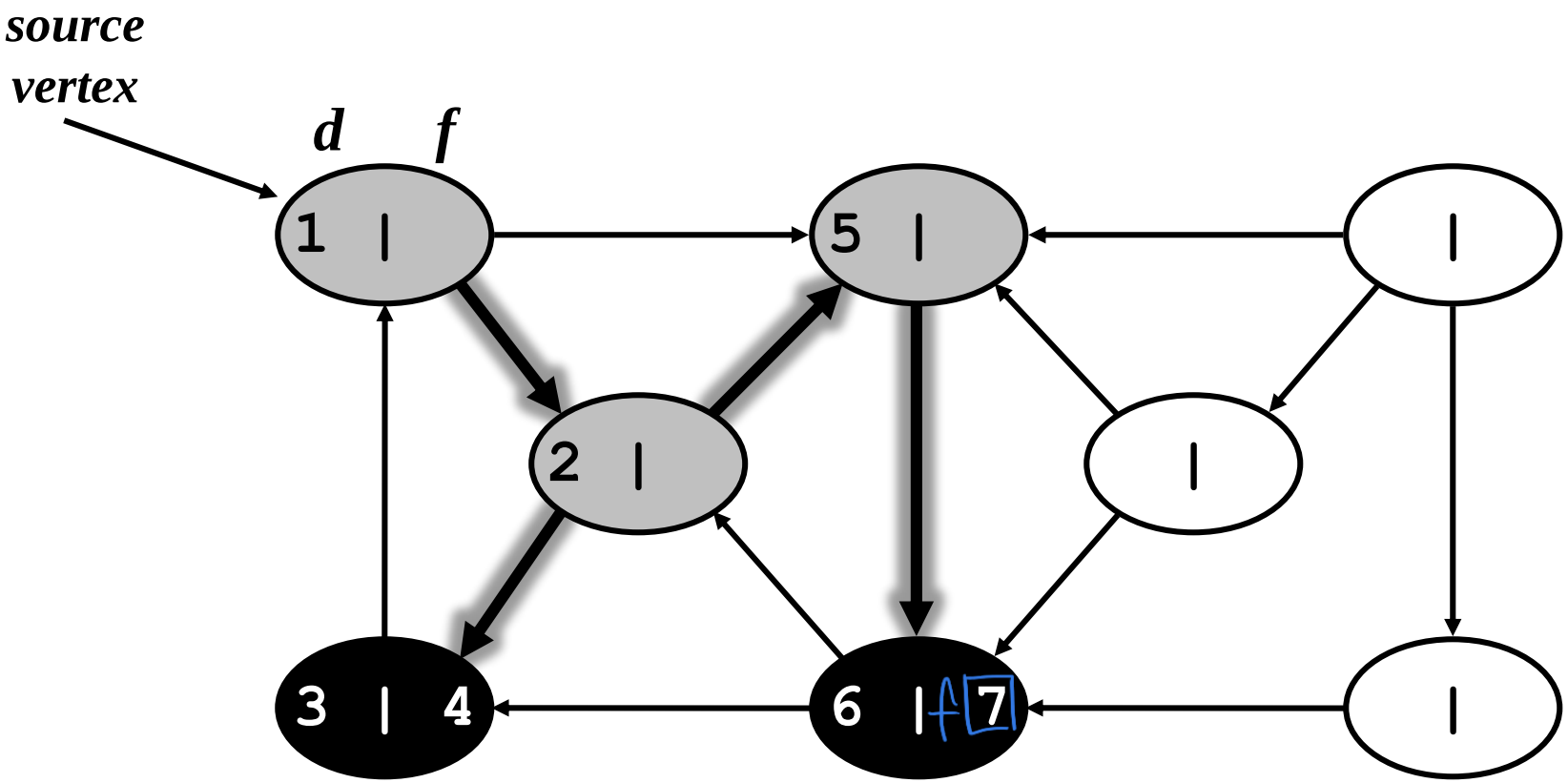
DFS Example



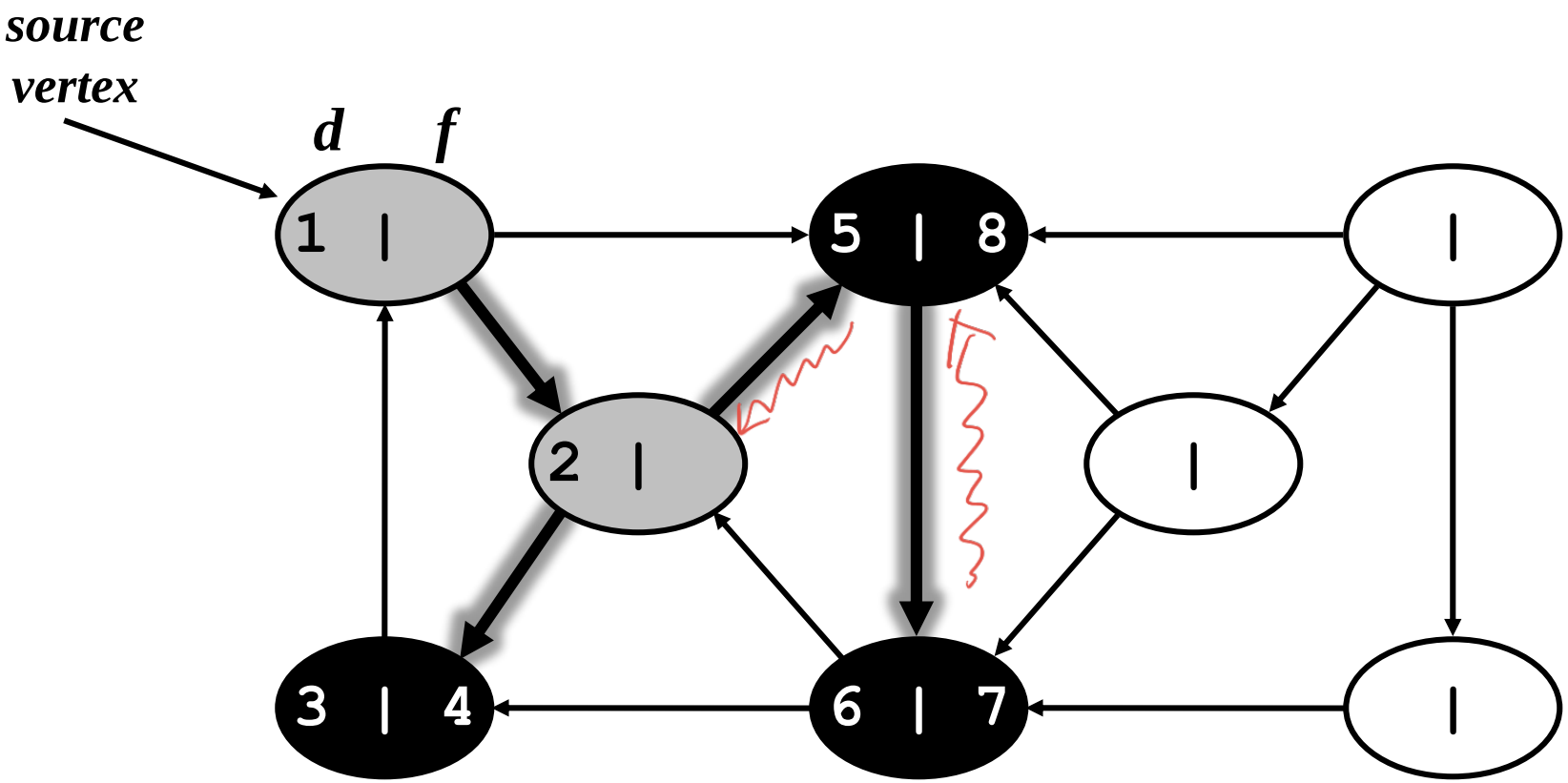
DFS Example



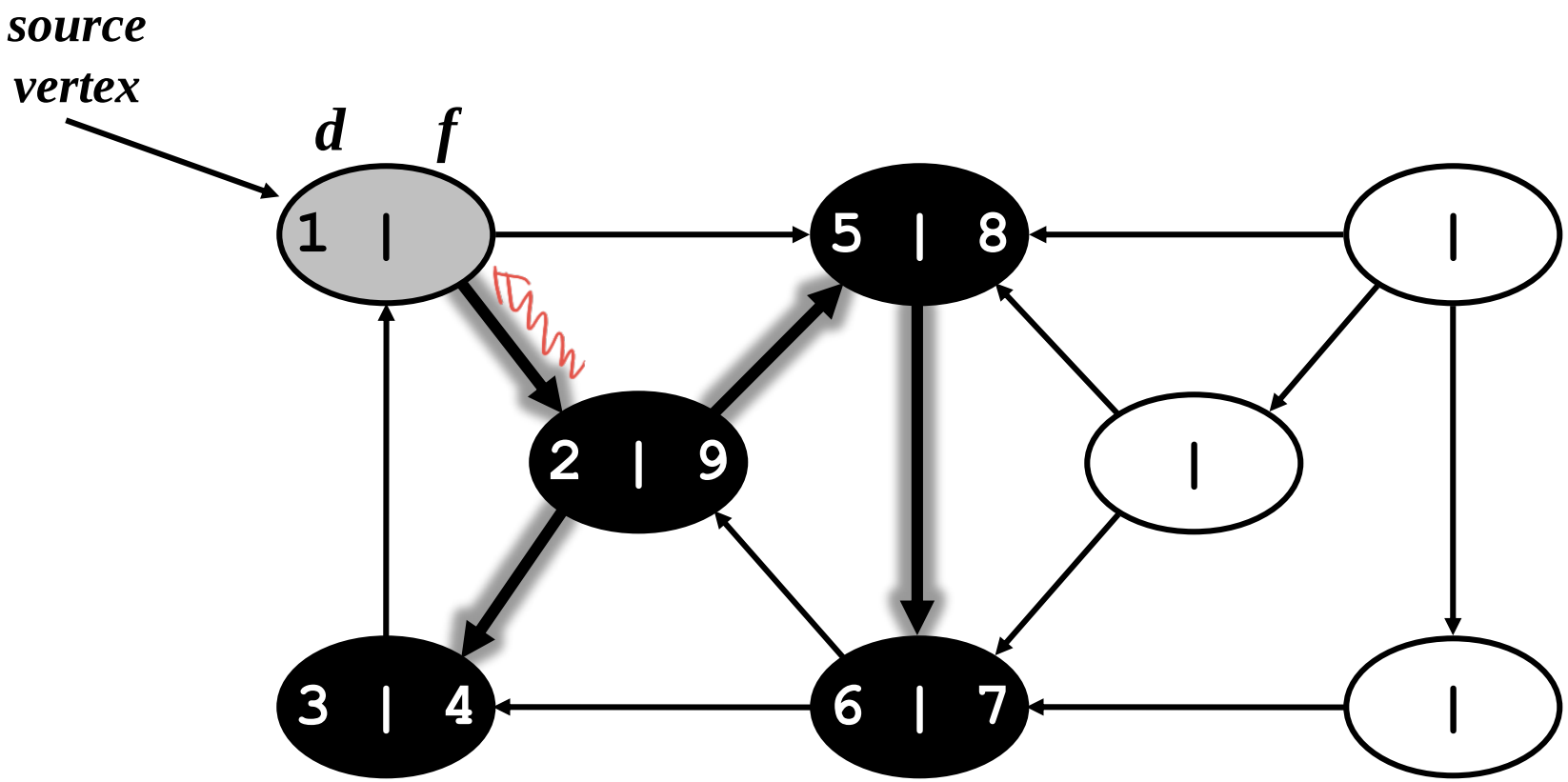
DFS Example



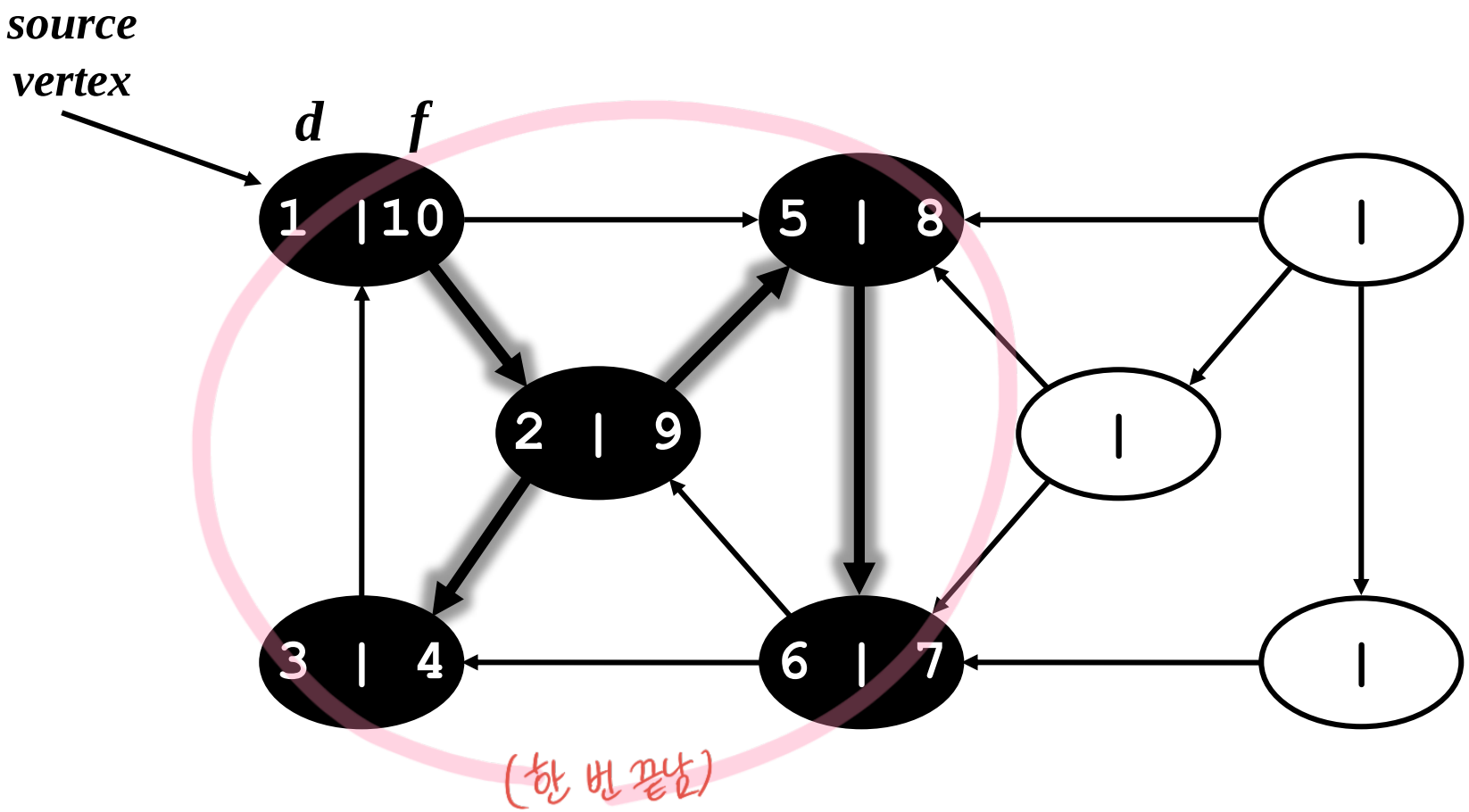
DFS Example



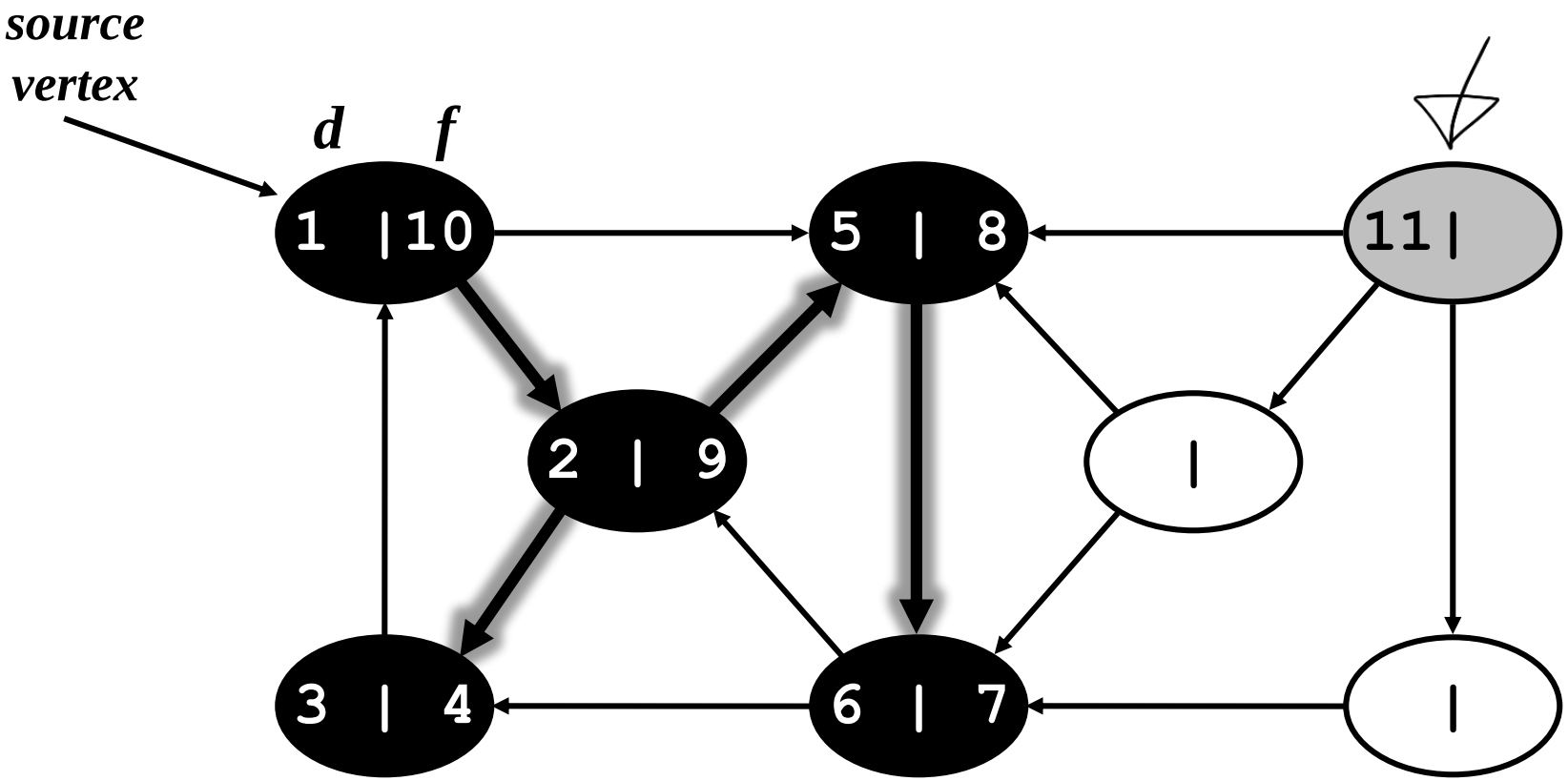
DFS Example



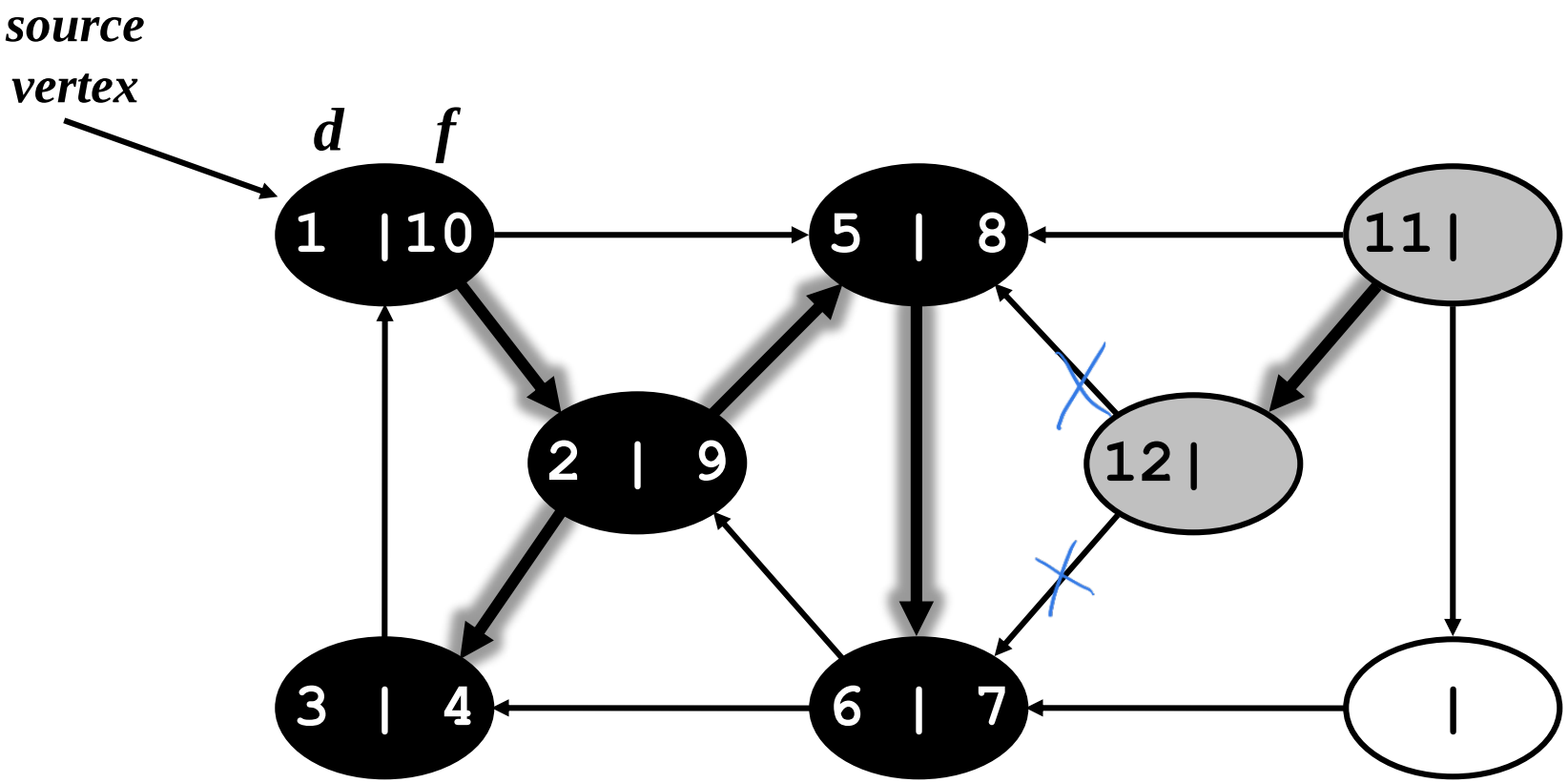
DFS Example



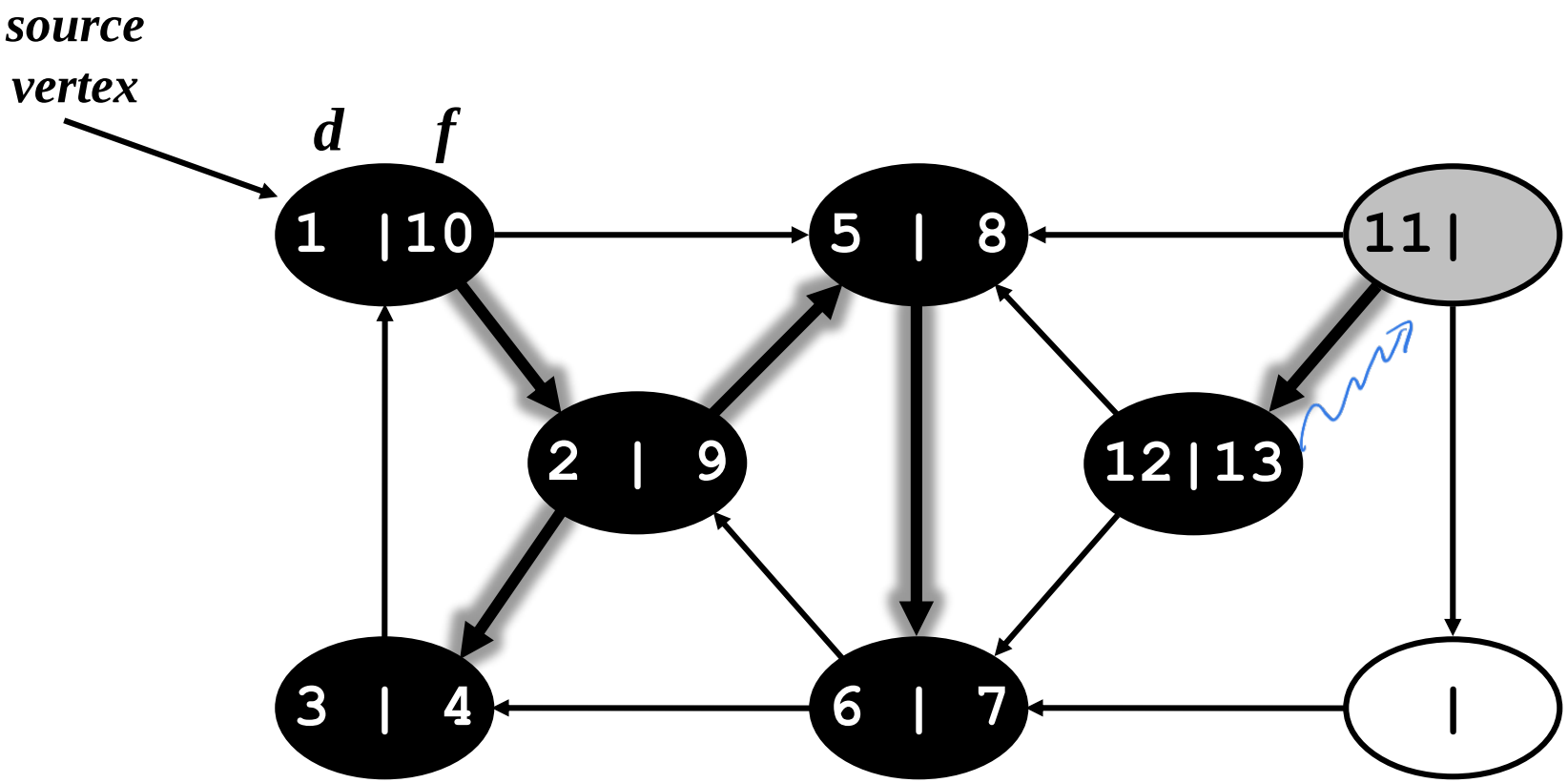
DFS Example



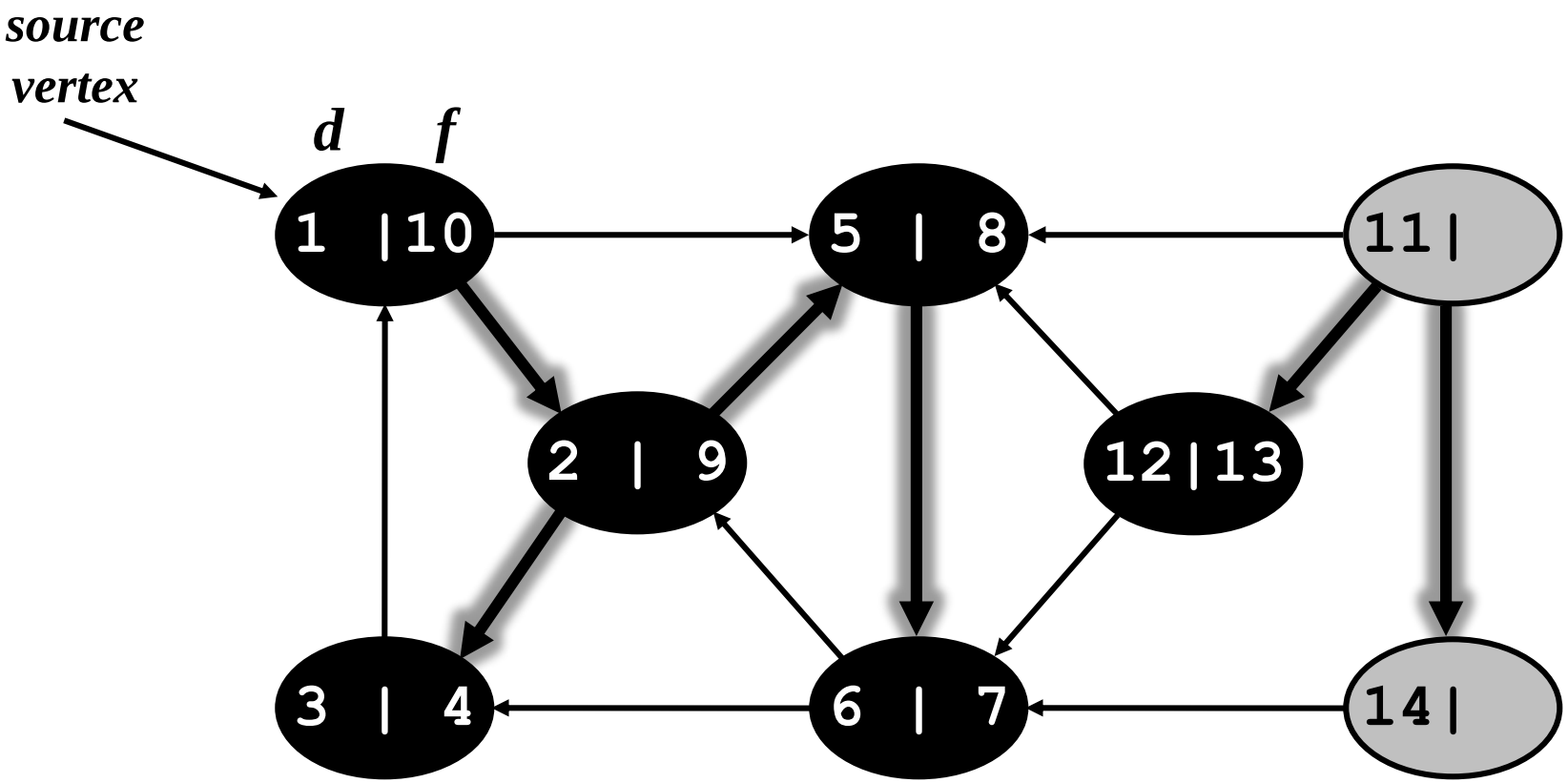
DFS Example



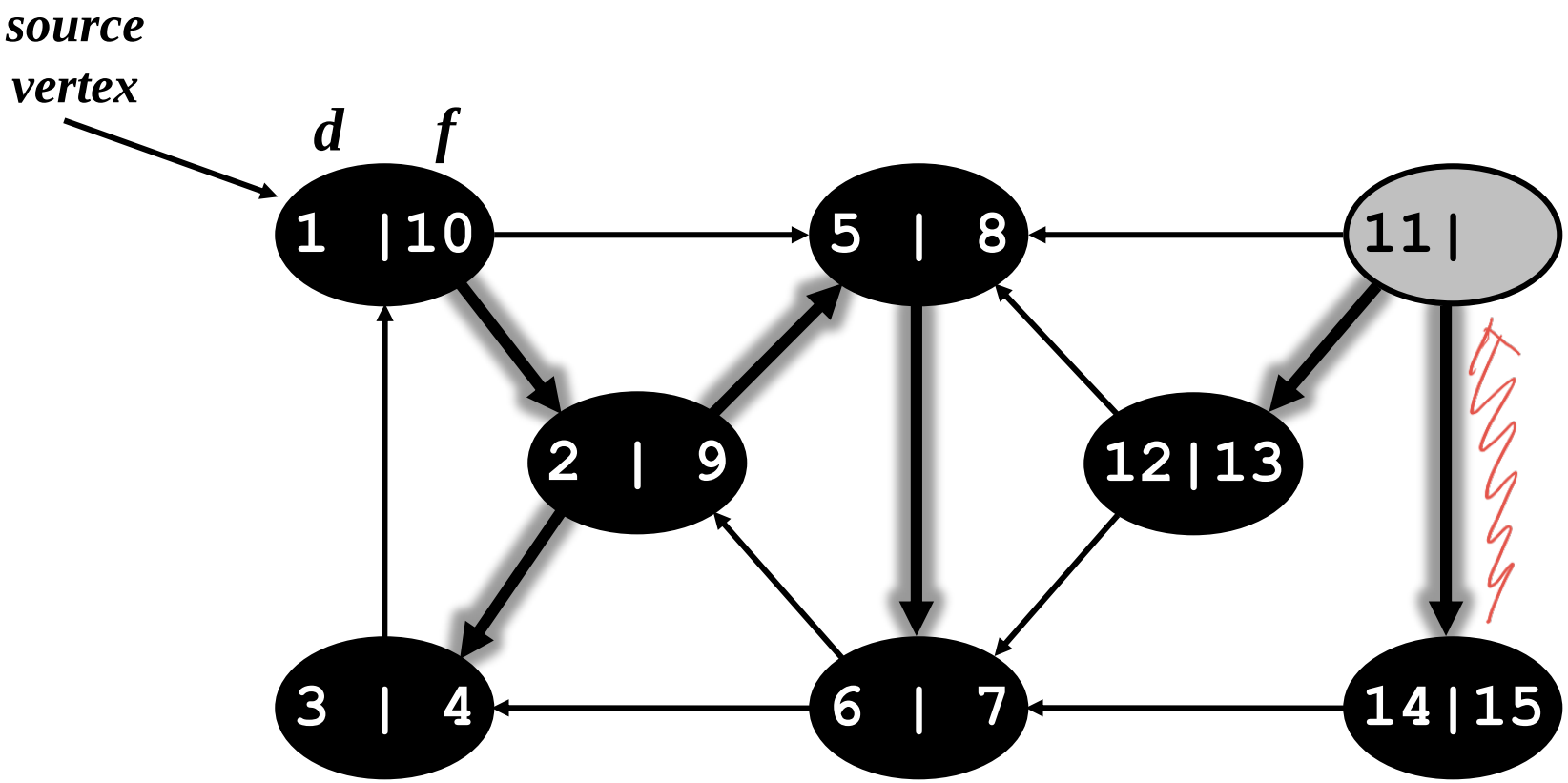
DFS Example



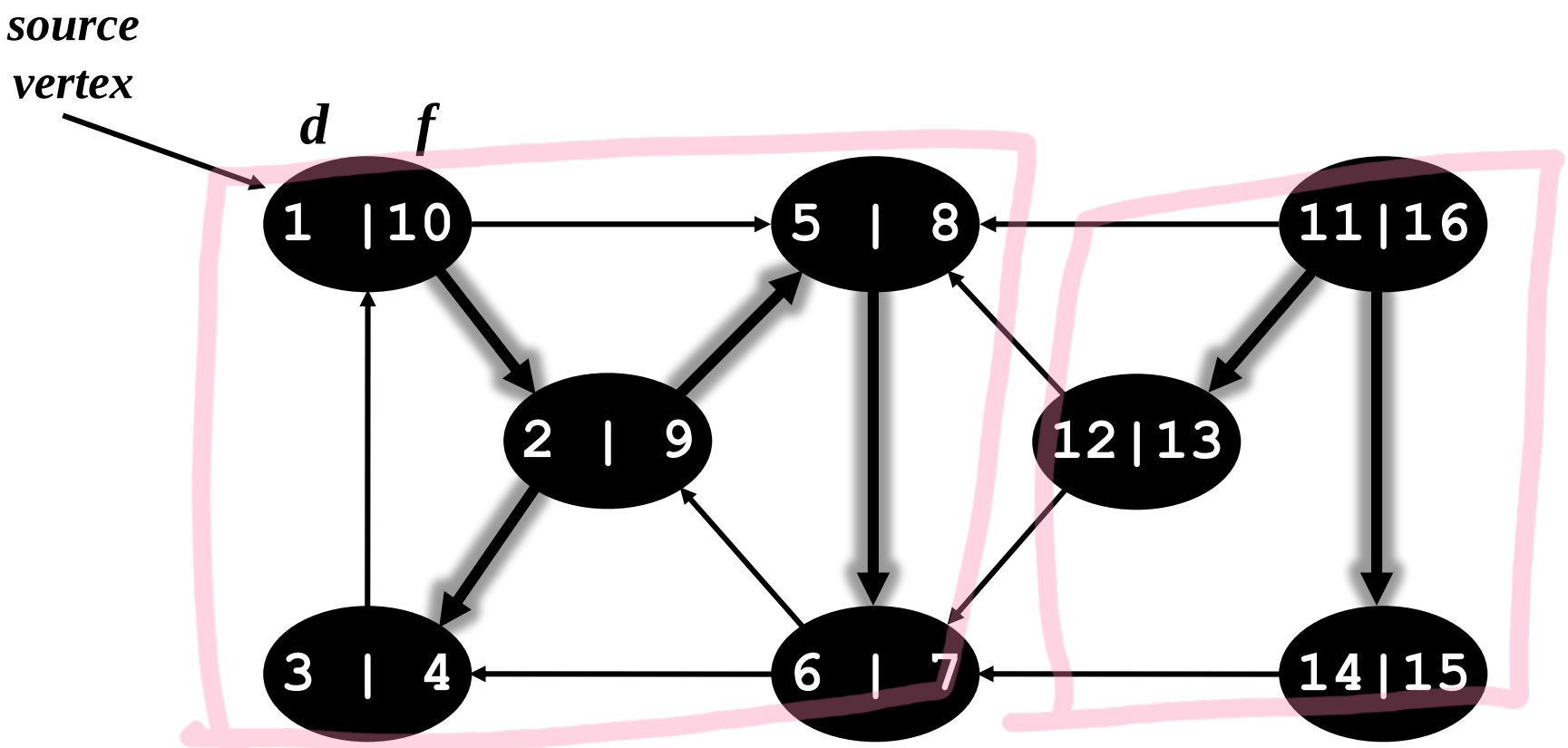
DFS Example



DFS Example

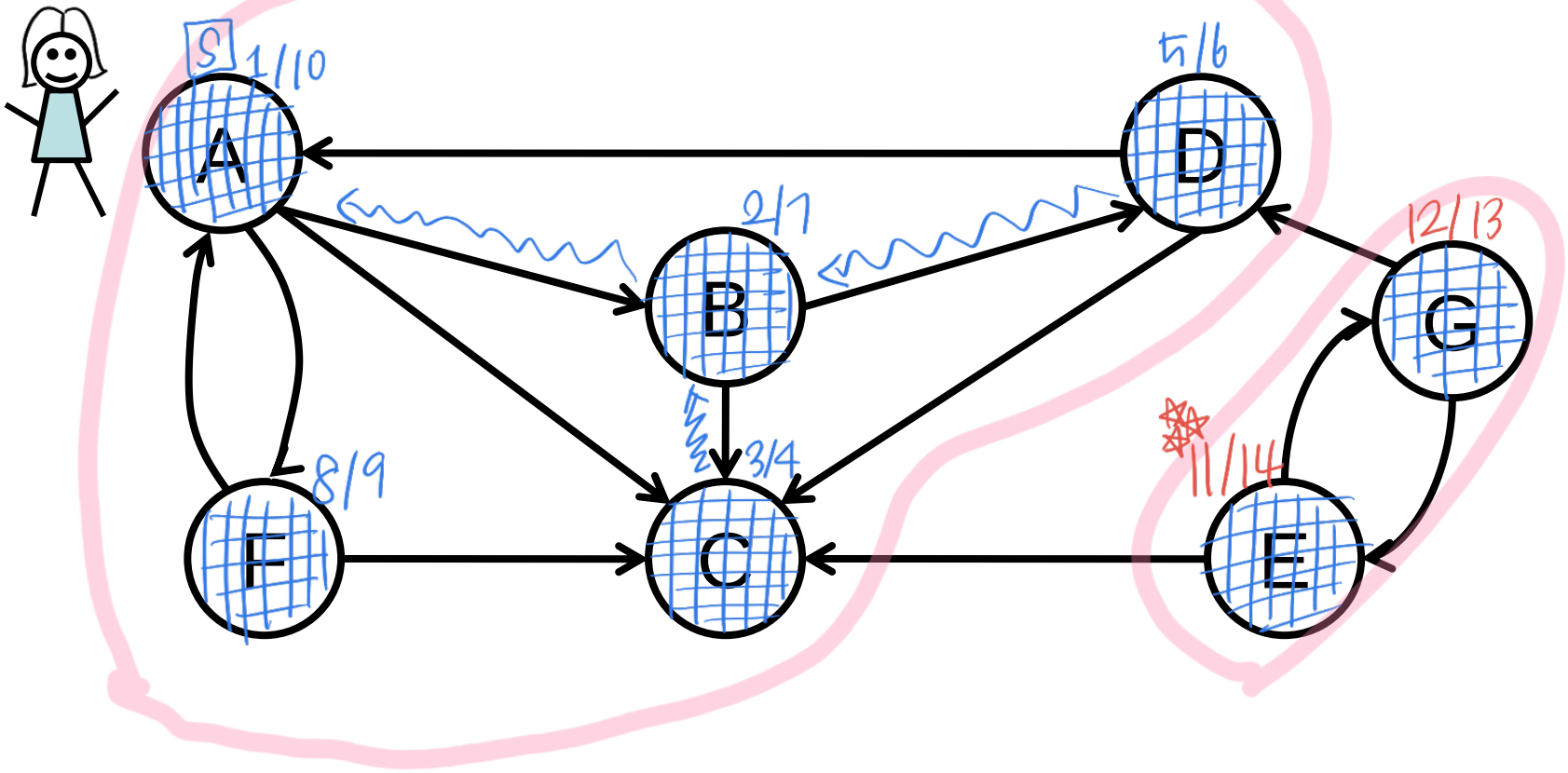


DFS Example



Exercise

Assume the vertices are in alphabetical order in the *Adj* array and that each adjacency list is in alphabetical order.

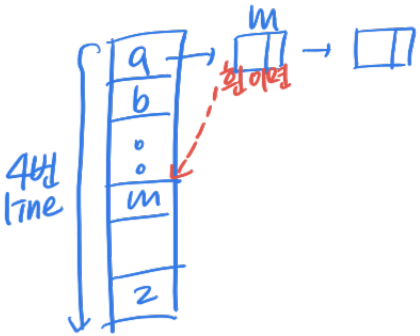


Analysis

```
DFS(G)
1 for each vertex u ∈ V[G]
2    do color[u] ← WHITE
3       π[u] ← NIL
4 time ← 0
5 for each vertex u ∈ V[G]
6    do if color[u] = WHITE
7       then DFS-VISIT(u)
```

Θ(V) — 모든 vertex에 대해

Θ(V)



Each vertex is visited once and each edge is explored or checked once

$\therefore \Theta(V+E)$

```
DFS-VISIT(u)
1 color[u] ←GRAY
2 d[u] ← time ← time + 1
3 for each v ∈ Adj[u]
4    do if color[v] = WHITE
5       then π[v] ← u
6          DFS-VISIT(v)
7 color[u] ← BLACK
8 f[u] ← time ← time + 1
```

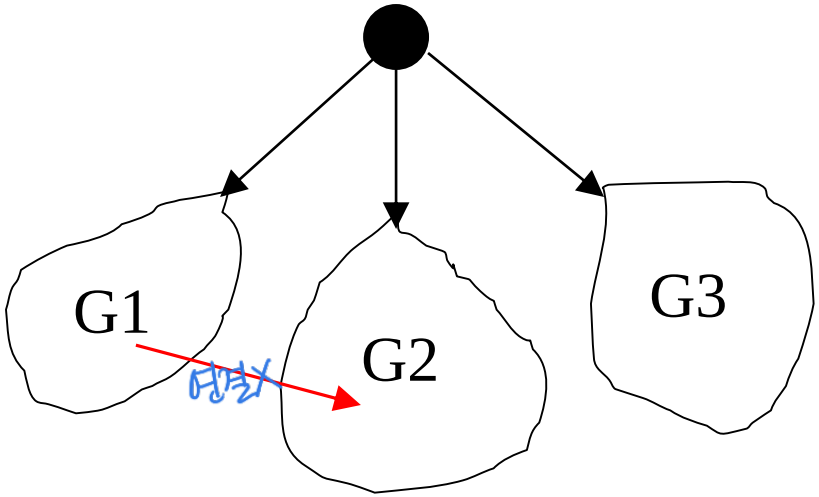
constant

constant

$\sum_{v \in V} |Adj[v]| = \Theta(E)$ — edge 수만큼!

DFS on tree and general graph

- G1, G2, G3 are disjoint set, so we don't have to worry about the possibility of visiting node already visited .

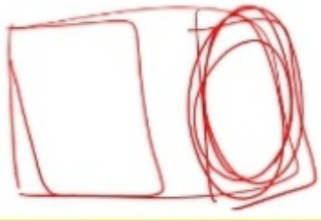


- Edge in a tree is one-way, i.e. no cycle.

(node : n, edge : (n+1) , n+2 connected)

DFS

```
void dfs(int  v){
    node_pointer  w;
    visited[v] = TRUE;
    printf(" %5d", v) ;
    for (w = graph[v]; w; w = w->link)
      /* graph[] : headnodes */
        if(!visited[w->vertex])
            dfs(w->vertex);
}
```



Deapth-first search

Preorder processing of vertex (*d*[*u*] : preorder)

Postorder processing of vertex (*f*[*u*] : postorder)

children 방문

```
DFS-VISIT(u)
color[u] = GRAY
time = time + 1
d[u] = time
for each v ∈ Adj[u]
    do if color[v] = WHITE
            then DFS-VISIT(v)
color[u] = BLACK
time = time + 1
f[u] = time
```

Parenthesis theorem

For all u, v, exactly one of the following holds :

1. $d[u] < f[u] < d[v] < f[v]$ or $d[v] < f[v] < d[u] < f[u]$ and neither of u and v is a descendant of the other.
2. $d[u] < d[v] < f[v] < f[u]$ and v is a descendant of u.
3. $d[v] < d[u] < f[u] < f[v]$ and u is a descendant of v.

So $d[u] < d[v] < f[u] < f[v]$ cannot happen.

Like parentheses :

()[] , ([]), [()] are O.K., but ([)] [(]) are not O.K.

∴ Corollary :

v is a proper descendant of u iff $d[u] < d[v] < f[v] < f[u]$.

Parenthesis theorem

Proof

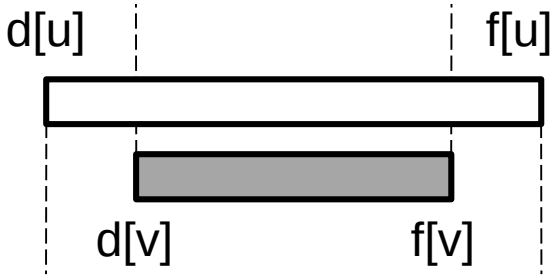
Assume that $d[u] < d[v]$ (because of symmetry)

① $d[v] < f[u]$

v was discovered while *u* was gray, and this implies that *v* is a descendant of *u*. Moreover, since *v* was discovered more recently than *u*, all of its outgoing edges are explored, and *v* is finished, before the search returns to and finishes *u*.

So, $f[v] < f[u]$.

$\therefore d[u] < d[v] < f[v] < f[u]$



Parenthesis theorem

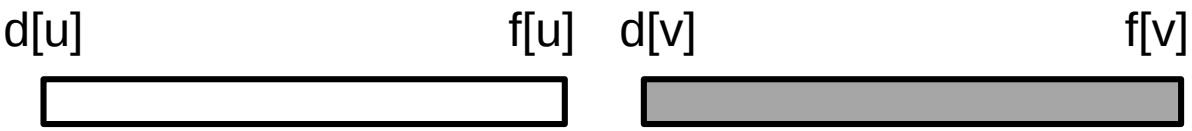
So, $f[v] < f[u]$.

$\therefore d[u] < d[v] < f[v] < f[u]$.

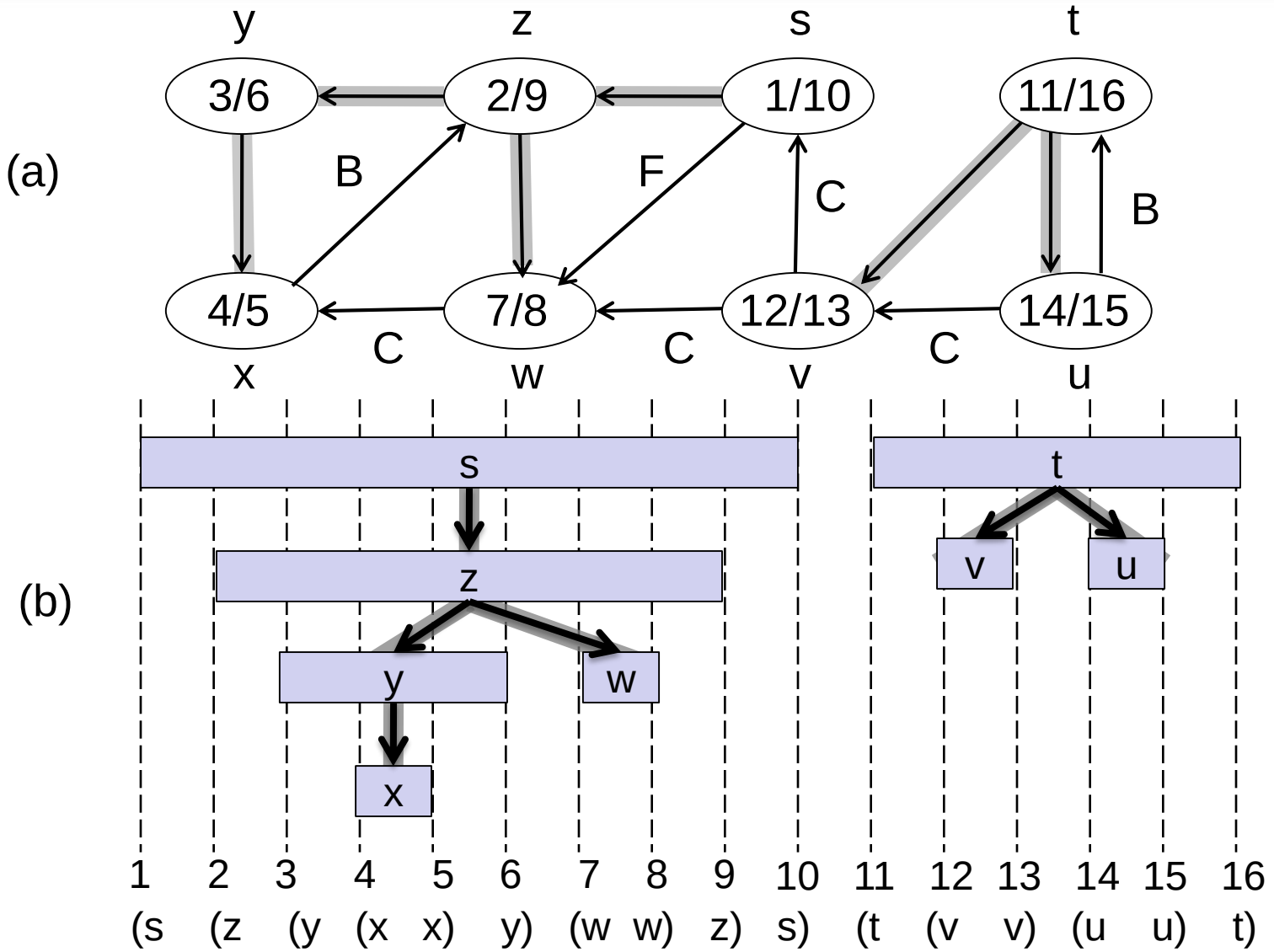
Thus, the interval $[d[v], f[v]]$ is entirely contained within the interval $[d[u], f[u]]$.

② $d[v] > f[u]$ (disjoint intervals)

$\therefore d[u] < f[u] < d[v] < f[v]$



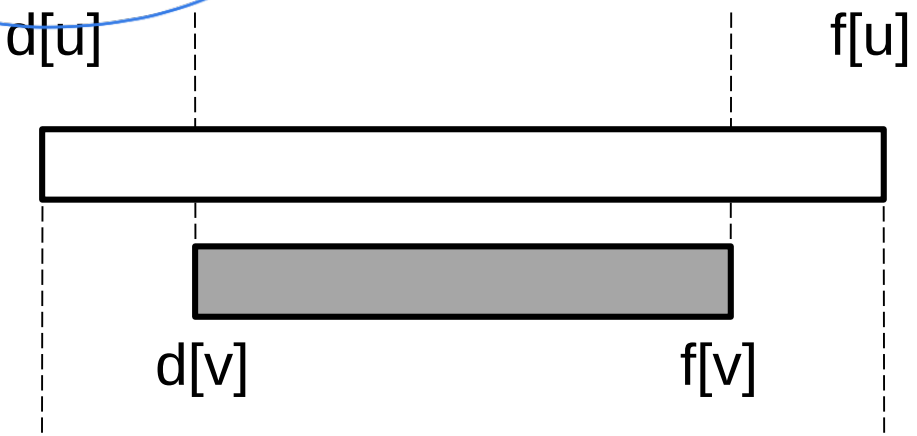
Parenthesis theorem



Parenthesis theorem

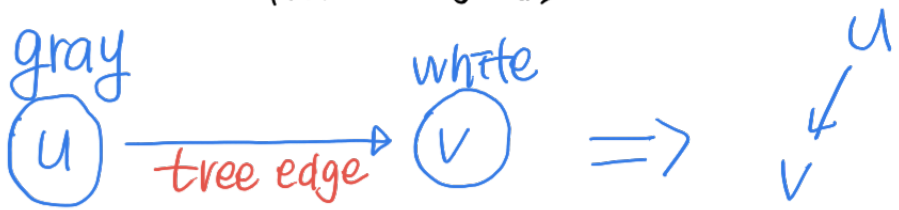
- Nesting of descendants' interval (Corollary 22.8)
  - Vertex *v* is a proper descendant of vertex *u* in the Depth-First Forest (same tree) for a graph *G* if and only if :

$$d[u] < d[v] < f[v] < f[u]$$

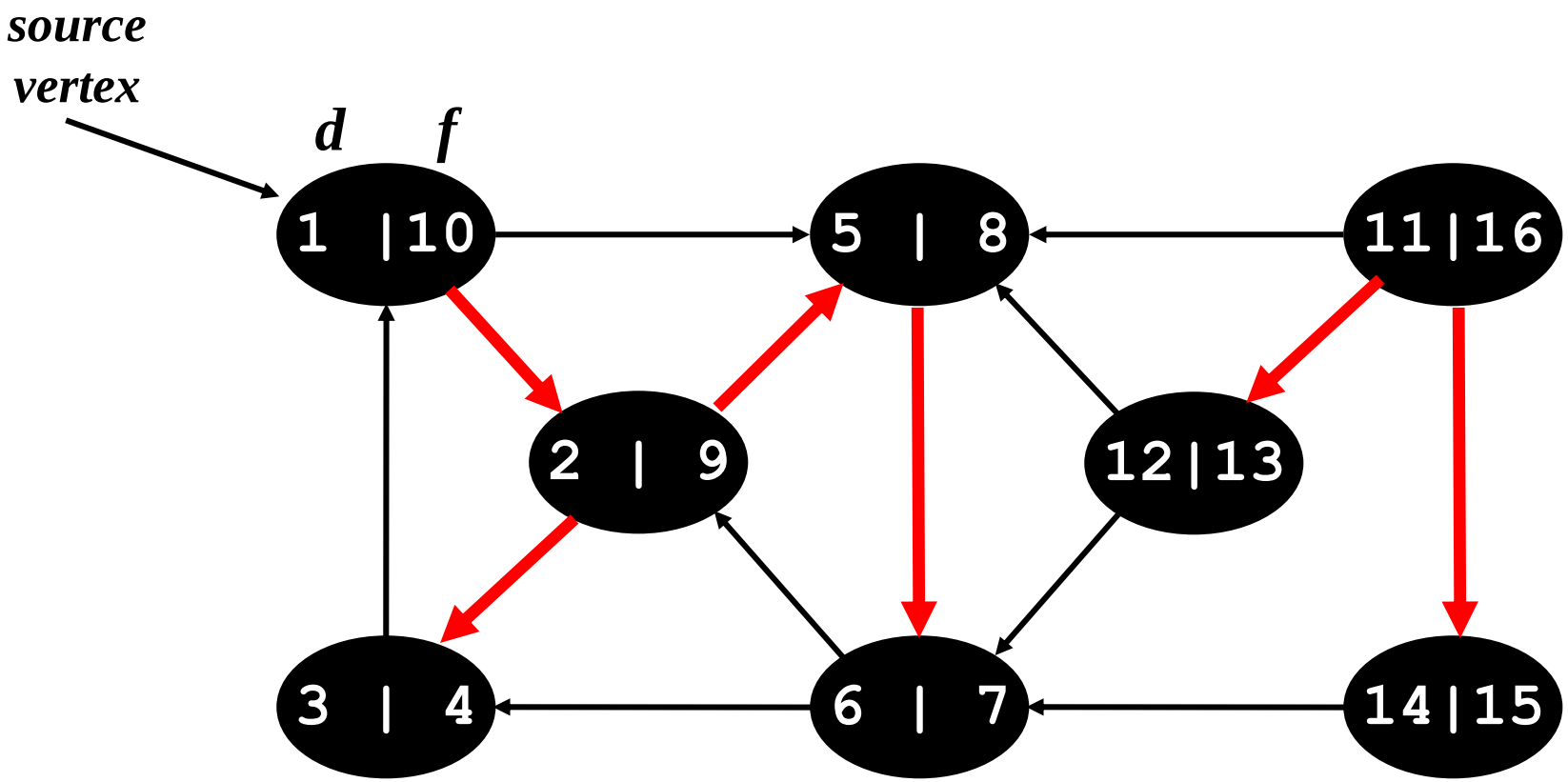


DFS: Kinds of edges

- DFS introduces an important distinction among edges in the original graph:
  - *Tree edge*: encounter new (white) vertex
    - The tree edges form a spanning forest (explored)
    - *Can tree edges form cycles? Why or why not?*



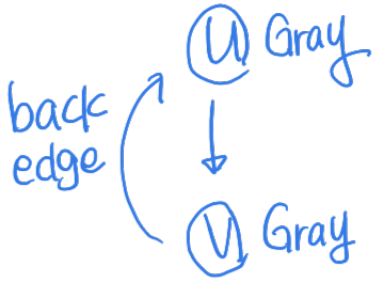
Tree edge



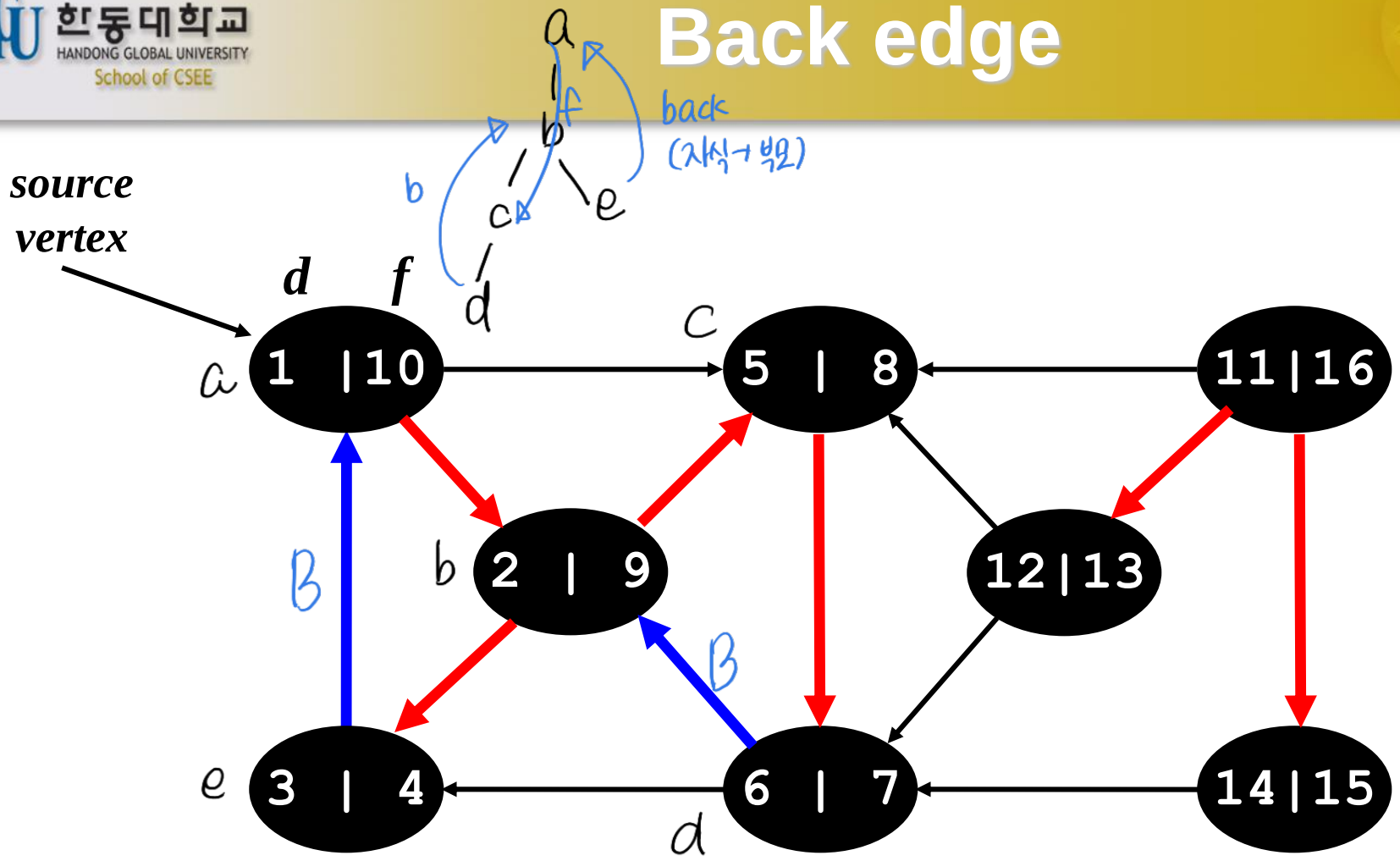
Tree edges (Gray → white 로 가는 edge)

DFS: Kinds of edges

- DFS introduces an important distinction among edges in the original graph:
  - *Tree edge*: encounter new (white) vertex (explored)
  - *Back edge*: from descendent to ancestor (checked)
    - Encounter a gray vertex (gray to gray)



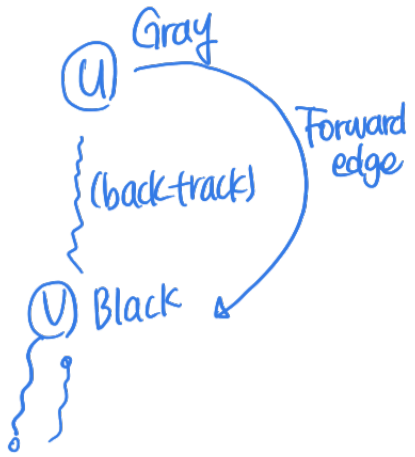
Back edge



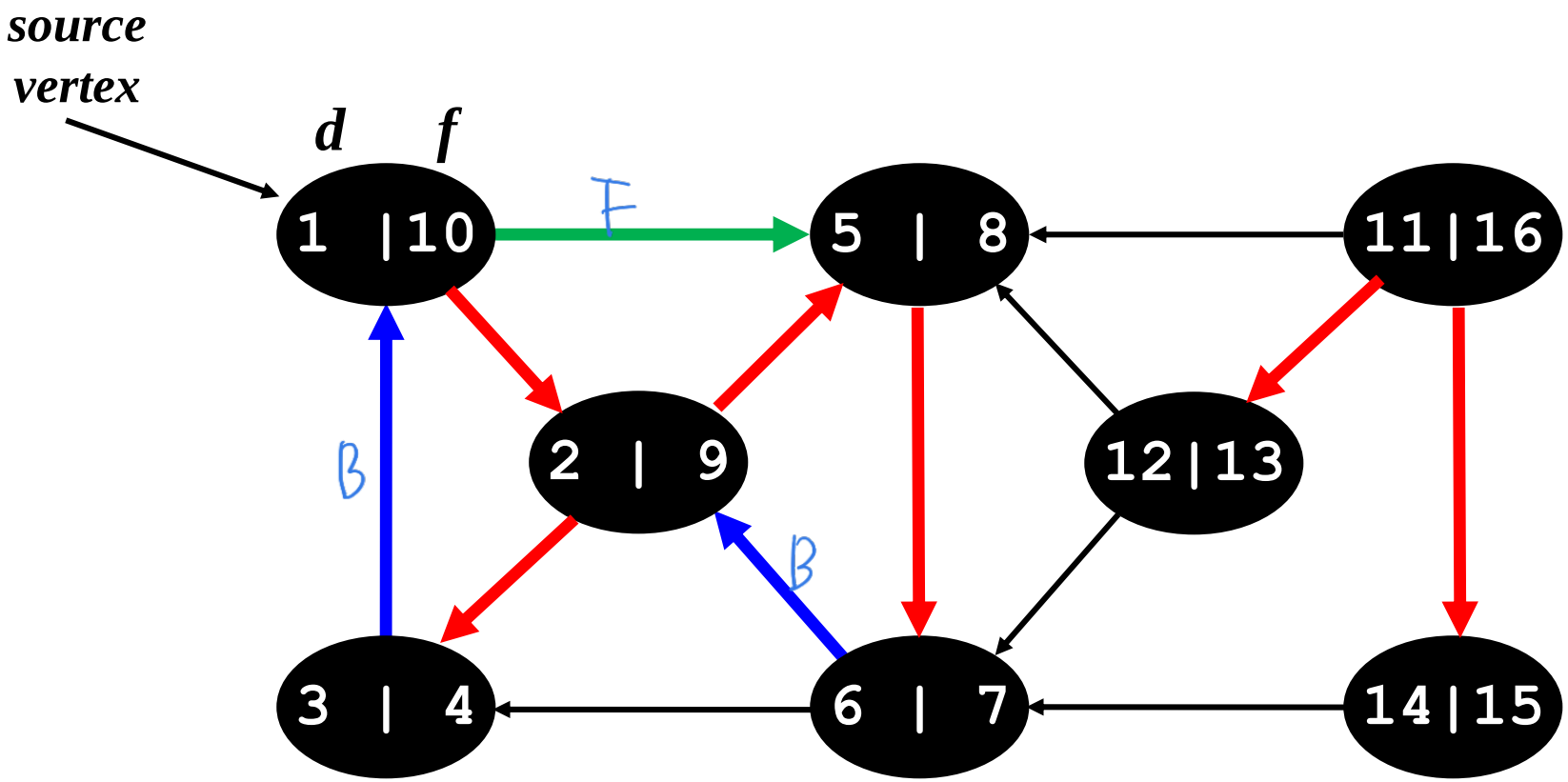
Tree edges *Back edges*

DFS: Kinds of edges

- DFS introduces an important distinction among edges in the original graph:
  - *Tree edge*: encounter new (white) vertex (explored)
  - *Back edge*: from descendent to ancestor (checked)
  - *Forward edge*: from ancestor to descendent (checked)
    - Not a tree edge, though (tree 제외)
    - From gray node to black node



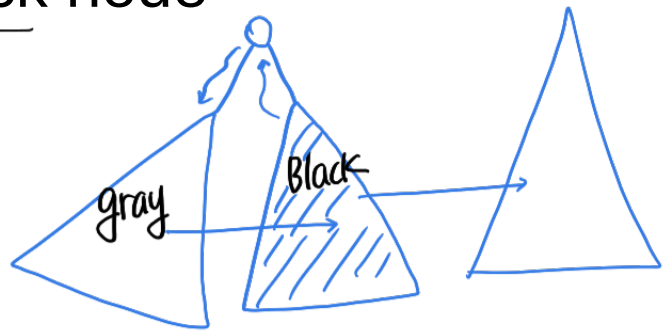
Forward edge



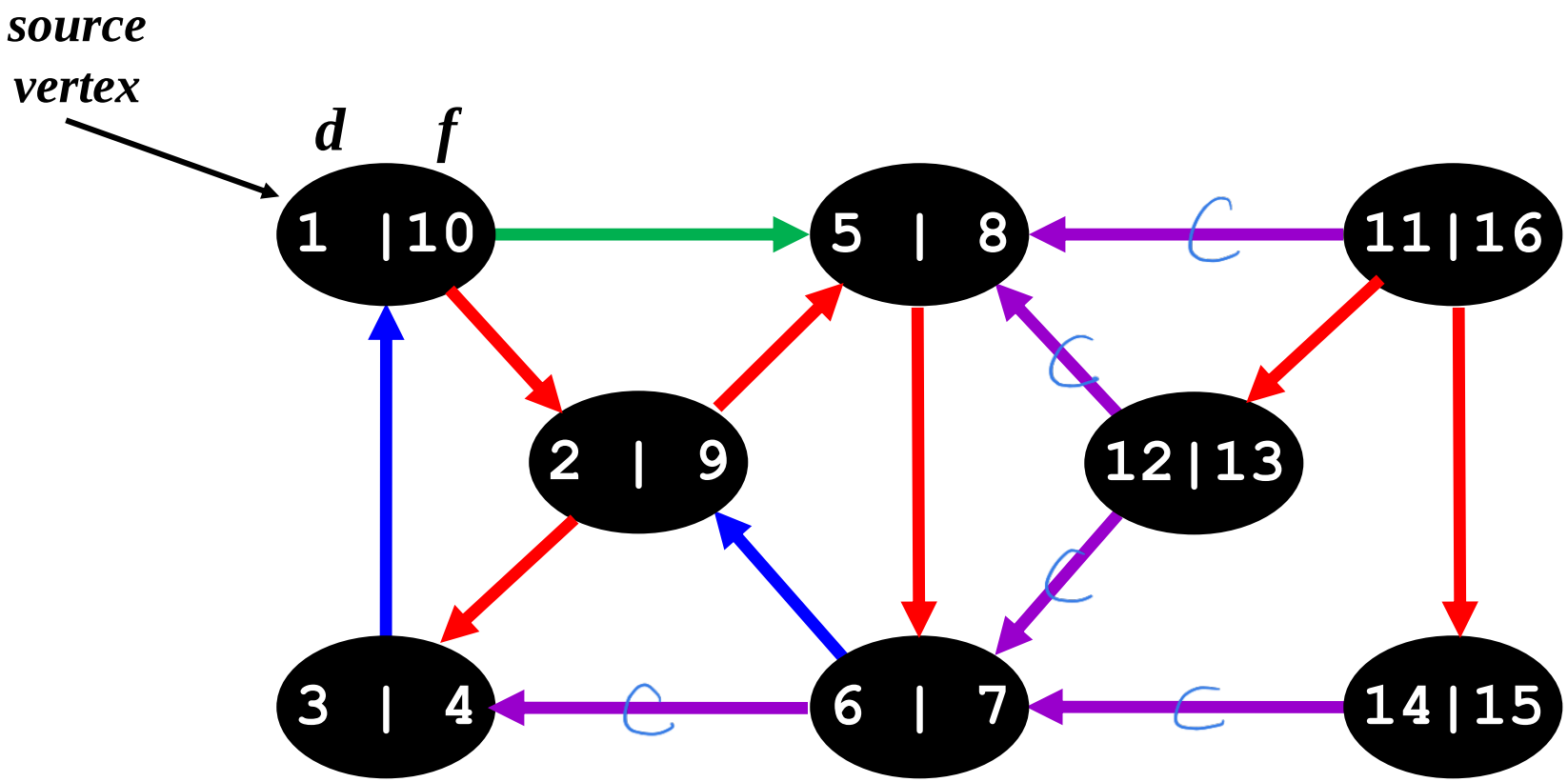
Tree edges *Back edges* *Forward edges*

DFS: Kinds of edges

- DFS introduces an important distinction among edges in the original graph:
  - *Tree edge*: encounter new (white) vertex
  - *Back edge*: from descendent to ancestor
  - *Forward edge*: from ancestor to descendent
  - *Cross edge*: between a tree or subtrees (checked)
    - From a gray node to a black node

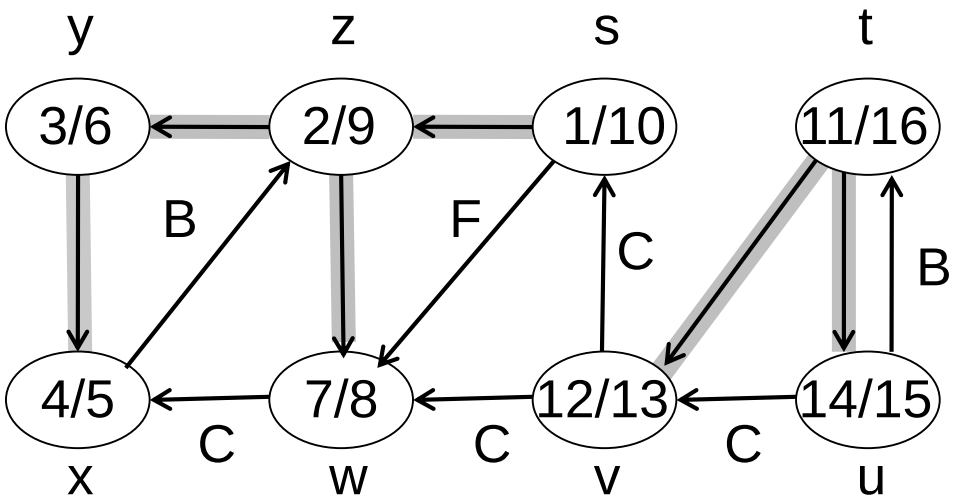


Cross edge

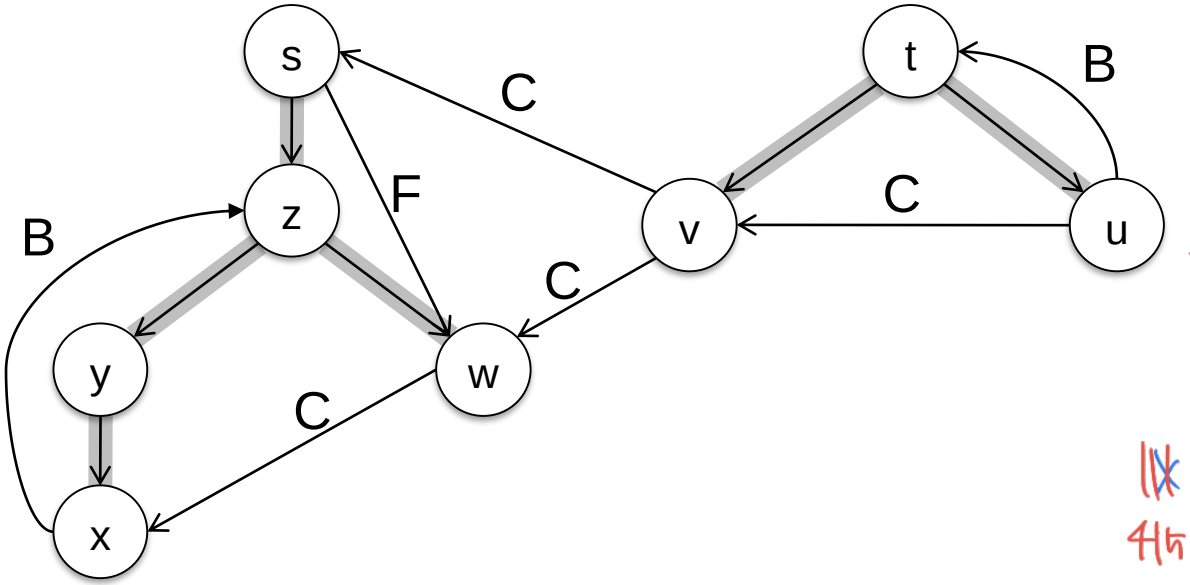


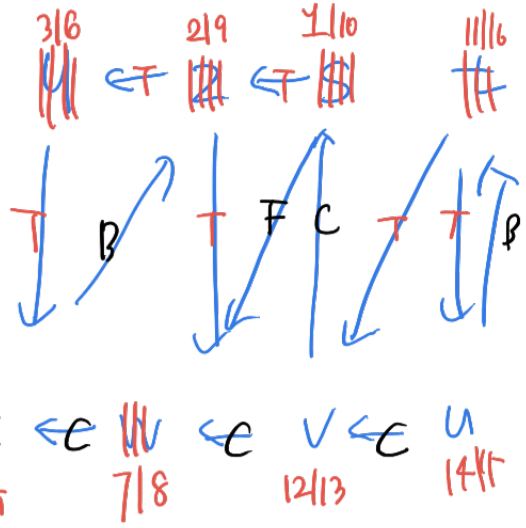
Tree edges *Back edges* *Forward edges* *Cross edges*

DFS Example



B : Back Edge (gray → gray)
F : Forward Edge
C : Cross Edge (gray → black)
Other : Tree Edge (shaded)





DFS: Kinds of edges

- From a gray node to a white node

  ➔ tree edge

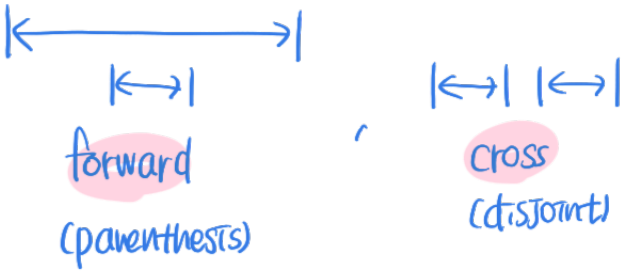
- From a gray node to a gray node

  ➔ back edge

- From a gray node to a black node

  ➔ forward or cross edge

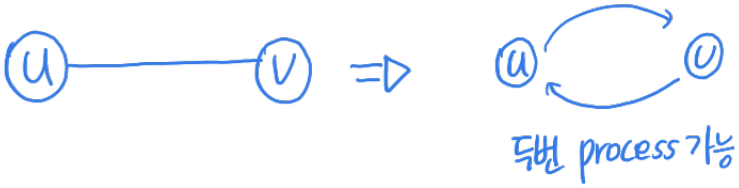
- Then how do you tell the forward edge from cross edge?



DFS on Undirected graph

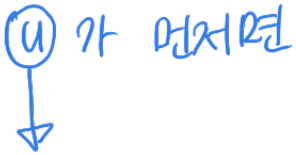
If it doesn't matter if an edge is processed twice

transform it into symmetric digraph, then process edge twice.



Else

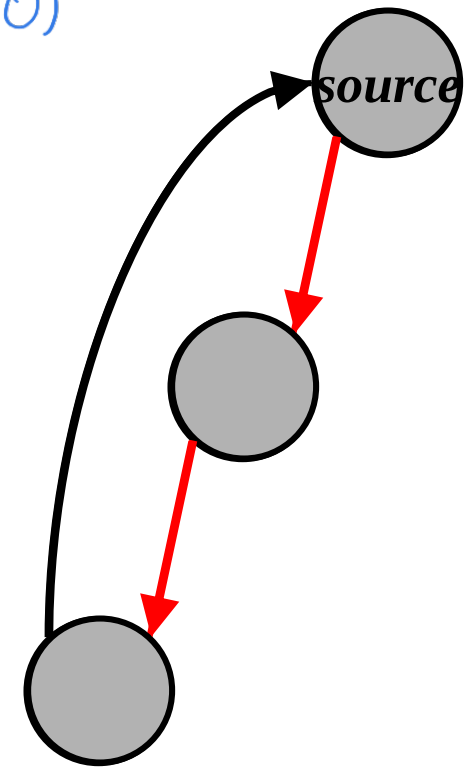
edge should be explored in one direction. The direction is determined when the edge is first encountered.



DFS: Kinds Of Edges

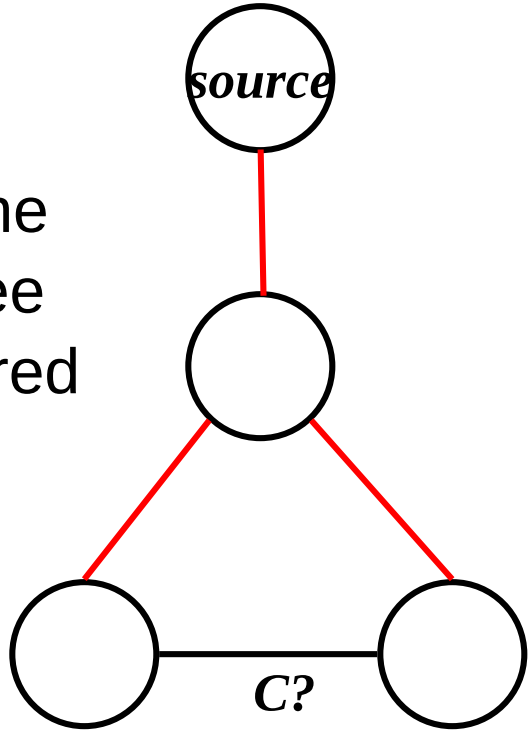
- Theorem 22.10: If *G* is undirected, a DFS produces only tree and back edges (no forward, cross)
- Proof by contradiction:
  - Assume there's a forward edge (back O)
    - But F? edge must actually be a back edge (*why?*)

edge는 V3부터 check 될수 X



DFS: Kinds Of Edges

- Theorem 22.10: If G is undirected, a DFS produces only tree and back edges
- Proof by contradiction:
  - Assume there's a cross edge.
    - But C? edge cannot be cross:
    - It should be explored from one of the vertices it connects, becoming a tree vertex, before other vertex is explored
    - So in fact the picture is wrong. The lower edge cannot be cross edge, but tree edge.



DFS vs. BFS

- DFS : Stack (≒ recursion)
  LIFO
  BFS : Queue
  FIFO
- DFS : Two processing opportunity
  - Once when it is discovered : Preorder (gray로 바꾸고 스택에 들어 감)
  - Once when it is marked finished : Postorder (pop 될 때 이제 해서 작업)
  - Postorder : We have information about adjacent vertices. (근처 애들에 대한 정보를 다 가져올 수 있음... 작업)

  ex) Counting the number of leaves in binary tree

  BFS : One processing opportunity

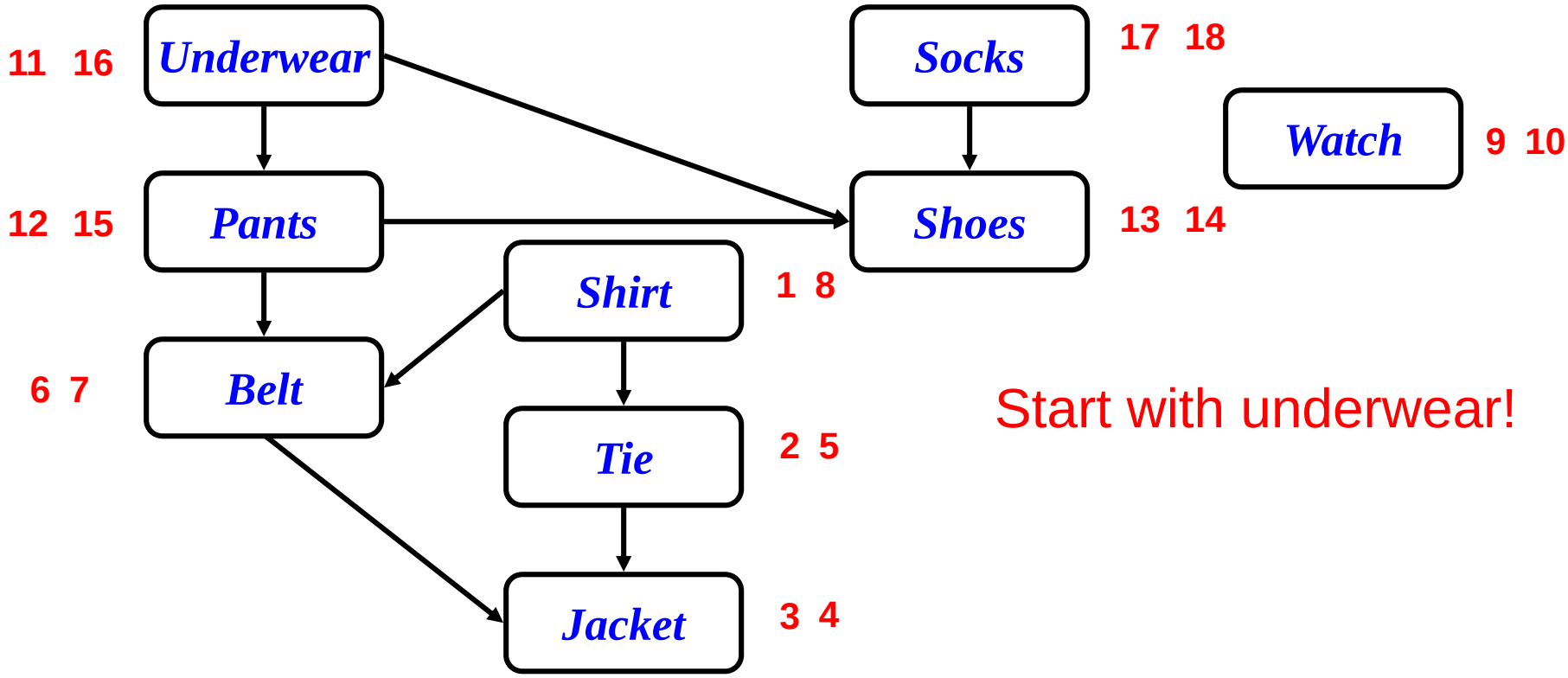
Applications of DFS

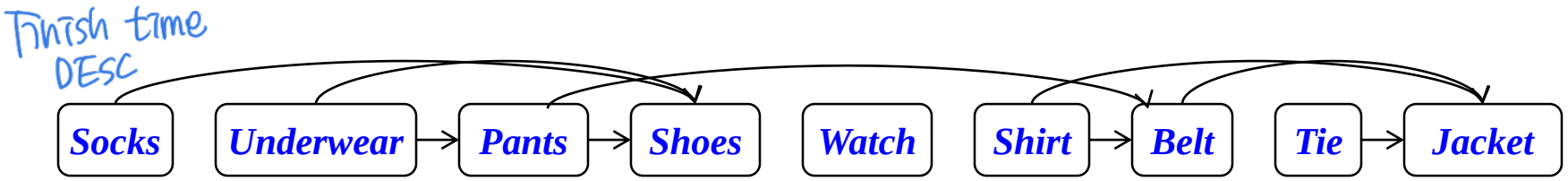
1. Topological Sort
2. Finding Connected Component (Undirected)
3. Finding Strongly Connected Component (Directed)
4. Critical Path Analysis
5. Biconnected Component (Articulation point) Problem
6. Etc…

1) Topological Sort

- *Topological sort* of a DAG:
  - Linear ordering (of all vertices in graph G) such that vertex u comes before vertex v if edge $(u, v) \in G$
- Real-world example: getting dressed

Getting Dressed Example





Topological Sort Algorithm

Topological-Sort()

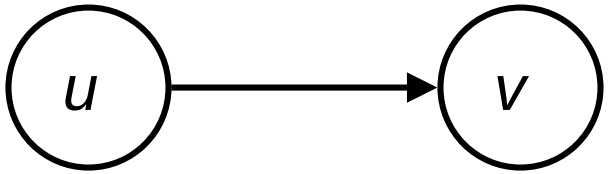
{

1. Run DFS to compute finishing times $f[v]$ for all $v \in V$
2. As each vertex is finished, insert it into the front of a linked list. (Or output vertices in order of decreasing finish time.)

}

- Time: $\Theta(V+E)$
- Correctness: Want to prove that

  $(u,v) \in G \Rightarrow f[u] > f[v]$



Correctness of Topological Sort

- Claim: $(u,v) \in G \Rightarrow f[u] > f[v]$

  - When (u,v) is explored, u is gray
    - v = gray $\Rightarrow$ (u,v) is back edge. Contradiction (*Why?*)

      cycle

      We are dealing with DAG (no cycle) here.
    - v = white $\Rightarrow$ v becomes descendent of $u \Rightarrow f[v] < f[u]$

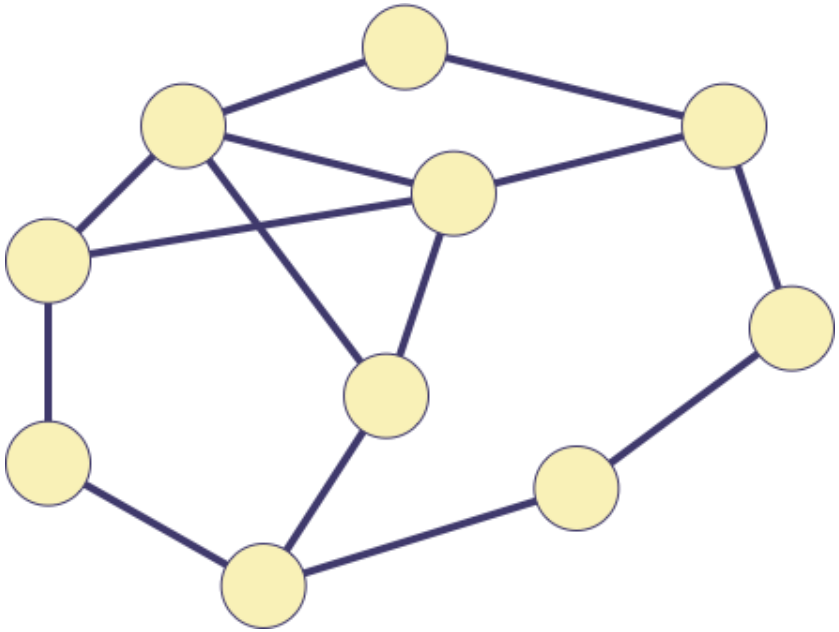
      u ⟶ v d(u) → d(v) → f(v) → f(u)

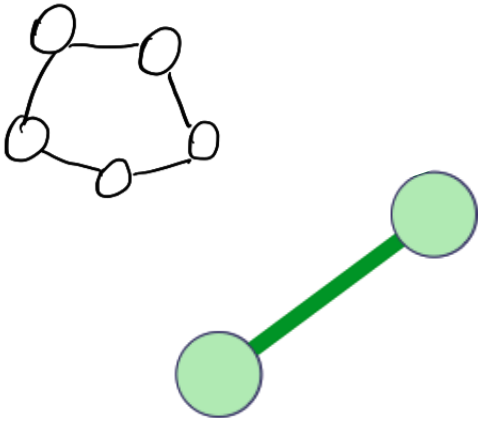
      (since must finish v before backtracking and finishing u.)
    - v = black $\Rightarrow$ v already finished $\Rightarrow f[v] < f[u]$

Connectivity : Undirected graph

A graph *G* is ***connected*** if there is a path between any two vertices in *G*.

The ***connected components*** of a graph are its maximal connected subgraphs.

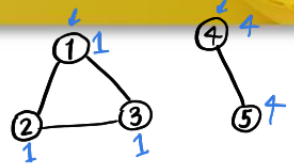




Not a connected component

2) Connected Component

undirected, DFS-VISIT 때 label을 전달



- Convert undirected graph into symmetric graph.
- Function DFS
  - **Find undiscovered vertices and initiate a DFS-VISIT at each undiscovered vertex.**
  - **Pass id – *label*(*v*) – of undiscovered vertex *v* to DFS-VISIT.**
- Function DFS-VISIT
  - **Determine the label of current vertex as *label*(*v*) and pass it to its white adjacent vertices.**
  - **Make a separate linked list for each component.**

3) Strongly Connected Component

directed

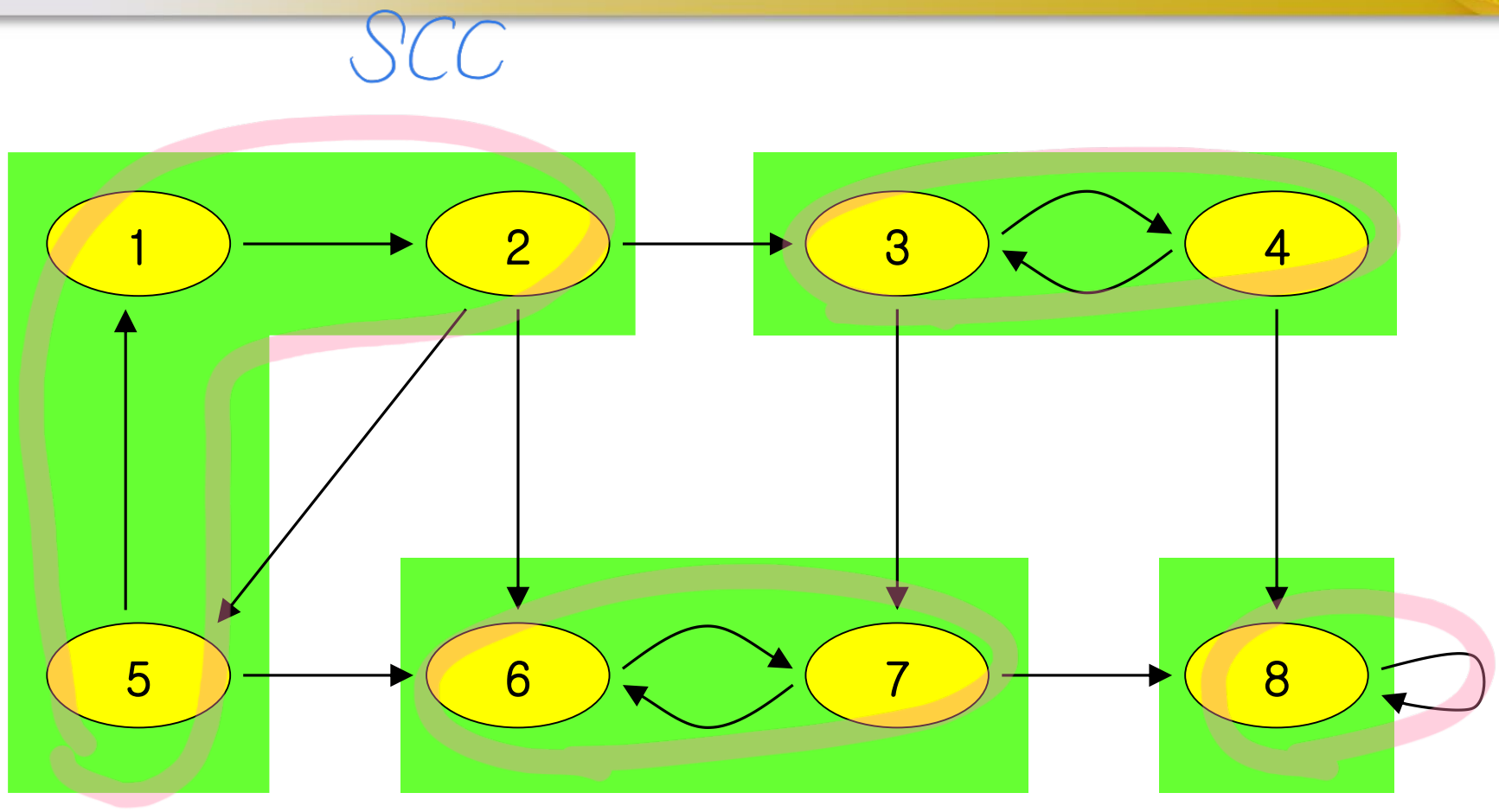
- Definition:

  A strongly connected component of a directed graph G=(V,E) is a *maximal* set of vertices $C \subseteq V$ such that that for every pair of vertices u and v in C, $u \rightarrow v$ and $v \rightarrow u$.

  Informally, a maximal sub-graph in which every vertex is *reachable* from every other vertex.

(모든 vertex : 다른 vertex 다 갈 수 있는) path 존재.

Example



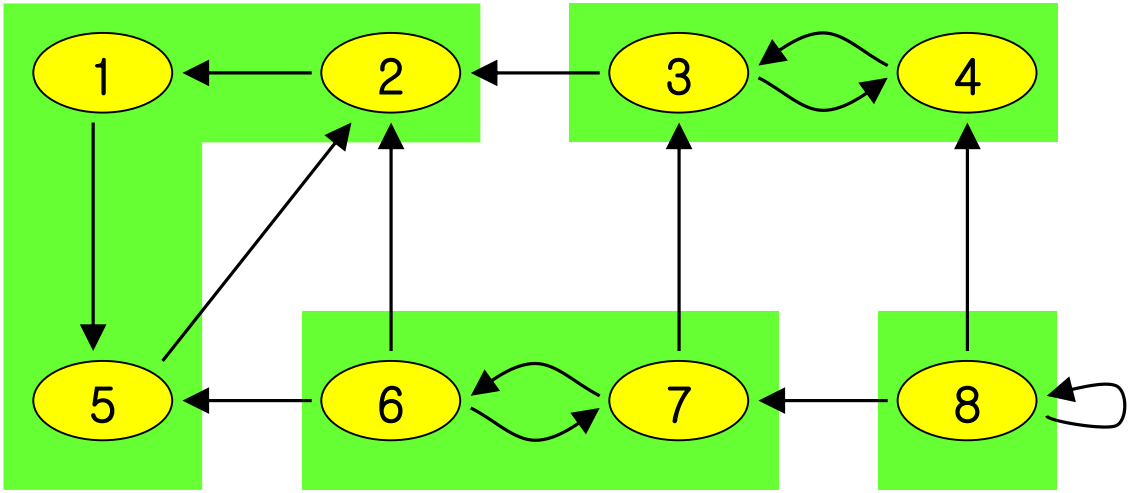
Green areas are the Strongly Connected Components.

Property

- G and G^T have exactly the same strongly connected components.

(transpose)

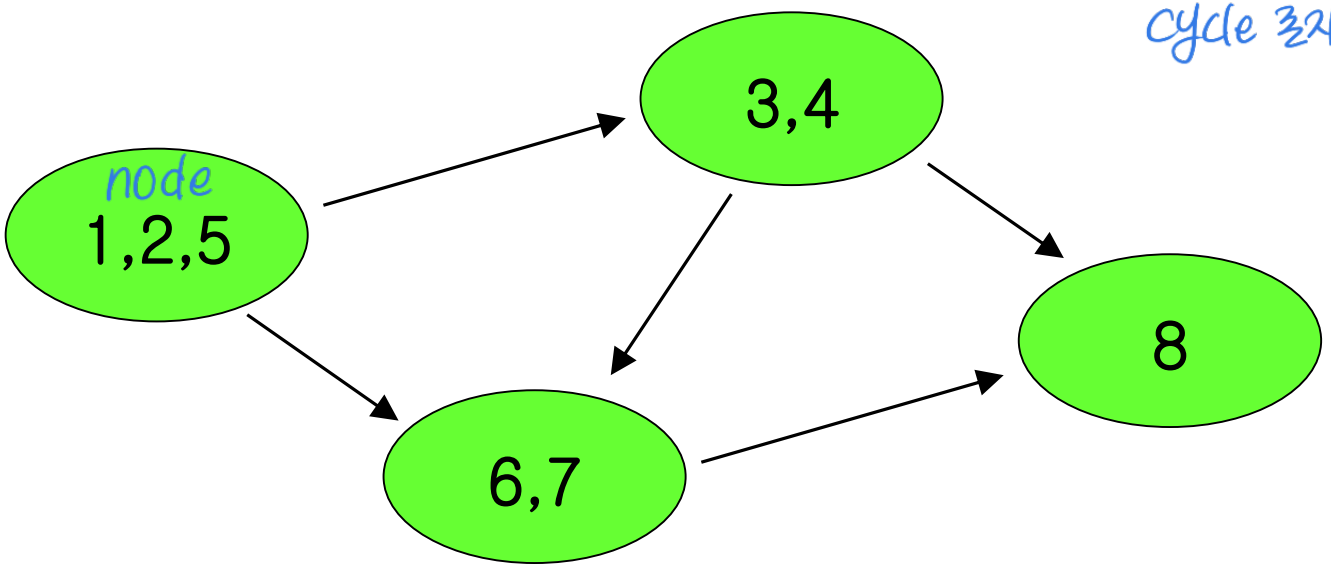
@→ⓑ ⇒ @←ⓑ transpose



Property

- The component graph G^{SCC} is directed acyclic graph.

SCC 내에서는 cycle 존재



Let's think…

How can we get
the Strongly Connected Components
of directed graph of G.

Don't be surprised!
The algorithm is very simple!

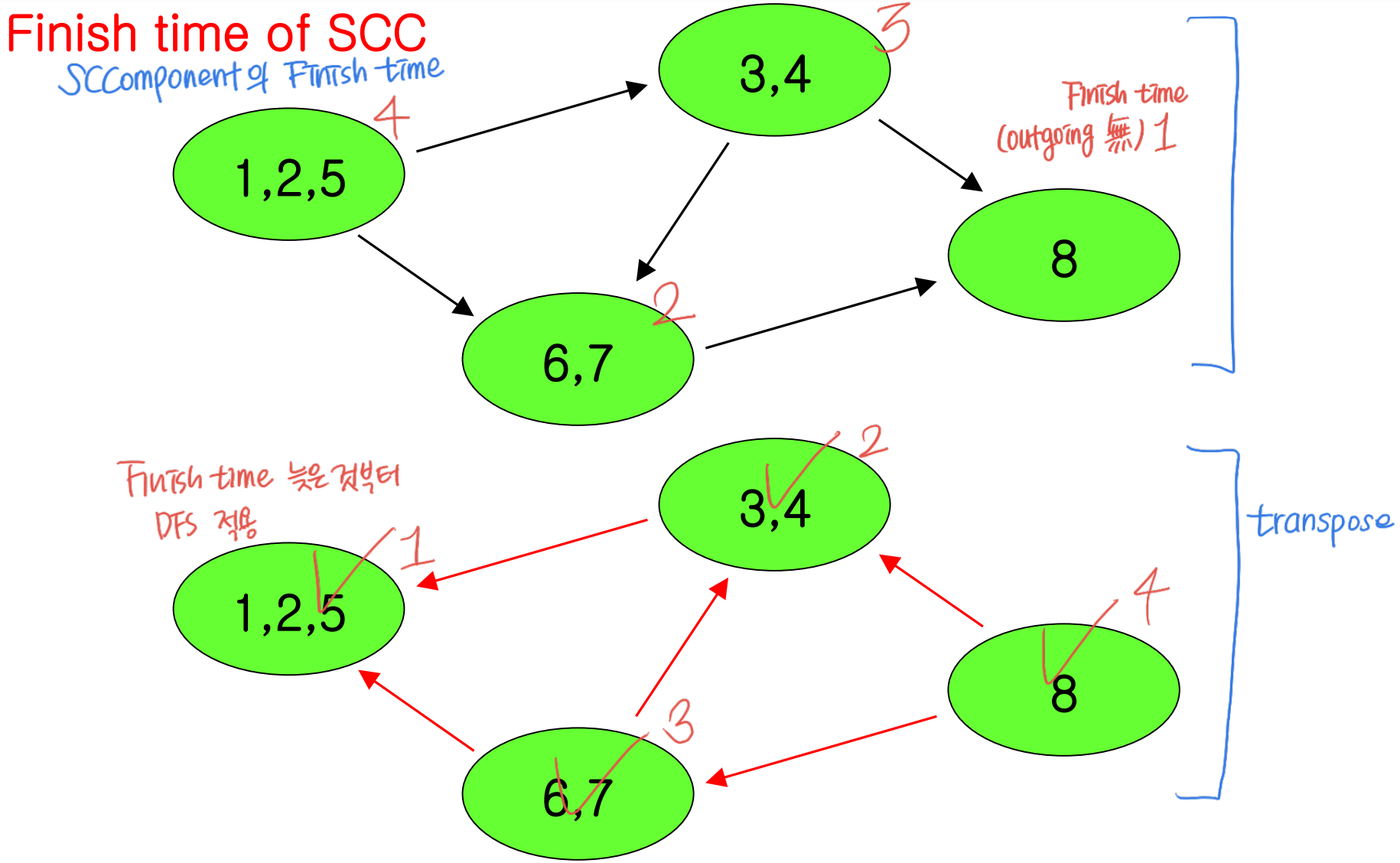
Algorithm

STRONGY-CONNECTED-COMPONENTS(G)

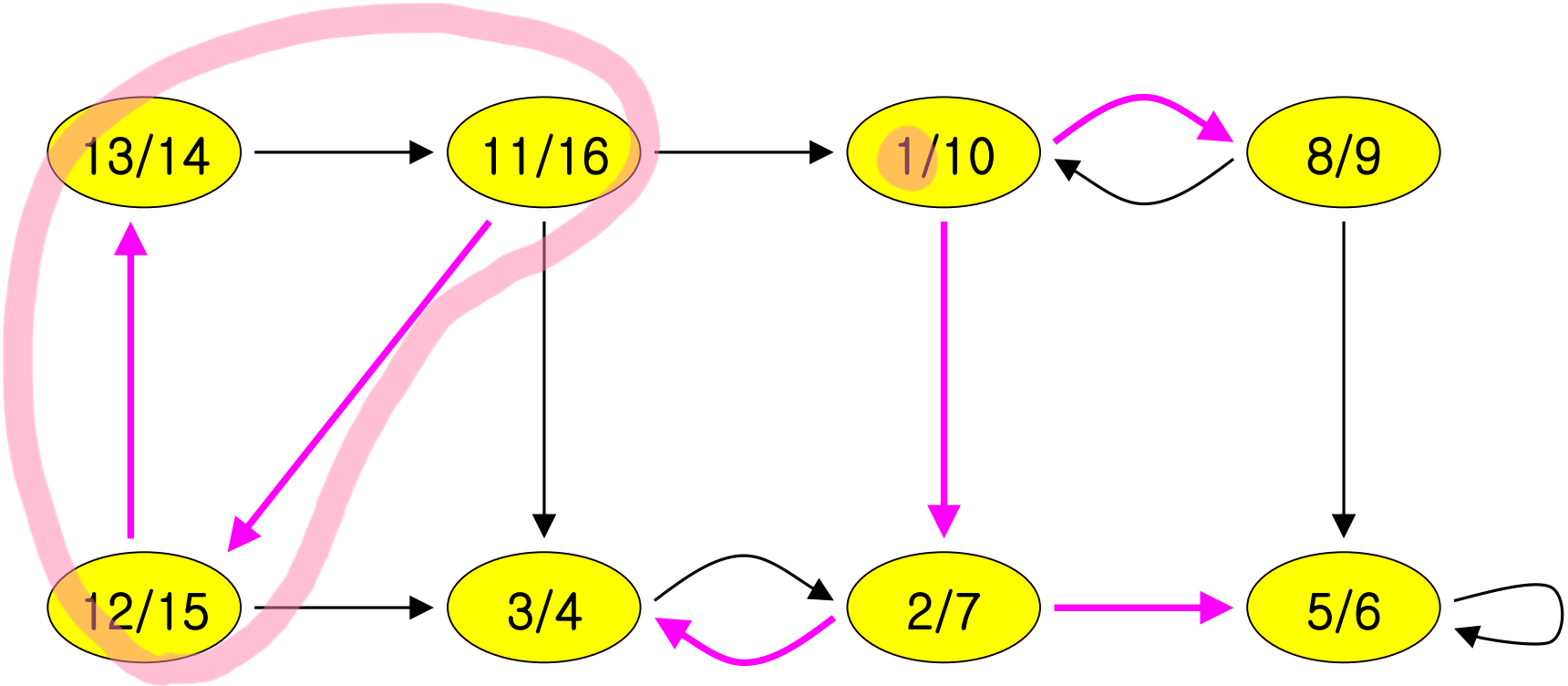
1. call DFS(G) to compute finishing times $f[u]$ for each vertex u $\Theta(V+E)$
2. compute G^T $\Theta(V+E)$ .. (adjacency list 가정)
3. Call DFS(G^T), but in the main loop of DFS, consider the vertices in order of decreasing $f[u]$ (as computed in line 1) $\Theta(V+E)$
   (큰 node 부터)
4. Output the vertices of each tree in the depth-first forest formed in line 3 as a separate strongly connected component

∴ **The running time of this algorithm is $\Theta(V+E)$**

G^{SCC} vs. transpose of G^{SCC}

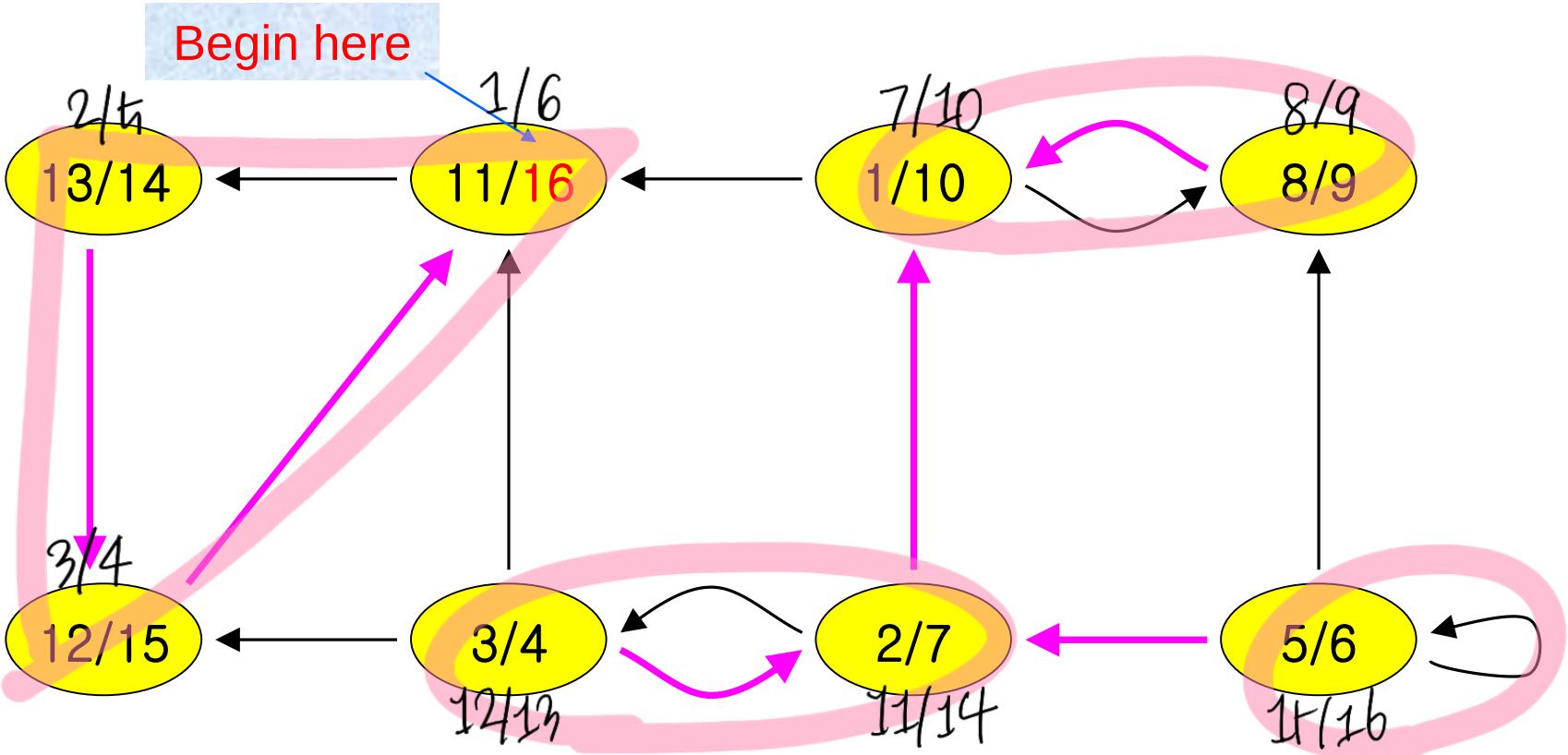


Algorithm : step1



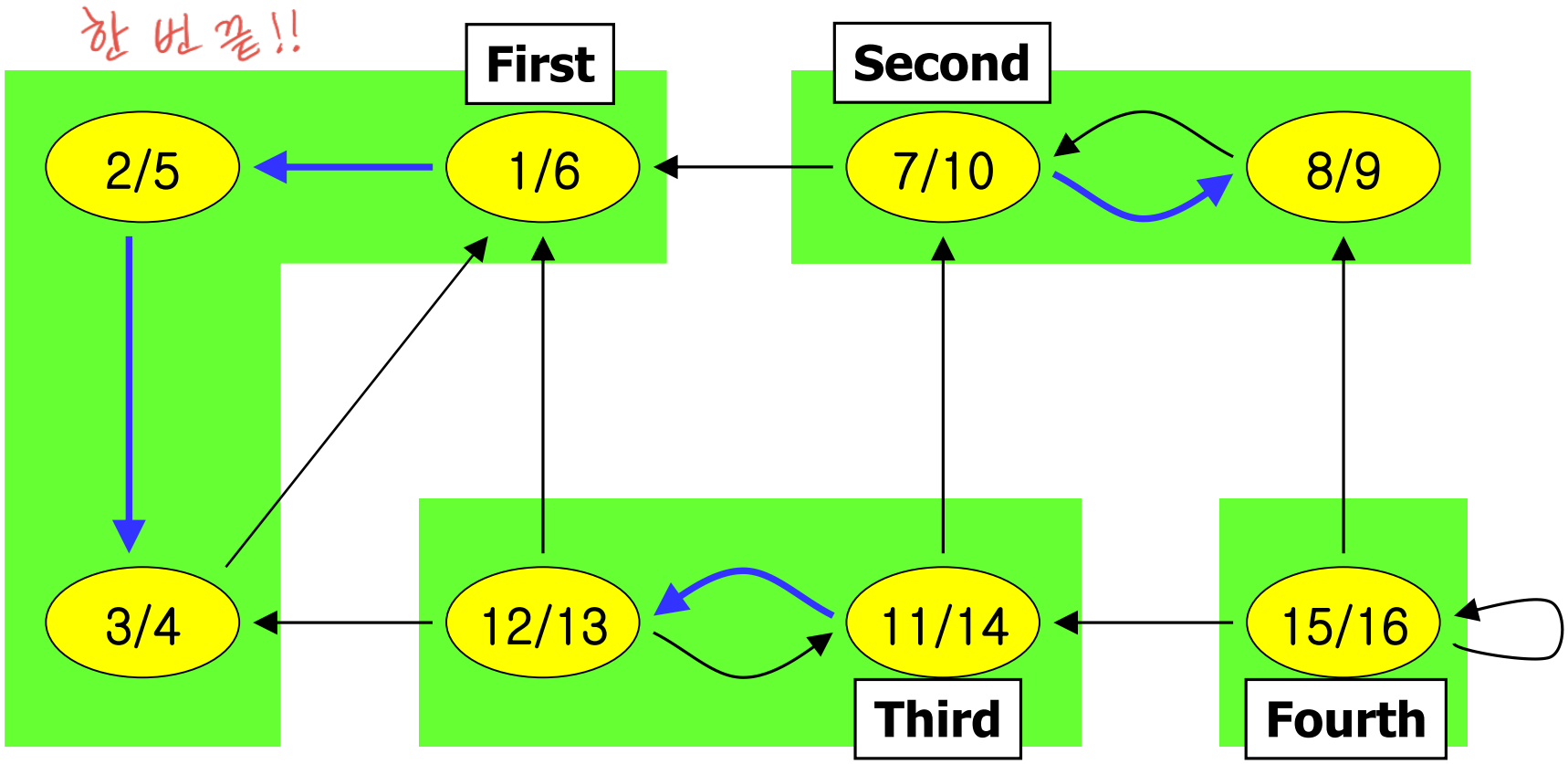
Call DFS(*G*)

Algorithm : step2



Compute transpose of *G* !!

Algorithm : step3

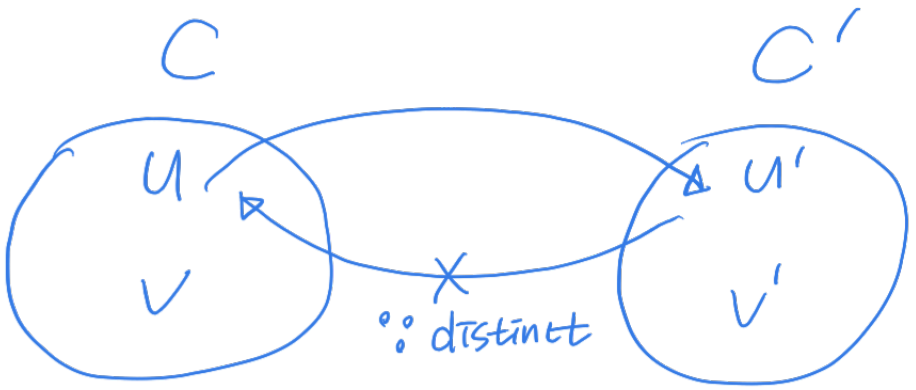


Compute DFS(G^T), but in order of decreasing f [u]

Lemma

- Lemma 22.13

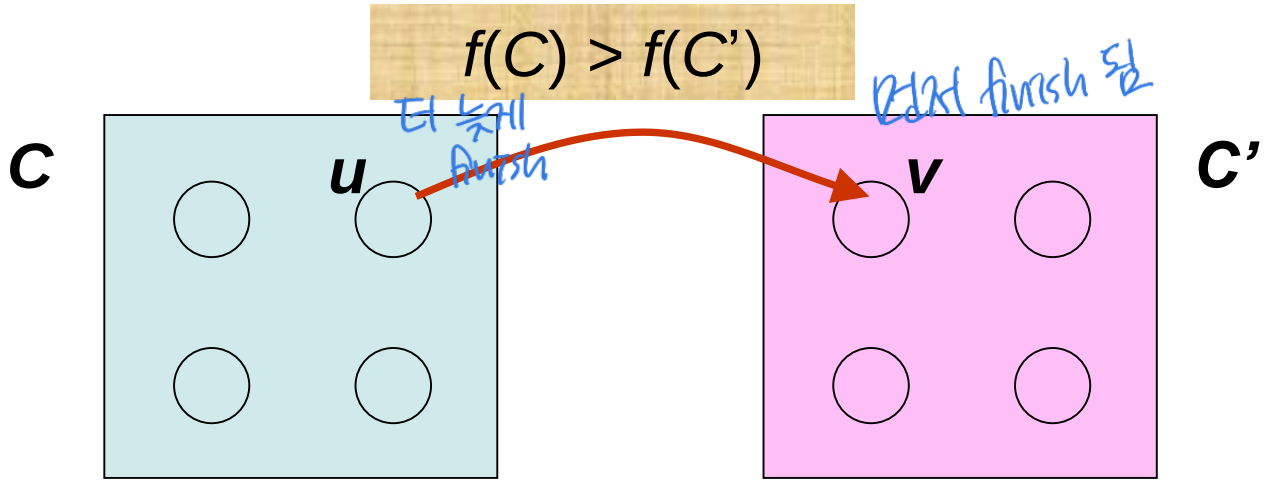
  Let C and C' be distinct strongly connected components in directed graph $G = (V, E)$, let $u, v \in C$, let $u', v' \in C'$, and suppose that there is a path $u \rightarrow u'$ in G. Then there cannot also be a path $v' \rightarrow v$ in G.



Lemma and Corollary

- Lemma 22.14

  Let C and C' be distinct strongly connected components in directed graph $G = (V, E)$. Suppose that there is an edge $(u, v) \in E$, where $u \in C$ and $v \in C'$. Then $f(C) > f(C')$.



- Note, $f(U) = \max_{u \in U}\{f[u]\}$. That is $f(U)$ is latest finishing time of any vertex in U.

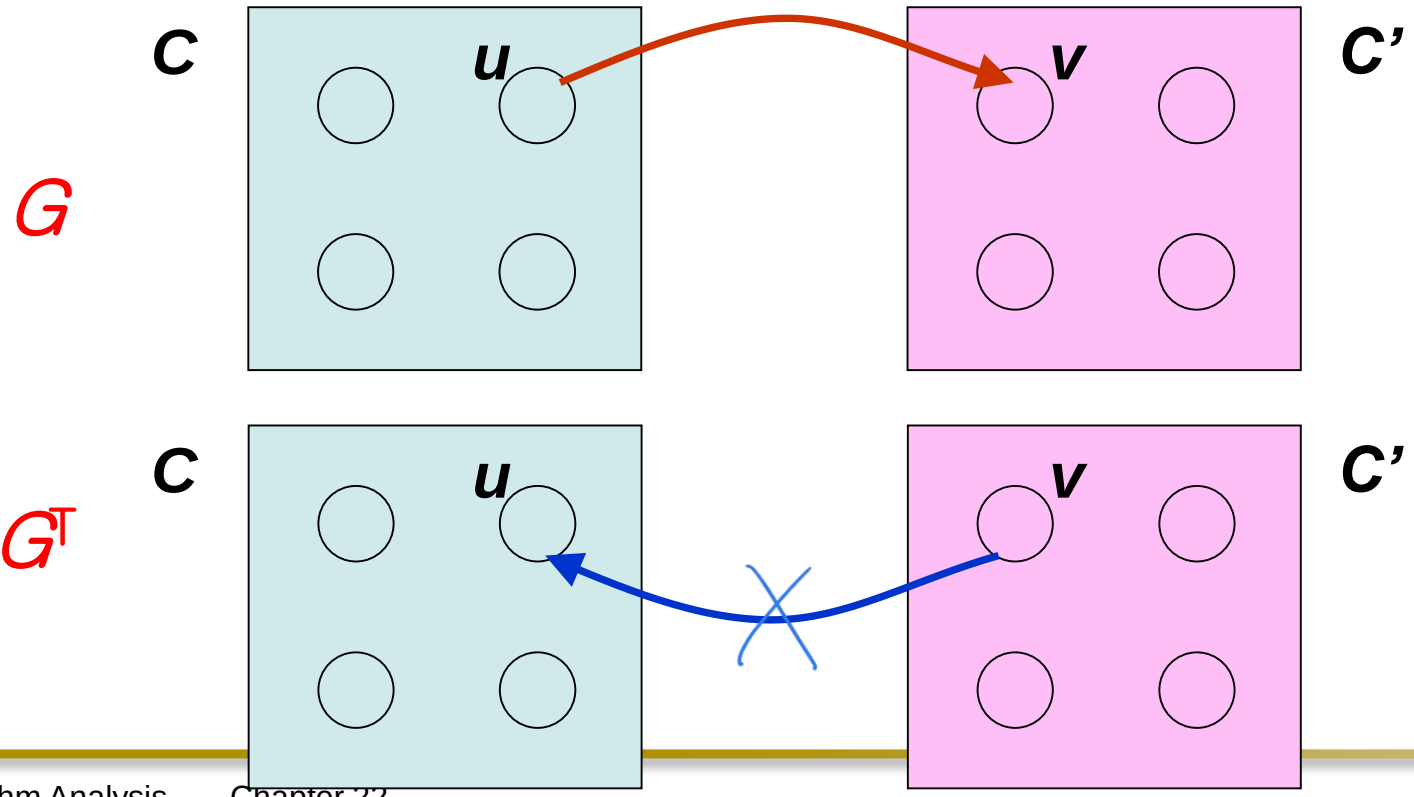
Lemma and Corollary

- Corollary 22.15

  Let C and C' be distinct SCC's in $G = (V, E)$. Suppose there is an edge $(v,u) \in E^T$, where $u \in C$ and $v \in C'$. Then $f(C) > f(C')$.

(transpose)

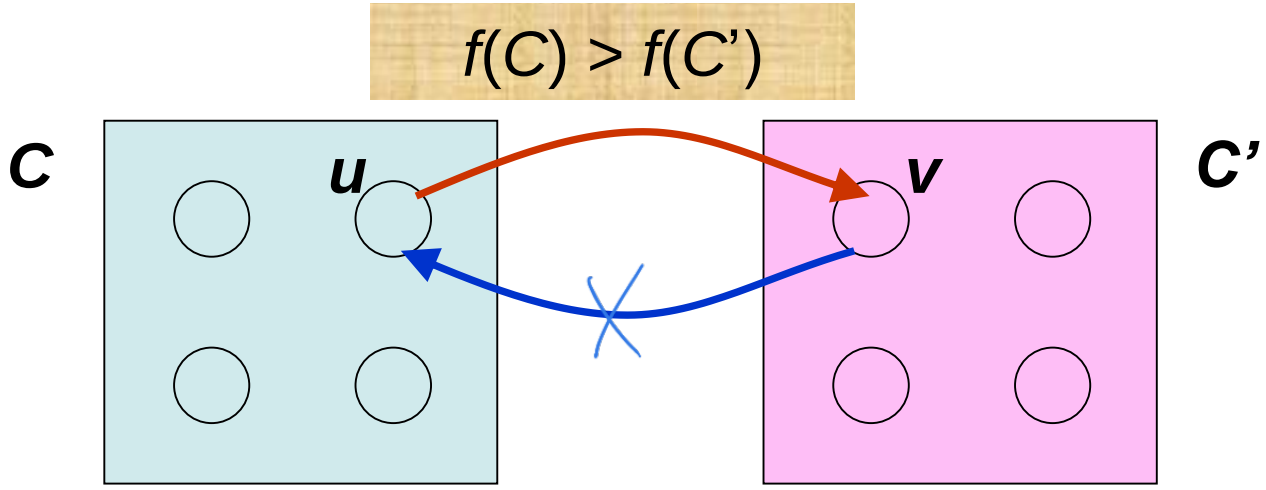
$f(C) > f(C')$



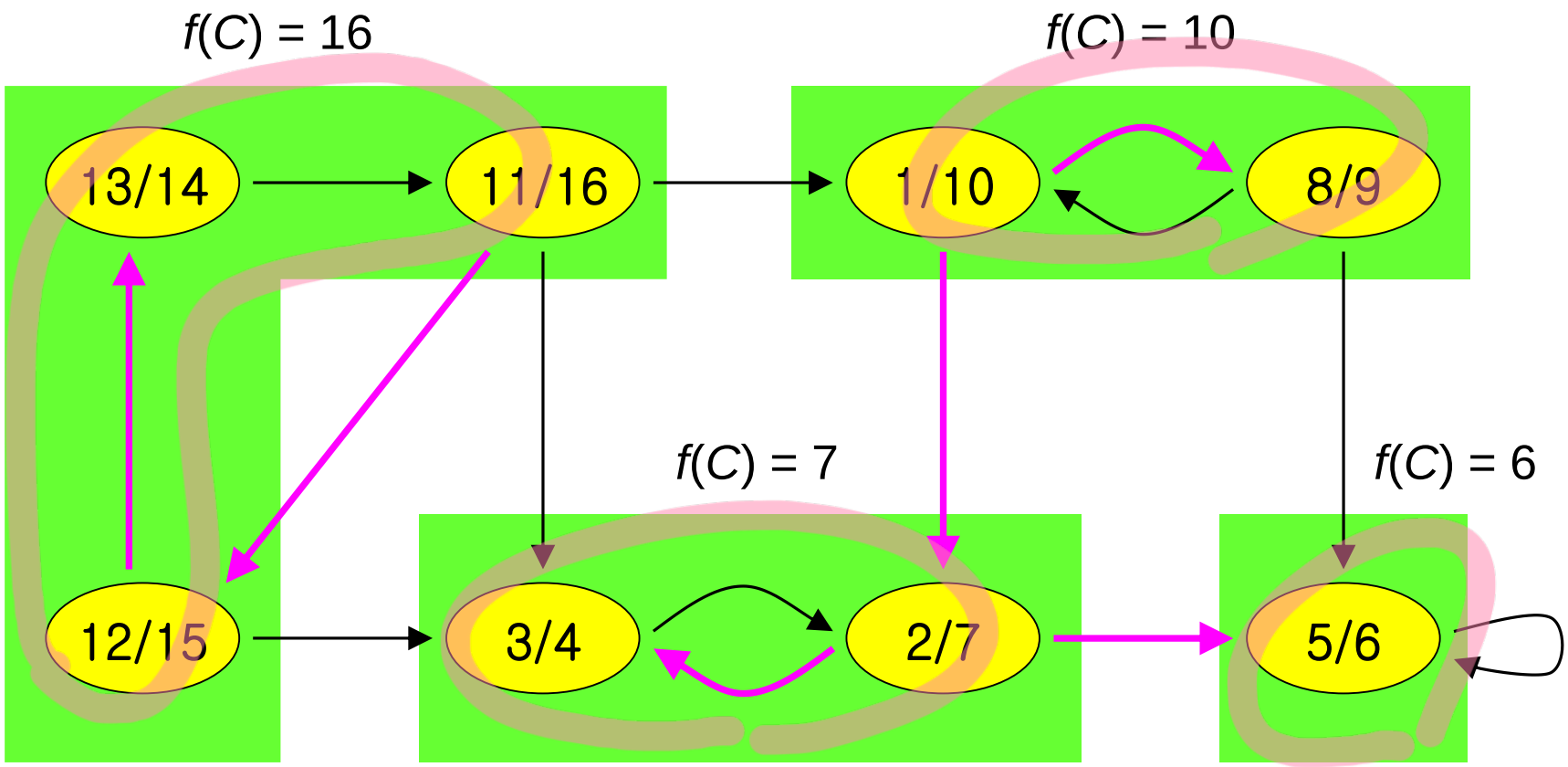
Lemma and Corollary

- Corollary

  Let C and C' be distinct SCC's in $G = (V, E)$. Suppose that $f(C) > f(C')$. Then there cannot be an edge from C to C' in G^T.

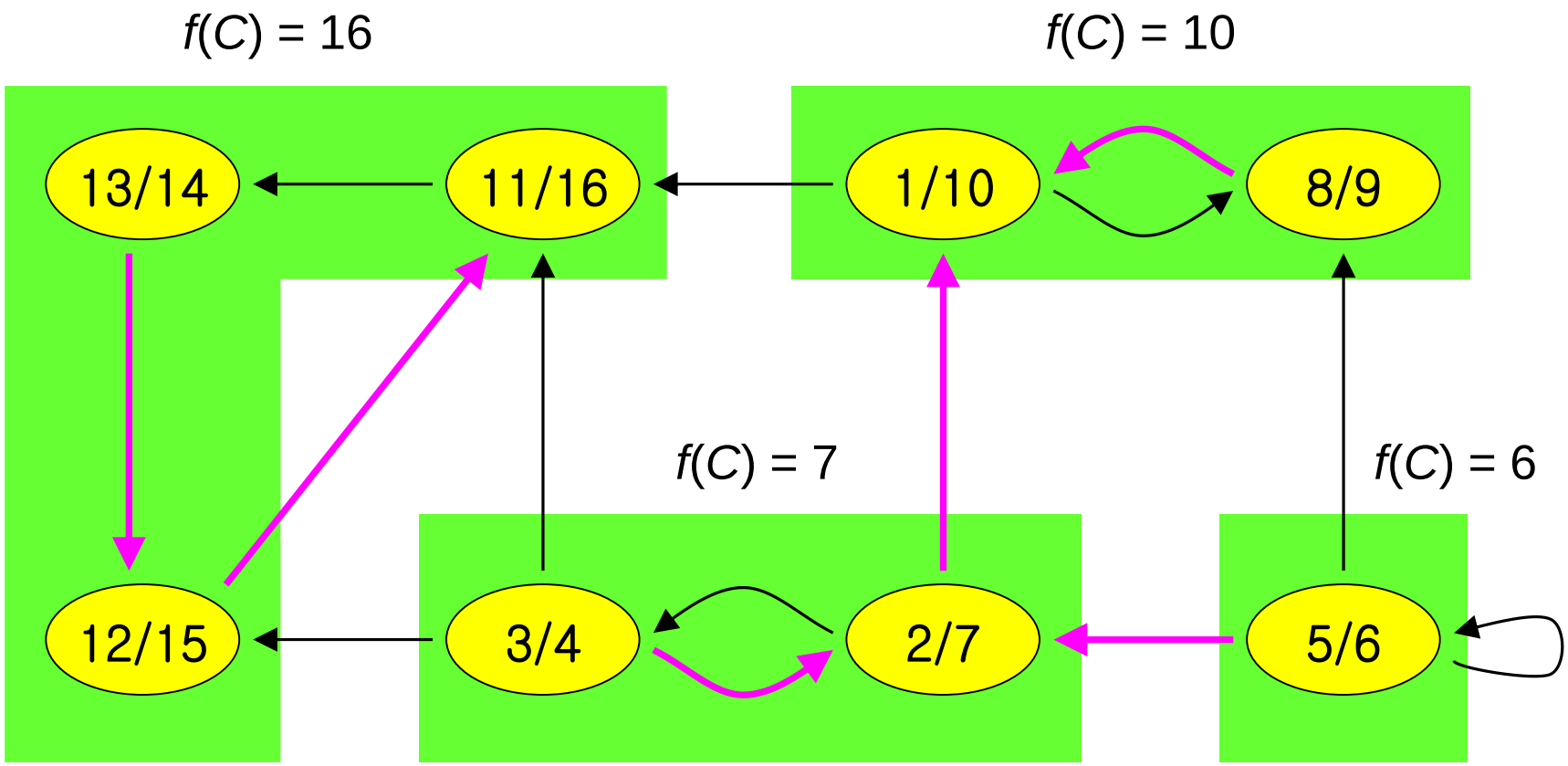


Recall Algorithm



For graph *G*

Recall Algorithm



Transpose of *G* !!

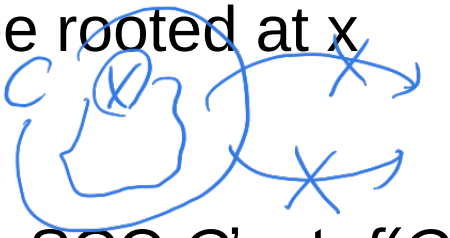
Theorem

- Theorem 22.16

  STRONGLY-CONNECTED-COMPONENTS(*G*) correctly computes the strongly connected components of a directed graph *G*.

Proof

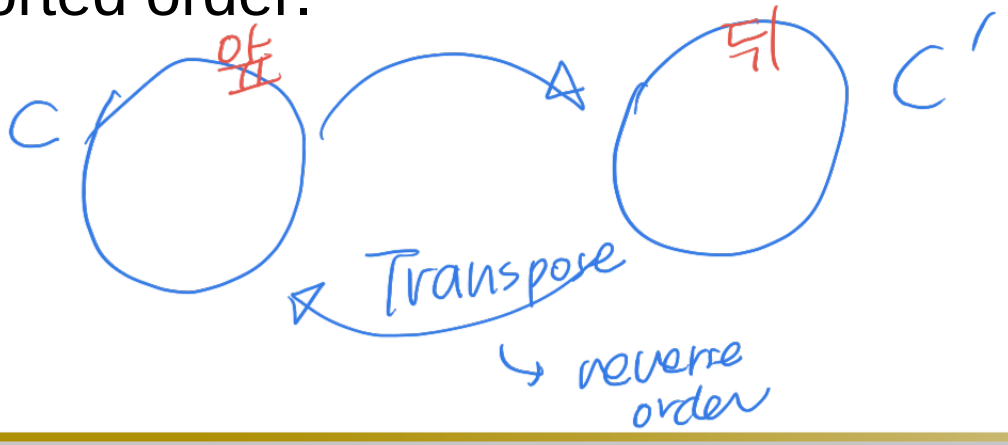
- When we do 2nd DFS on G^T, start with SCC C s.t. $f(C)$ is maximum. The second DFS from some $x \in C$, and it visits all vertices in C. Corollary says that since $f(C) > f(C')$ for all $C' \neq C$, there are no edges from C to C' in G^T. Therefore DFS will visit only vertices in C. (DFS tree rooted at x contains exactly the vertices of C.)



- The next root chosen in the 2nd DFS is in SCC C' s.t. $f(C')$ is maximum over all SCC's other than C. DFS visits all vertices in C', but the only edges out of C' go to C, which we've already visited.

Proof

- Each time we choose a root for the 2nd DFS, it can reach only
  - vertices in its SCC – get tree edges to these,
  - vertices in SCC's already visited in second DFS – get no tree edges to these.
- We are visiting vertices of $(G^T)^{scc}$ in reverse of topologically sorted order.



4) Critical Paths

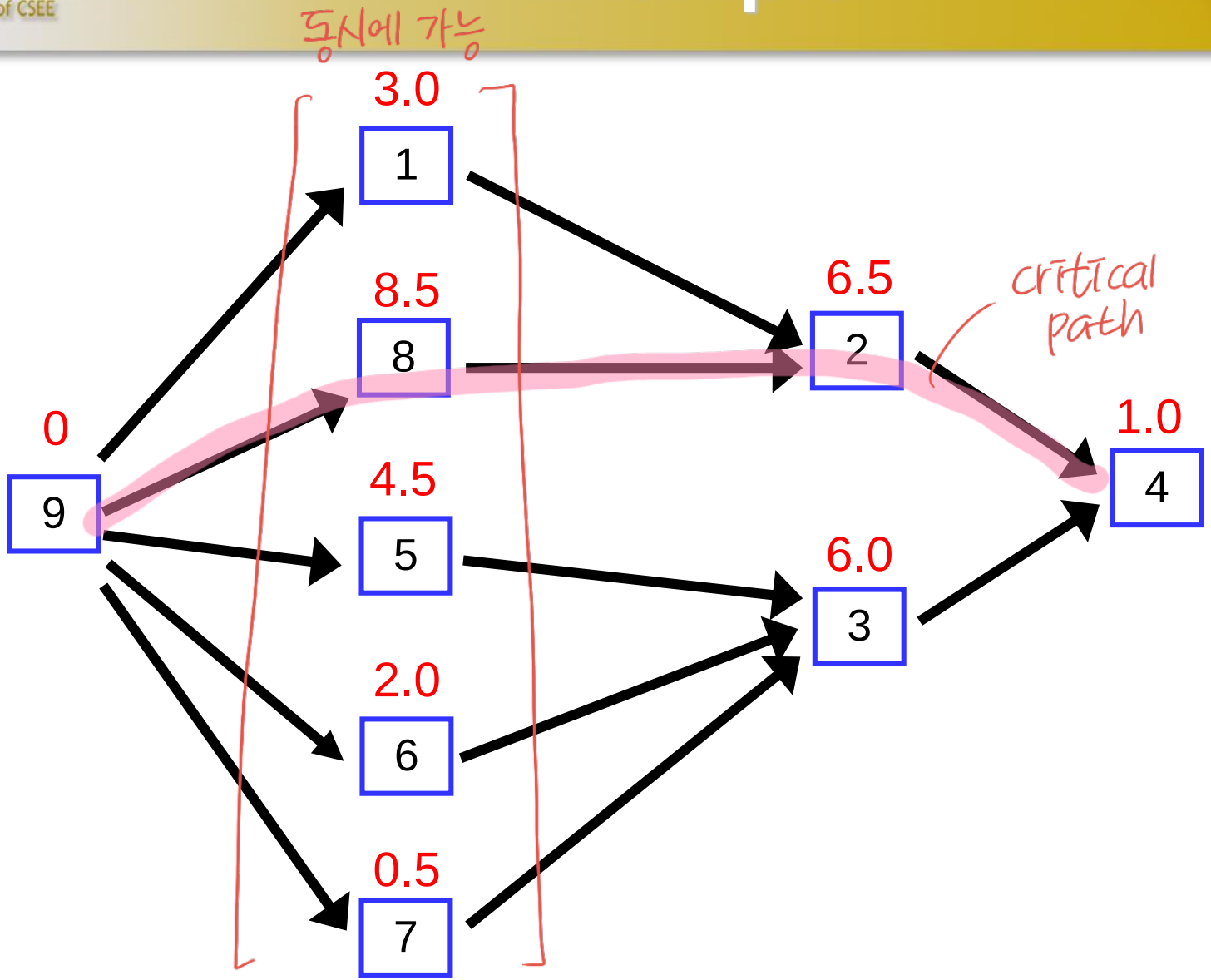
- Say a DAG represents a dependency graph for a series of tasks
  - Each task has a duration
  - We can work on more than one task at a time
- Note there could be several paths from start node to end node
  - Each requires a certain amount of total time
  - One is the longest: the critical path

Example

- Tasks and their duration

1. choose clothes : 3.0
2. dress : 6.5
3. eat breakfast : 6.0
4. leave : 1.0
5. make coffee : 4.5
6. make toast : 2.0
7. pour juice : 0.5
8. shower : 8.5
9. wake up : 0

Example



Finding Critical Paths

- Guess what? We modify DFS (again)
  - Time-complexity: $\Theta(V+E)$
- We record
  - each node's "earliest start time" (*est*)
  - each node's "earliest finish time" (*eft*)
    - *eft* = *est* + *duration*
- How do we calculate *est*?
  - Depends on nodes dependencies!
  - If no dependencies, start right away: *est*=0
  - If depending on one other task, then *est* is that node's *eft*.
  - If depending on more than 1 task, then *est* = max. of *eft* of dependencies.

Modified DAG

- Add a special task to the project, called *done*, with duration 0; it can be task number *n*+1.
- Every regular task that is not a dependency of any task (i.e., potential final task) is made a dependency of *done*.
- The project DAG has a weighted edge *vw* whenever *v* depends on *w* (reverse the direction), and the weight of this edge is the duration of *w*.

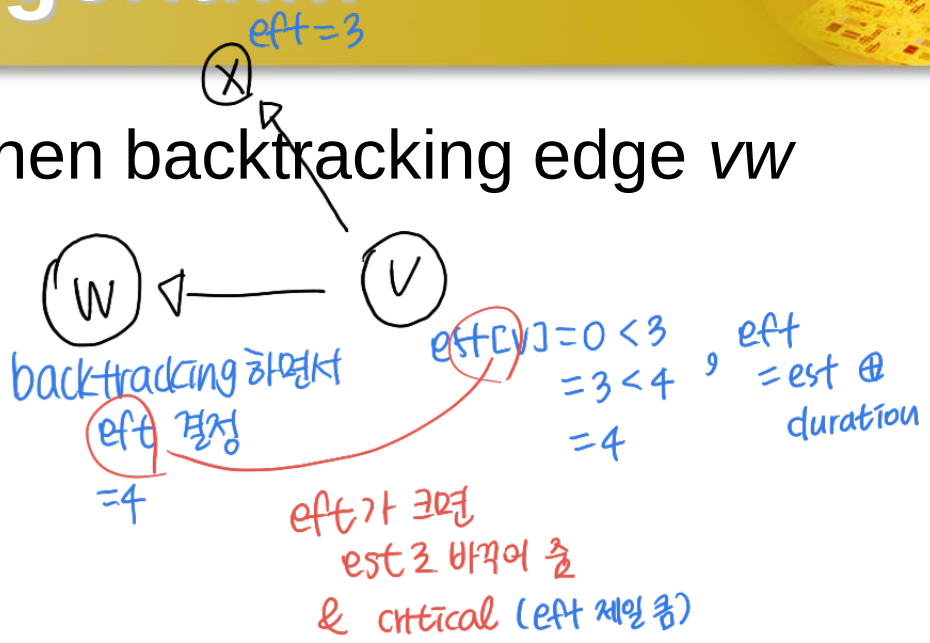
Algorithm

- DFS on project DAG. When backtracking edge *vw*

  if *eft*[*w*] > *est*[*v*]

  est[*v*] = *eft*[*w*]

  Crit_Dep[*v*] = *w*

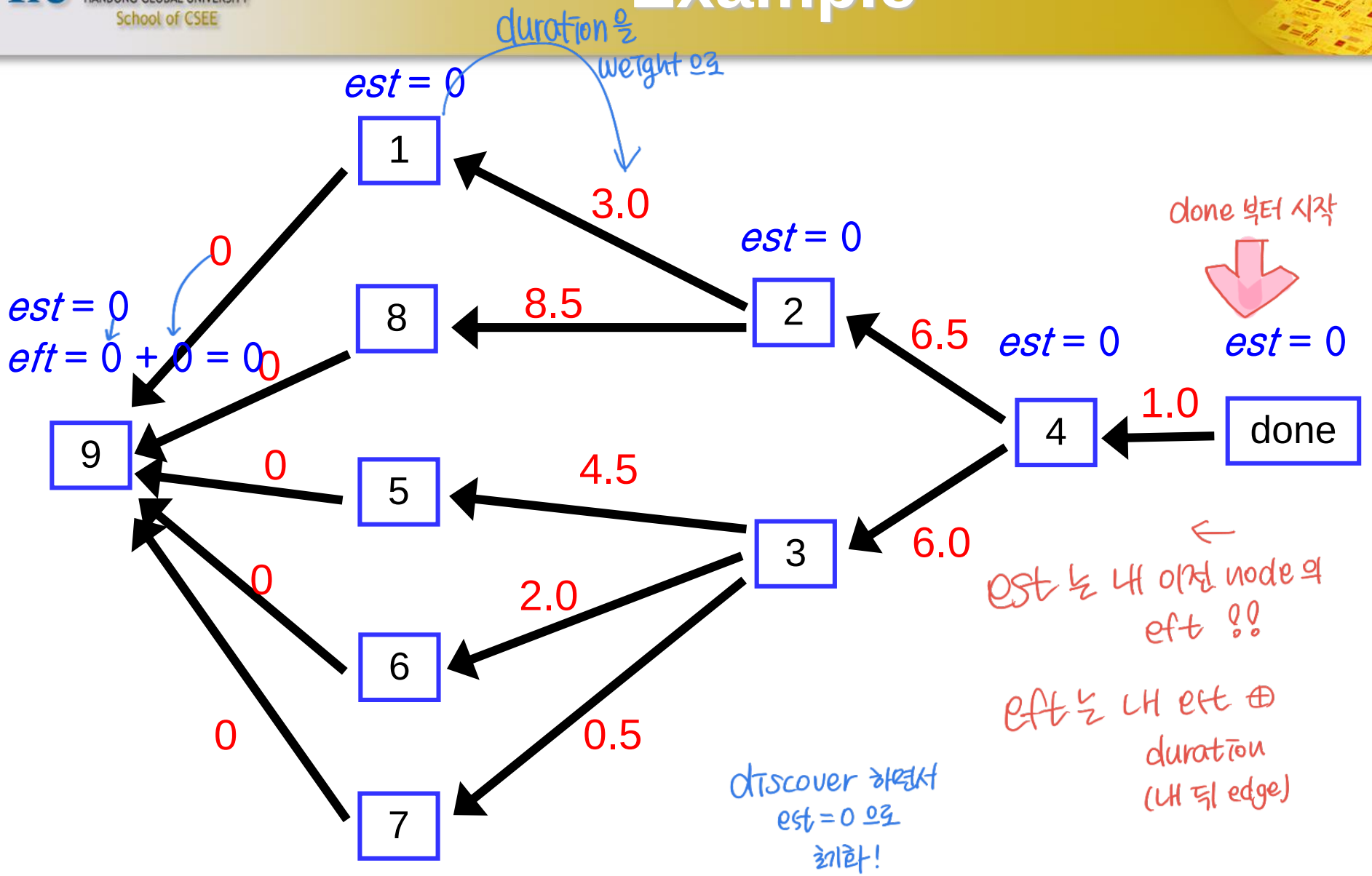


- At postorder processing time, insert

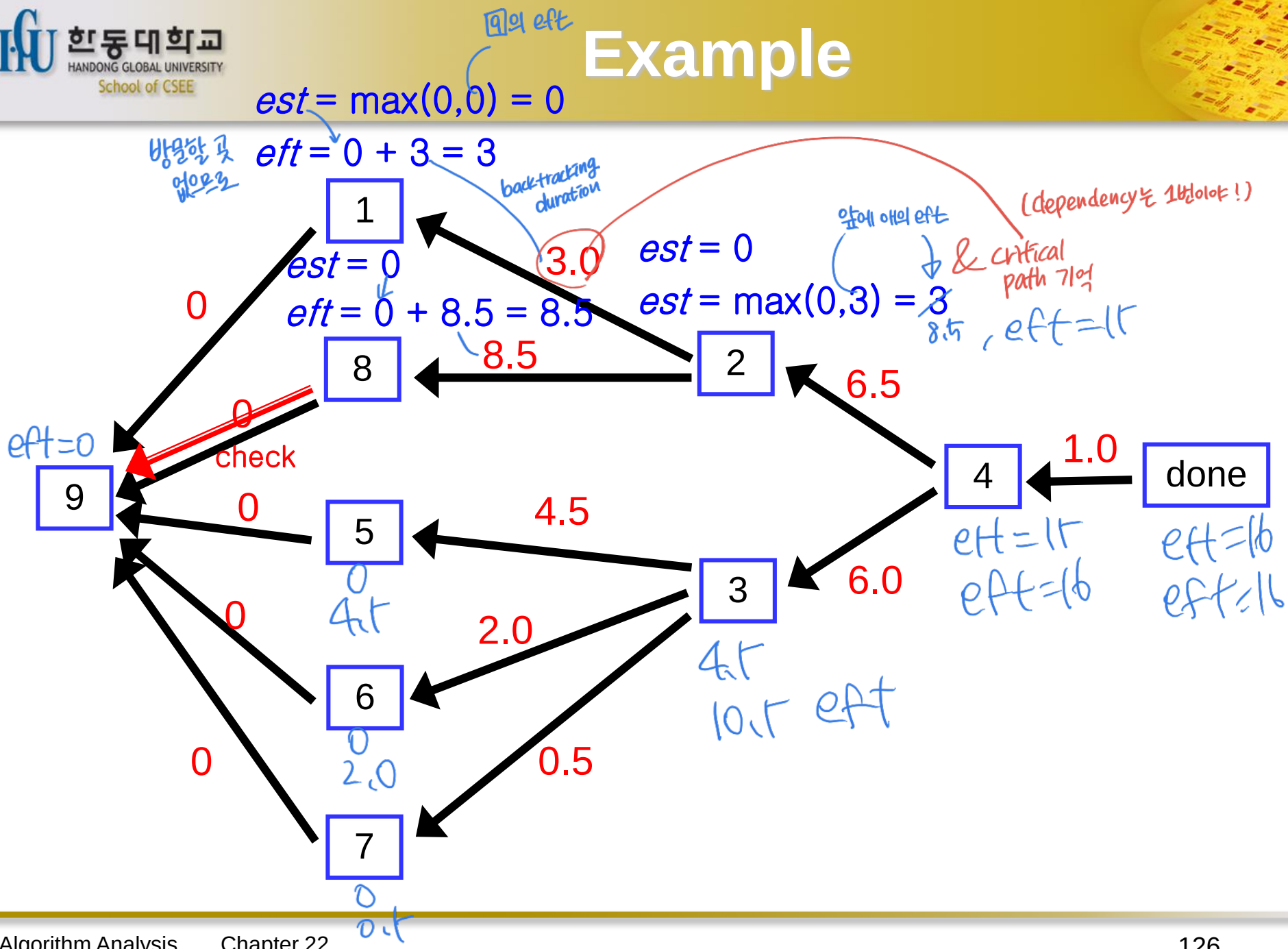
  eft[*v*] = *est*[*v*] + *duration*[*v*]

  (weight)

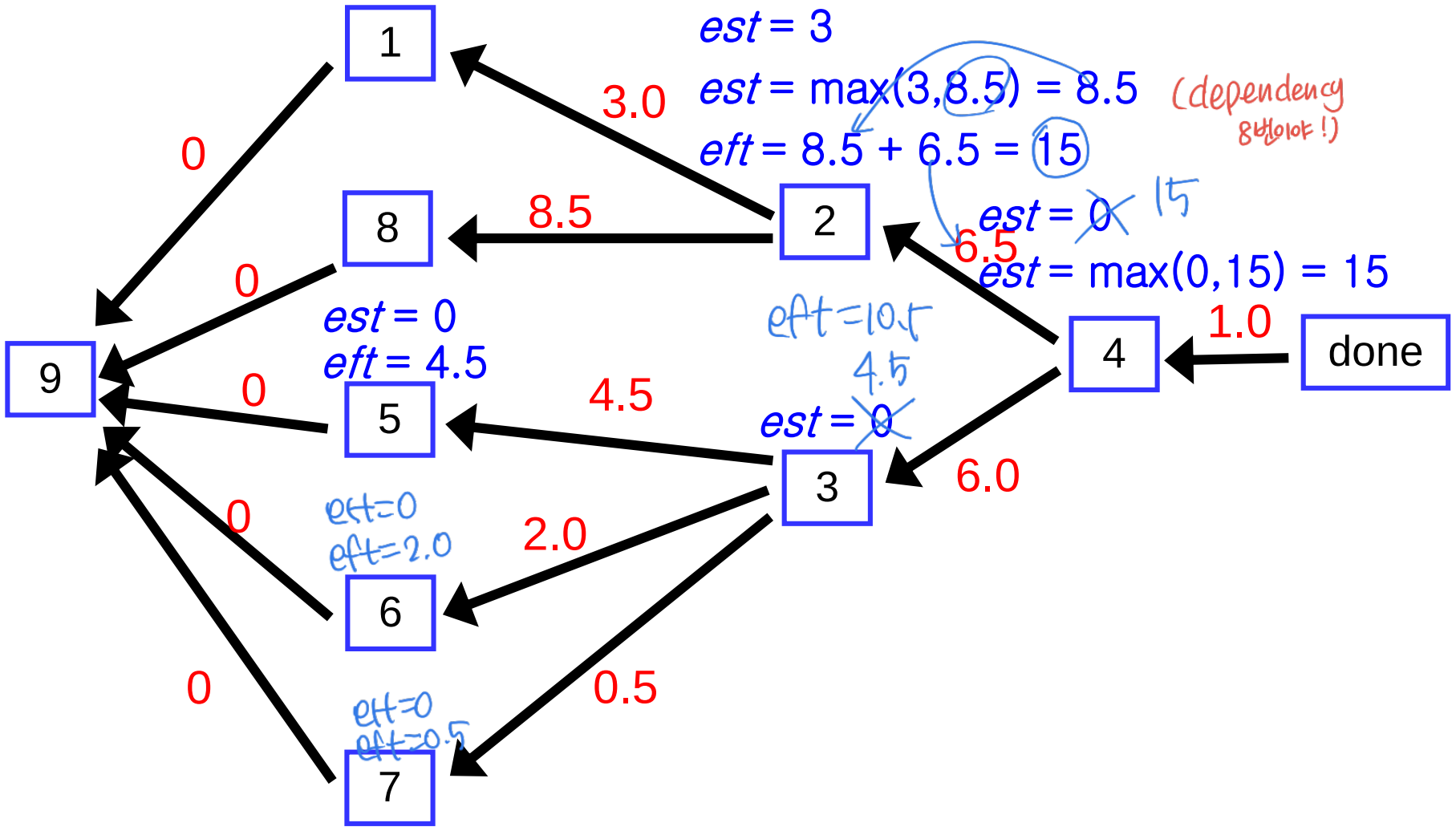
Example


duration을 weight으로
est = 0
1
3.0
0
est = 0
eft = 0 + 0 = 0
8
8.5
2
est = 0
6.5
done부터 시작
est = 0
est = 0
1.0
done
4
9
0
0
5
4.5
3
6.0
0
2.0
6
0
0.5
7
est는 내 이전 node의 eft !!
eft는 내 est + duration (내 뒤 edge)
discover 하면서 est = 0 으로 초기화!

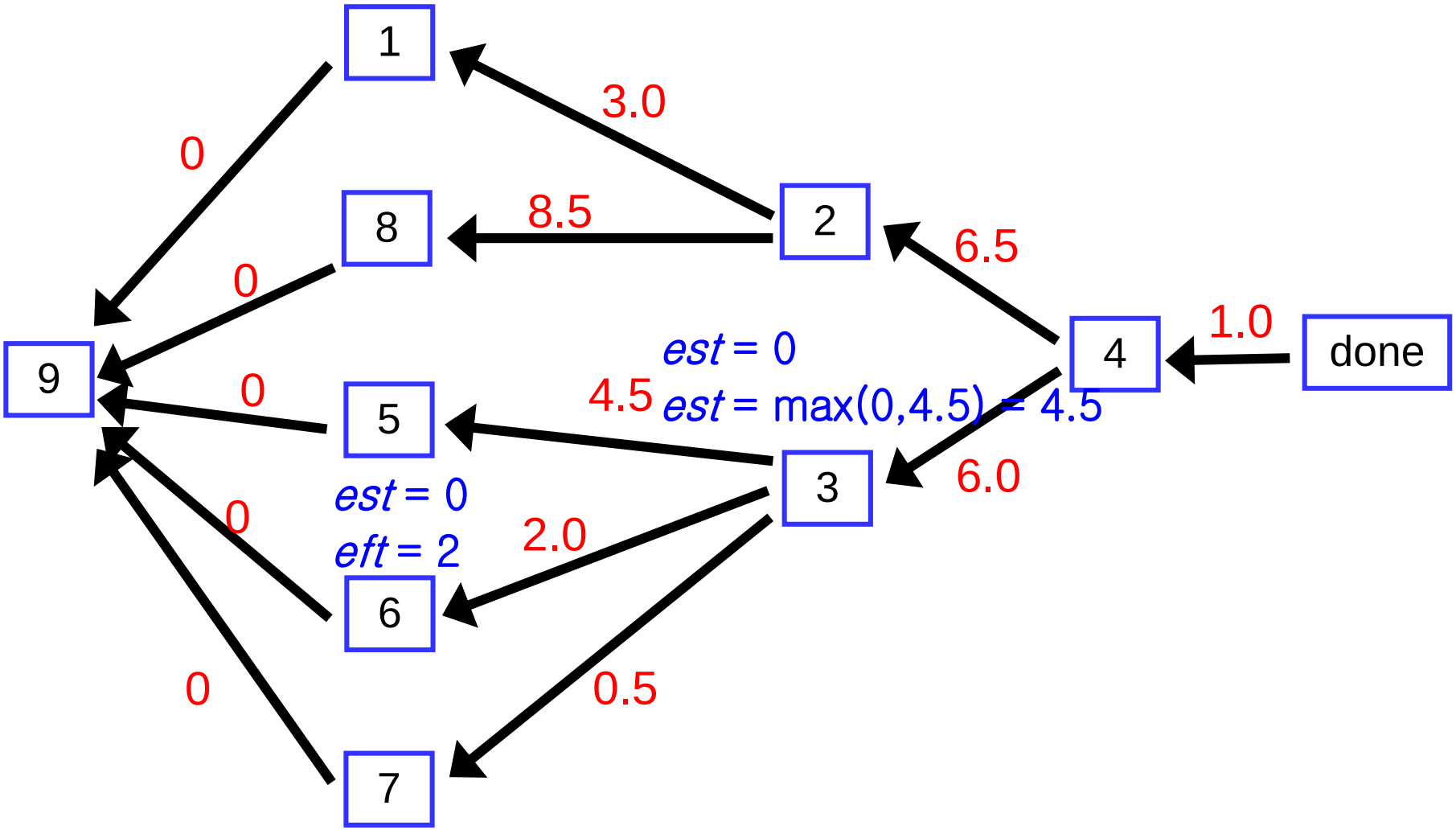
Example


est = max(0,0) = 0
eft = 0 + 3 = 3
back-tracking duration
3.0
est = 0
eft = 0 + 8.5 = 8.5
est = 0
est = max(0,3) = 3
& critical path 기억
(dependency는 1번이야!)
1
2
3
4
5
6
7
8
9
done
0
8.5
6.5
1.0
4.5
6.0
2.0
0.5
check
eft=0
eft=11
eft=16
eft=16

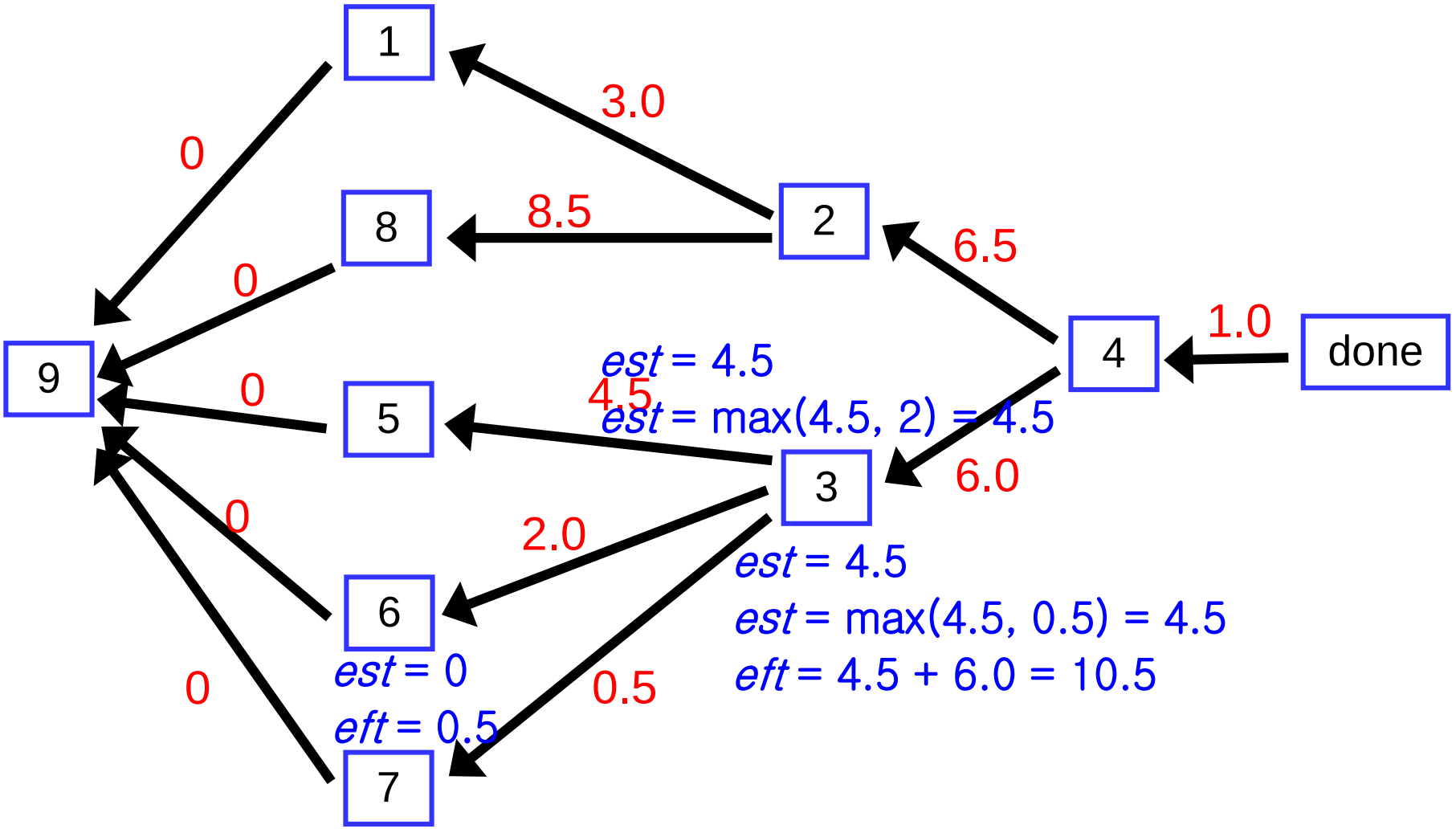
Example


1
2
3
4
5
6
7
8
9
done
est = 3
est = max(3,8.5) = 8.5
eft = 8.5 + 6.5 = 15
(dependency 생각해야 !)
est = 0
est = max(0,15) = 15
3.0
8.5
6.5
1.0
0
0
0
0
0
0
est = 0
eft = 4.5
4.5
est = 0
6.0
2.0
0.5
eft=0
eft=2.0
eft=0
eft=0.5

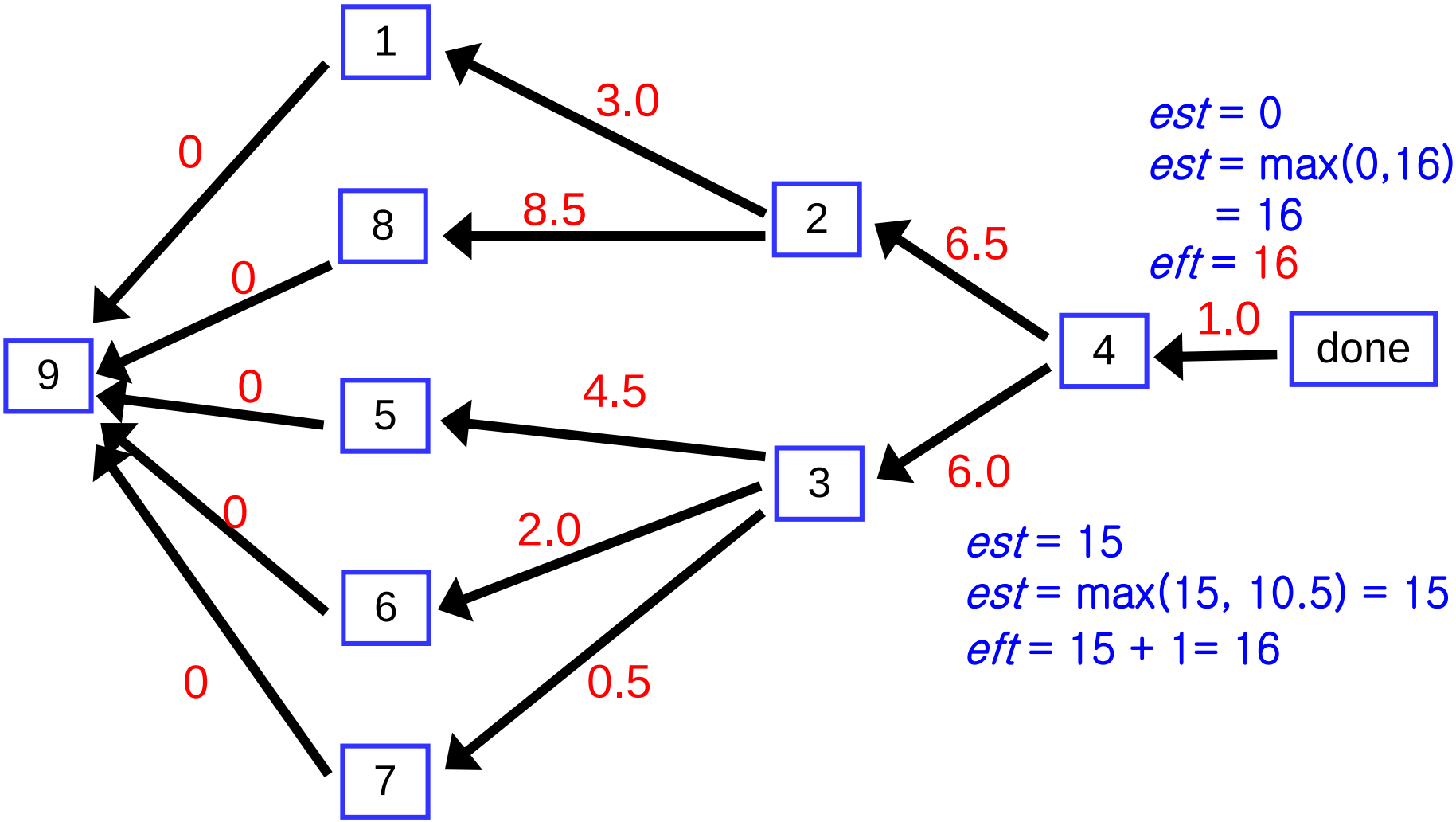
Example


1
2
3
4
5
6
7
8
9
done
3.0
0
8.5
0
6.5
1.0
est = 0
4.5
est = max(0,4.5) = 4.5
0
6.0
est = 0
eft = 2
0
2.0
0
0.5

Example


1
2
3
4
5
6
7
8
9
done
3.0
0
8.5
0
6.5
1.0
0
4.5
est = 4.5
est = max(4.5, 2) = 4.5
6.0
0
2.0
est = 4.5
est = max(4.5, 0.5) = 4.5
eft = 4.5 + 6.0 = 10.5
0
0.5
est = 0
eft = 0.5

Example


1
2
3
4
5
6
7
8
9
done
3.0
0
8.5
0
6.5
1.0
0
4.5
6.0
0
2.0
0
0.5
est = 0
est = max(0,16)
= 16
eft = 16
est = 15
est = max(15, 10.5) = 15
eft = 15 + 1= 16

5) Bi-connected components

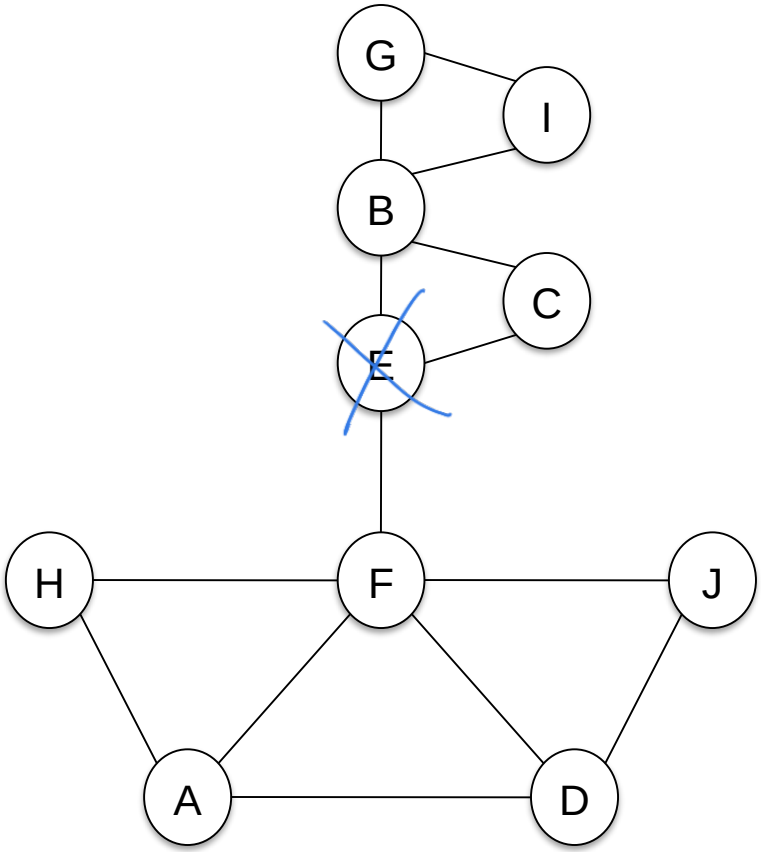
- Problem: 어떤 vertex 지우더도, 나머지가 다 connection 유지? → O Biconnected
  - If any *one* vertex (and the edges incident upon it) are removed from a connected graph, is the remaining subgraph still connected?
- Biconnected graph:
  - A connected undirected graph *G* is said to be biconnected if it remains connected after removal of any one vertex and the edges that are incident upon that vertex.

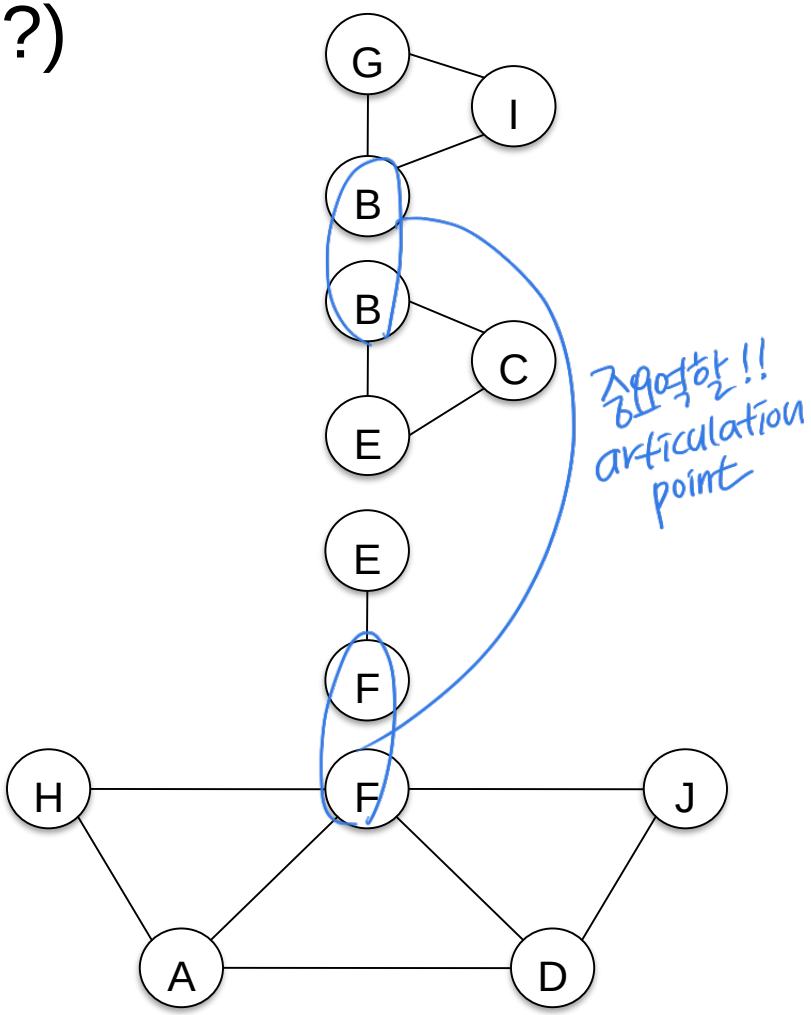
Bi-connected components

- Biconnected component:
  - A biconnected component of a undirected graph is a maximal biconnected subgraph, that is, a binconnected subgraph not contained in any larger binconnected subgraph.
- Articulation point: 그 길에 항상 개가 있음
  - A vertex *v* is an articulation point for an undirected graph *G* if there are distinct vertices *w* and *x* (distinct from *v* also) such that *v* is in every path from *w* to *x*.

Bi-connected components, e.g.

- Some vertices are in more than one component (which vertices are these?)





Determining Articulation Point

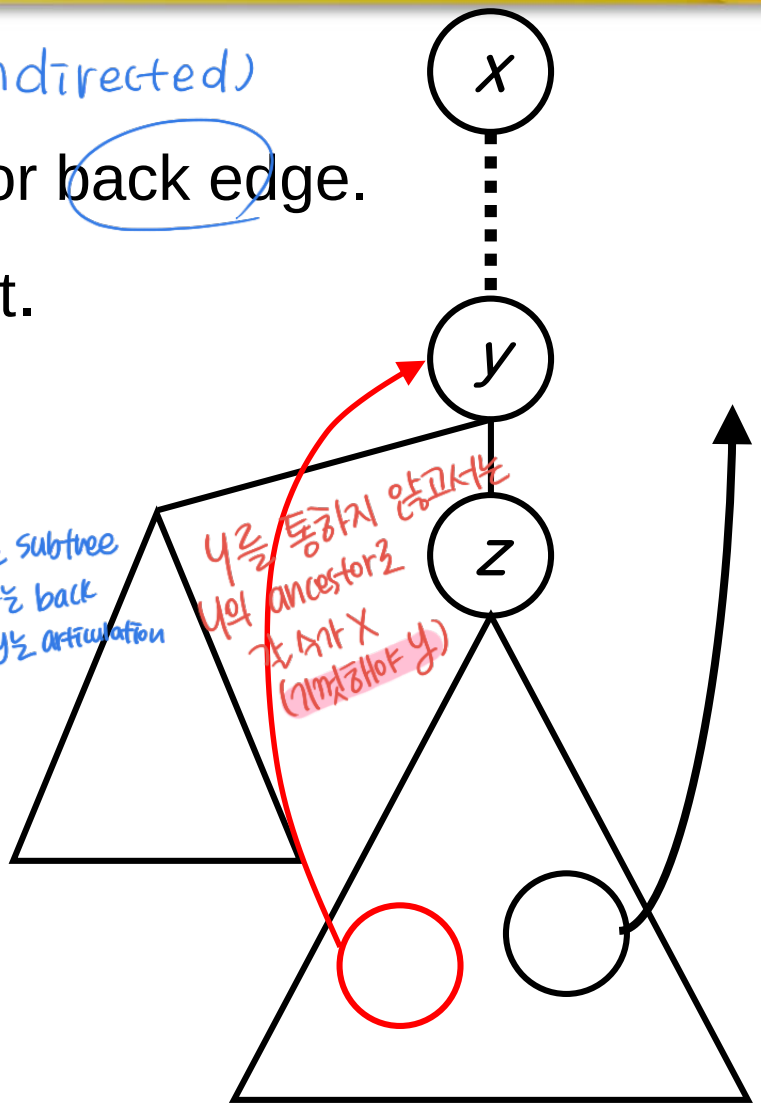
- Fact

  - Every edge is either tree edge or back edge.

  - Leaf cannot be articulation point.

  - When backing up from *z* to *y*, if there is no back edge from any vertex in the subtree rooted at *z* to proper ancestor of *y*, *y* is articulation point.

- Call DFS on the graph. We store '*back*'



Articulation Point

For all $v < y$, check back edge vw.

a. *back* is initialized to $d[v]$. (discovery time)

b. When encounteded with back edge vw

$back_v = \min(back_v, d(w))$

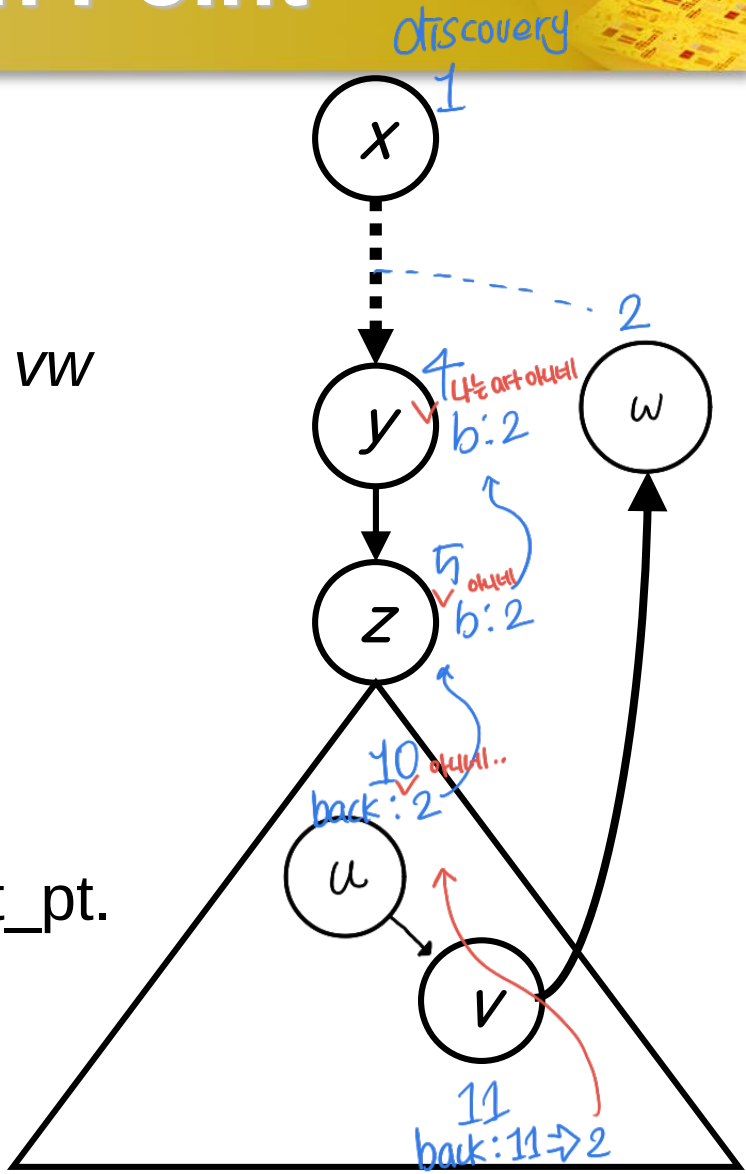
c. When backtracking from v to u,

$back_u = \min(back_u, back_v)$

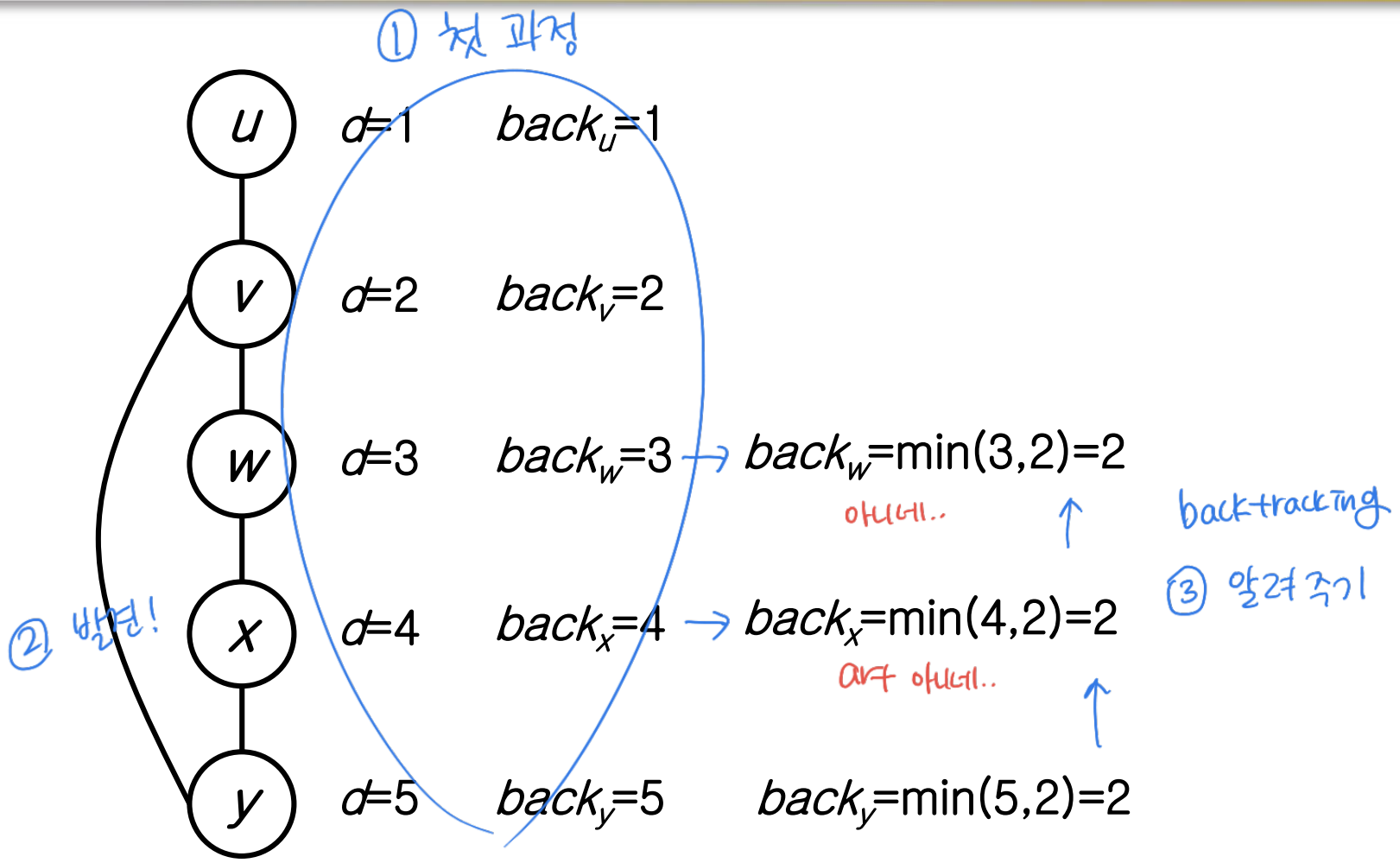
When backing up form z to y,

if all vertices ≤ z, $back_z \geq d(y)$. : y is art_pt.

else $back_z < d(y)$. : y isn't art_pt.

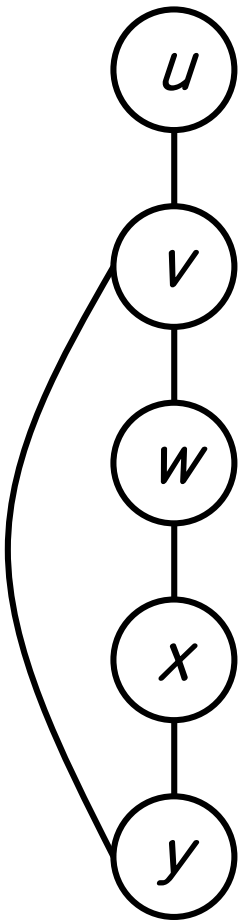


Example


① 첫 과정
u
v
w
x
y
d=1
back_u=1
d=2
back_v=2
d=3
back_w=3
back_w=min(3,2)=2
아니네..
back-tracking
③ 알려주기
② 발견!
d=4
back_x=4
back_x=min(4,2)=2
art 아니네..
d=5
back_y=5
back_y=min(5,2)=2

Example

root는 art가 될 수 X



d=1 $back_u$=1

d=2 $back_v$=2 $(back_w=2) \geq (d(v)=2)$: v is art_pt

나다!

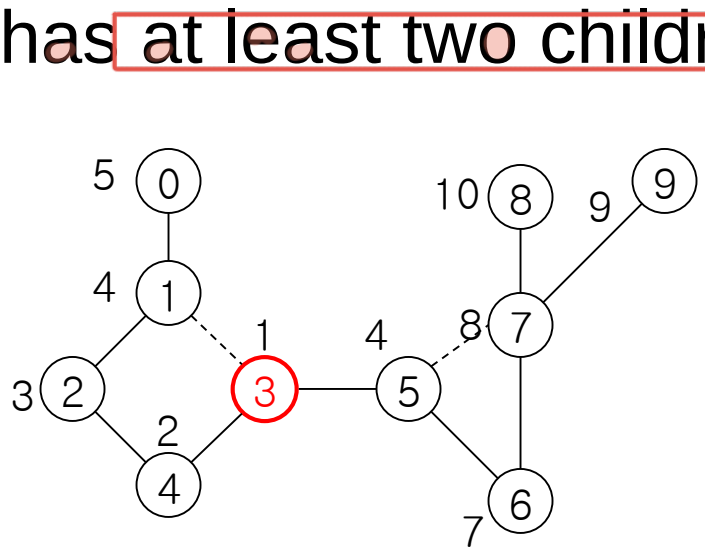
d=3 $back_w$=3 $(back_x=2) < (d(w)=3)$: w isn't art_pt

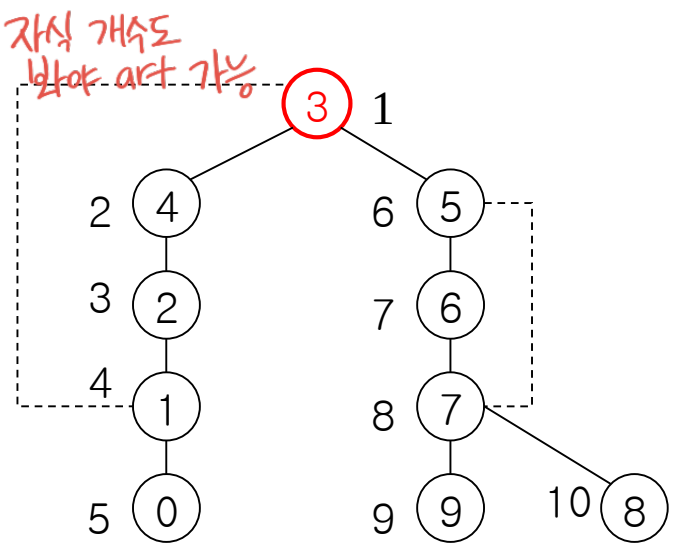
d=4 $back_x$=4 $(back_y=2) < (d(x)=4)$: x isn't art_pt

d=5 $back_y$=5 $back_y=\min(5,2)=2$

Biconnected Component and Articulation Points

- A root of DFS tree is an articulation point iff it has at least two children.

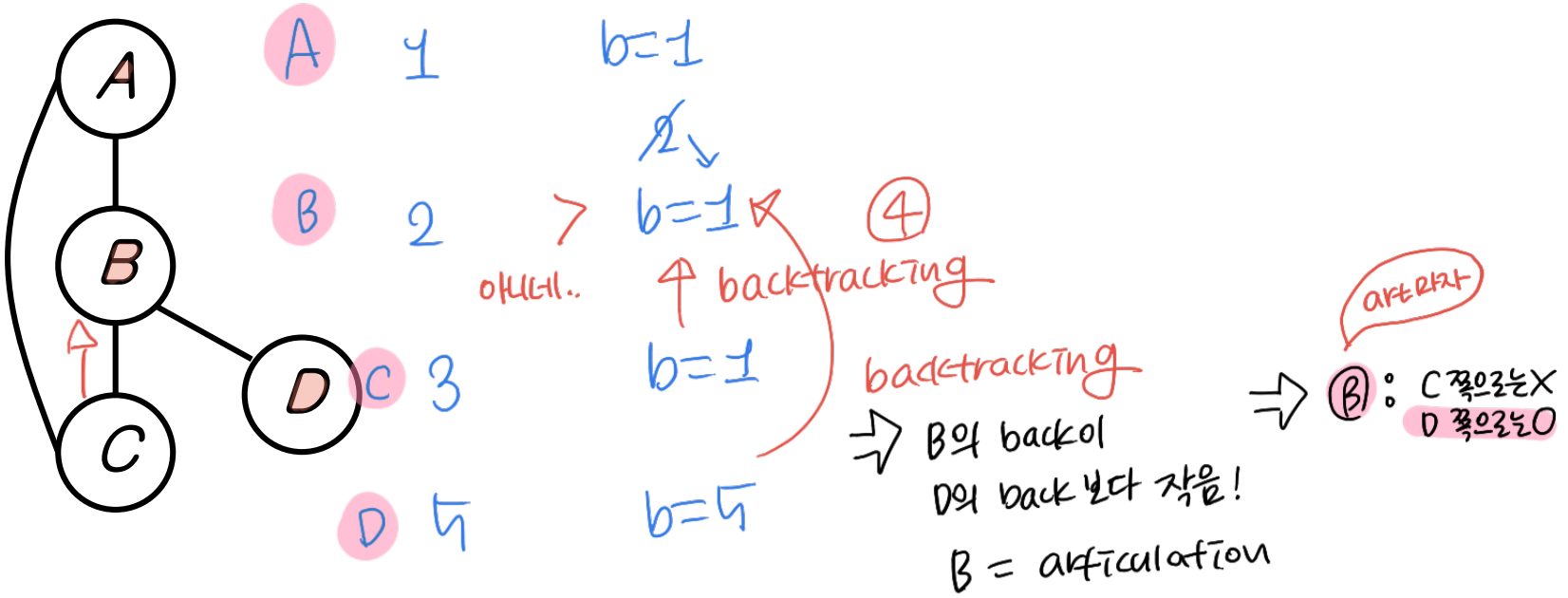




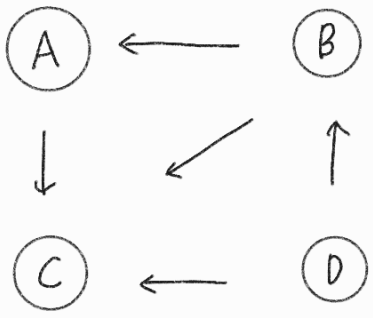
- A non-root vertex *u* is an articulation point iff it has at least one child *w* s.t. there is no back edge from ant vertex in the subtree rooted *w* to a proper ancestor of *v*.

Exercise

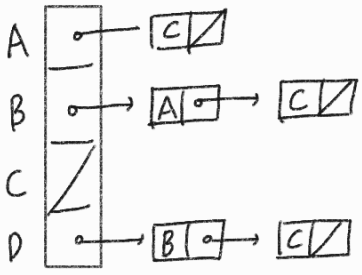
Identify articulation point on the following graph. Assume the vertices are in alphabetical order in the *Adj* array and that each adjacency list is in alphabetical order.



< Question 1 >



- adjacency list, matrix

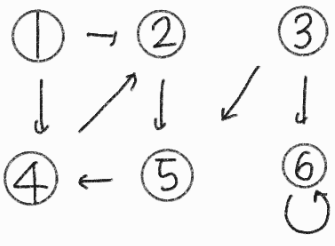


	A	B	C	D
A	0	0	1	0
B	1	0	1	0
C	0	0	0	0
D	0	1	1	0

- space complexity

$\Theta(V+E)$ $\Theta(V^2)$

< Question 2 >



breadth-first search

vertex 3

	pred π	dist d
3	NIL	0
5	3	1
6	3	1
4	5	2
2	4	3
1	NIL	∞

< Question 3 >

running time of BFS

(1) adjacency list ? $\Theta(V+E)$

(2) matrix ? $\Theta(V^2)$ 셀의 개수

⊕ running time의 OFS?

① list ? $\Theta(V+E)$

② matrix ? $\Theta(V^2)$

✓ Can u think asymptotically faster algorithm ?

→ 없음

< Question 4 >

(Discovery time, Finish time 찾기)

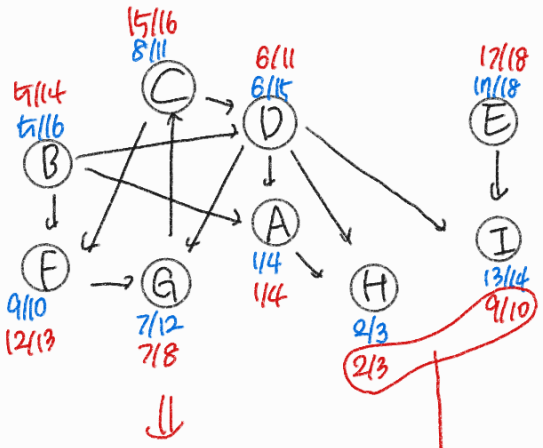
Determine the number of following

① Tree edges 6

② Forward edges 1

③ Back edges 2

④ cross edges 4



data structure에 algorithm 적용

< Exercise 2 >

Strongly connected component를 찾아라.

* After step 1

→ transpose 취하고 → finish time 가장 큰 vertex 부터 DFS

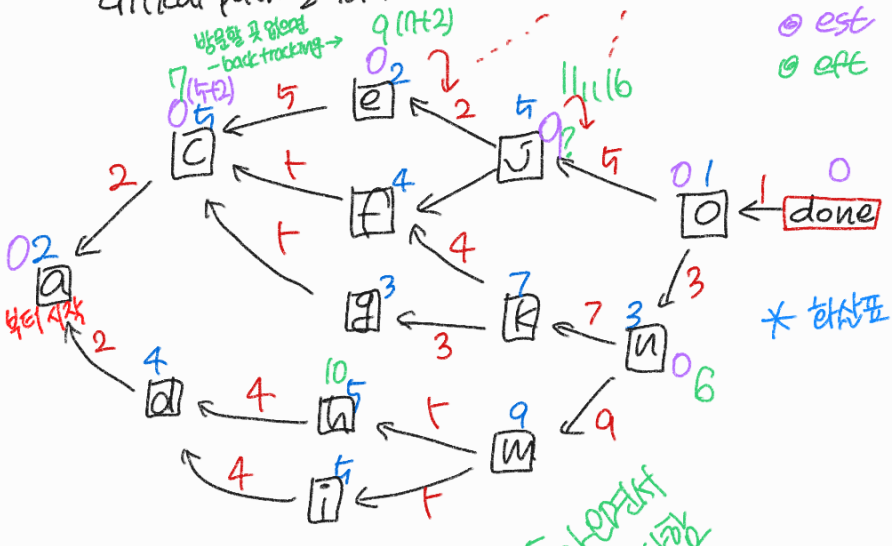
< Exercise 3 >

critical path를 구하라.

(node) duration → (edge) weight 변환

* est
* eft

방향을 꼭 알려줌 - back tracking →



* 화살표 방향 바뀐 것 이어

답 24

0 → C → e → J → O

h → k → f → k → n

n → k → g → n

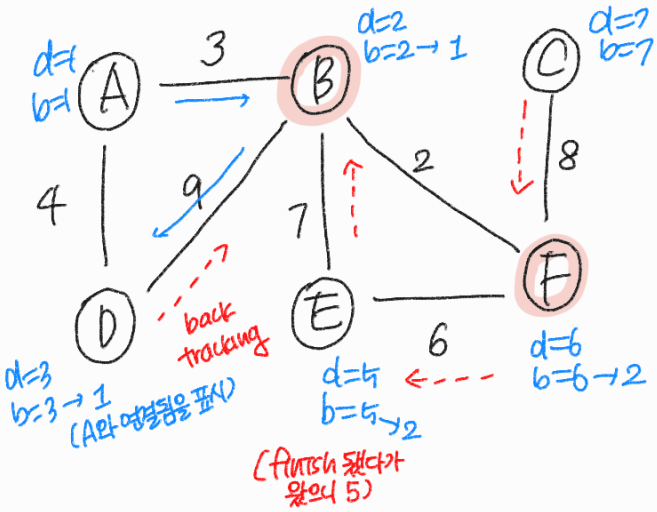
→ m → h → d → h → m

→ T → d → m → n → o

<Exercise 4>

Determine the value d[E] and back$_F$ when the node B is determined as articulation point. Assume adjacency array and adjacency list in alphabetical order.

(백트래킹 할 때마다)



back-tracking

tree edge

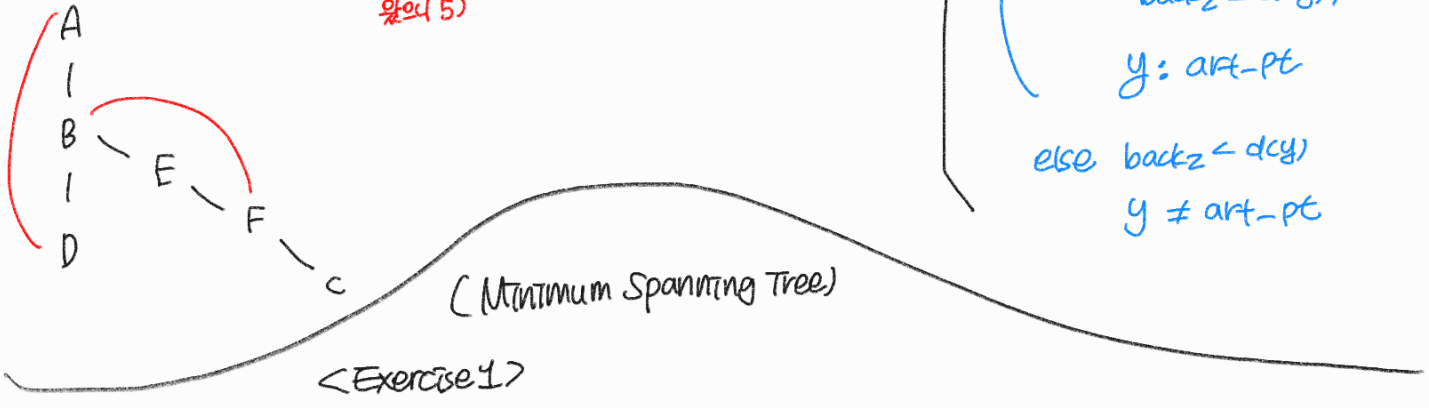
back

두 가지 경우 때 back-tracking

When backing up z-to y,

If all vertices ≤z,
back$_z$ ≥ d(y),
y : art-pt

else back$_z$ < d(y)
y ≠ art-pt



<Exercise 1>

Prim algorithm에서 사용된 structure

: priority queue , greedy (어느 상황에 크고 작은 것을 결정)

(logn)

(key 값이 바뀔 수 있기 때문에)

OR

가장 좋은 방법을 찾아요

-sorting ?

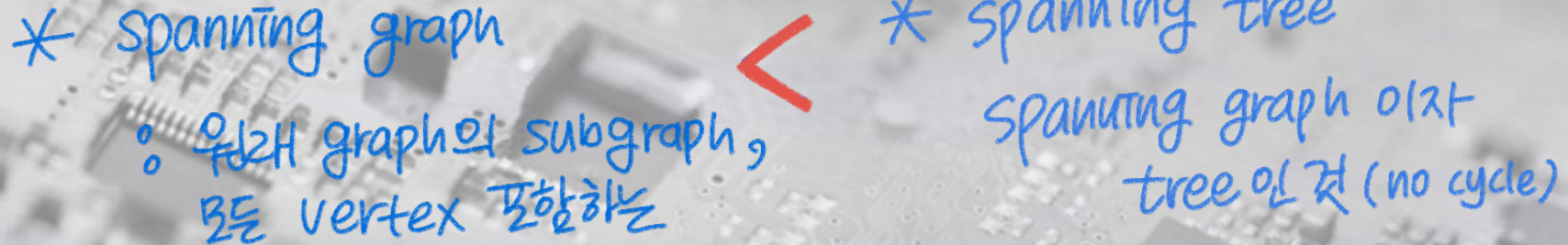
(nlogn) 시간 오래 걸림

HANDONG GLOBAL UNIVERSITY

Chapter 23
Minimum Spanning Trees

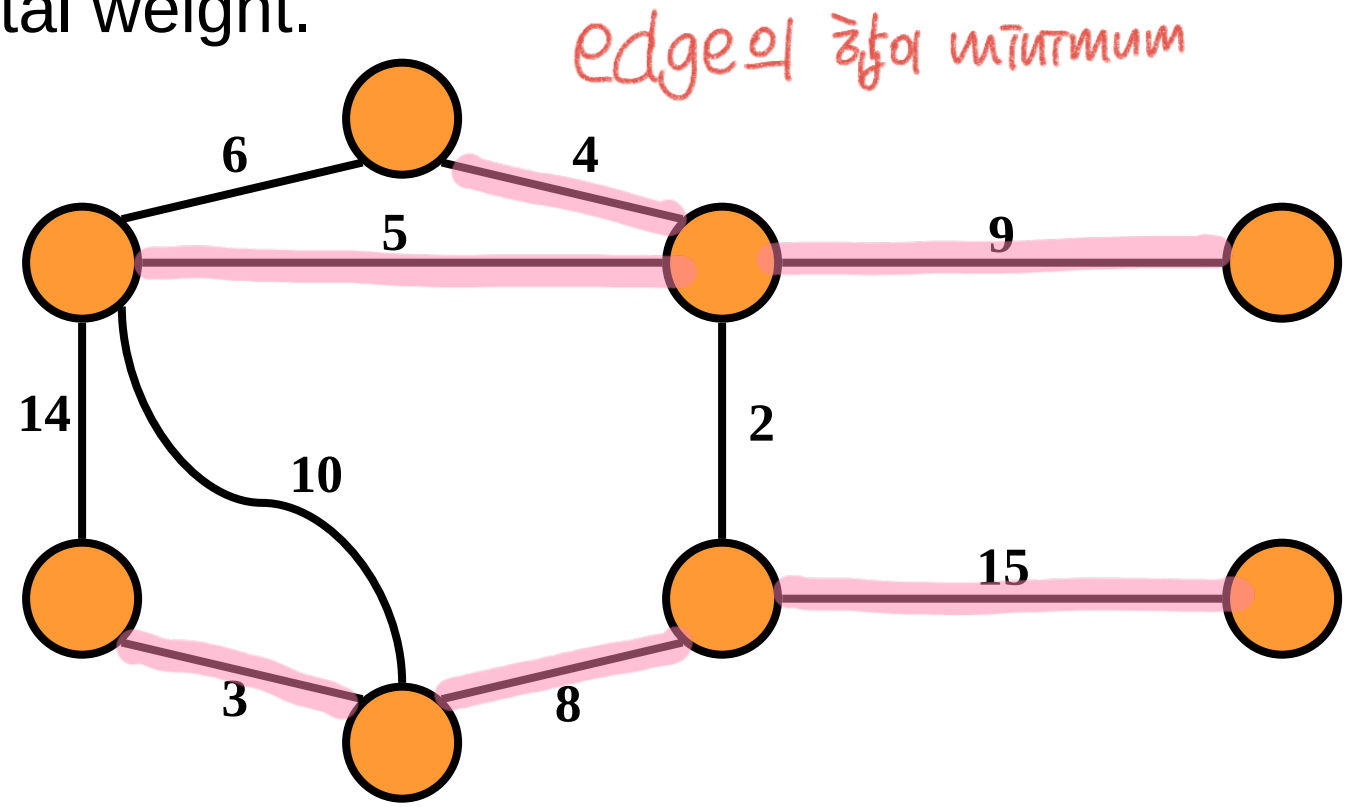
Algorithm Analysis

School of CSEE



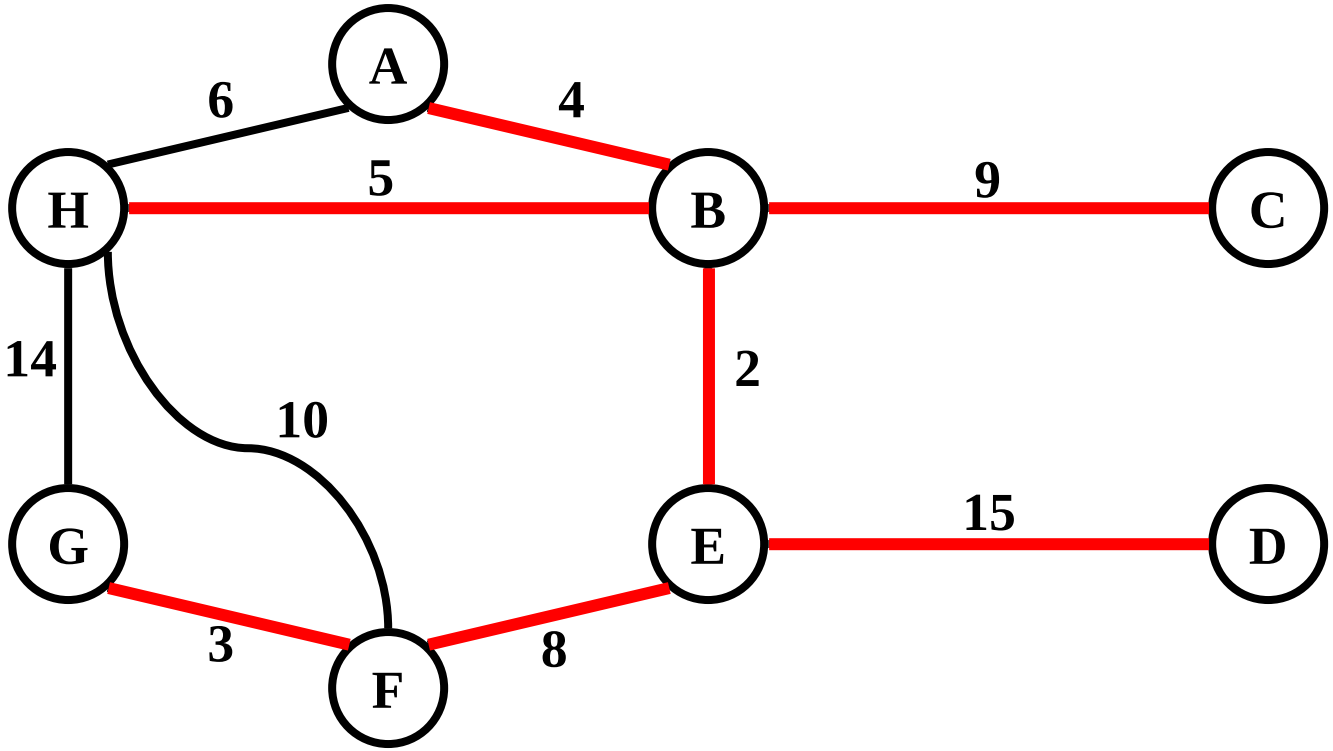
Minimum Spanning Tree

- Problem: given a connected, undirected, weighted graph, find a *spanning tree* using edges that minimize the total weight.



Minimum Spanning Tree

- *Which edges form the minimum spanning tree (MST) of the below graph?*



Minimum spanning tree

- Undirected graph $G = (V, E)$
- Weight $w(u,v)$ on each edge $(u,v) \in E$
- Spanning tree of G' is a minimal subgraph of G such that
  - $V(G') = V(G)$ and G' is connected.
  - Any connected graph with n vertices must have at least n-1 edges. All connected graphs with n-1 edges are trees.

no cycle

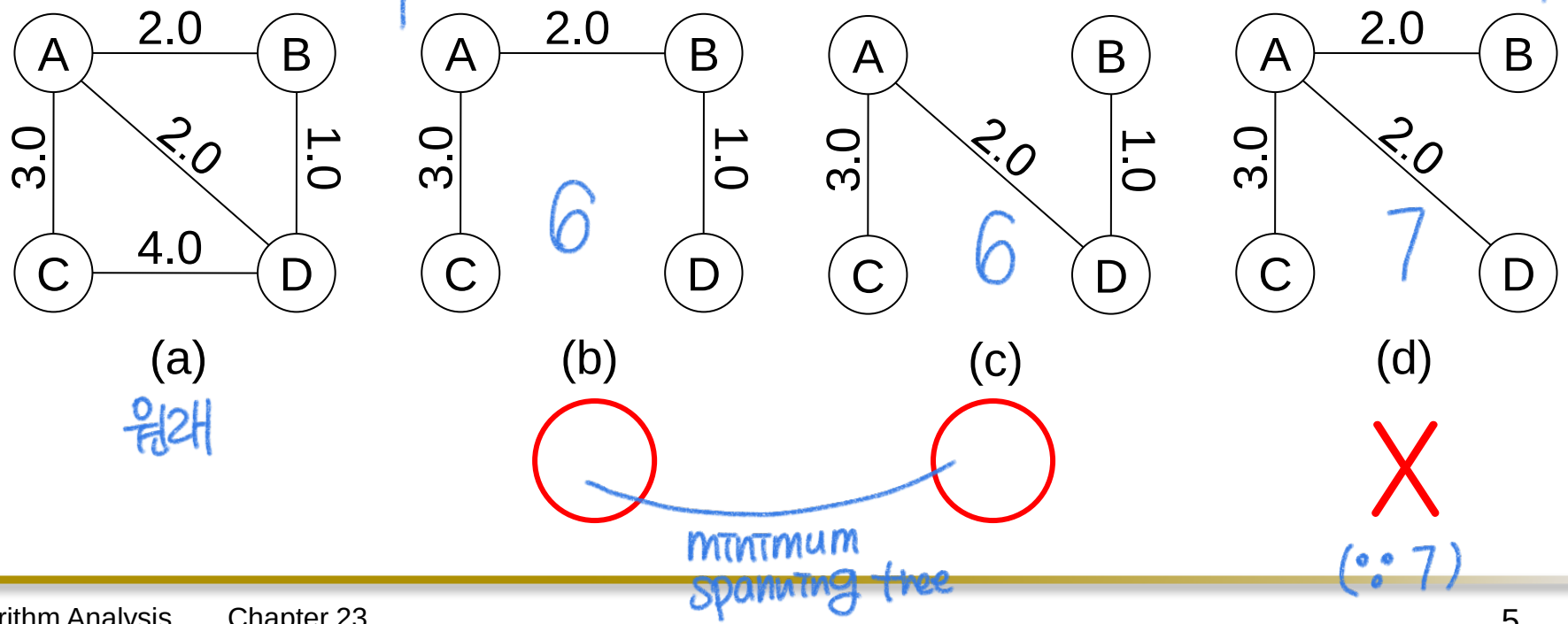
- Find $T \subseteq E$ s.t.
  - T connects all vertices (T is a spanning tree), and
  - $w(T) = \sum_{(u,v) \in E} w(u,v)$ is minimized.

모든 edge의 합이 minimum

Minimum spanning tree

- A spanning tree whose weight is minimum over all spanning trees is called a minimum spanning tree or MST.
  - Has *n* -1 edges, no cycle, might not be unique

4개 node, 3개 (n-1) edge ⇒ tree!
no cycle



Minimum spanning tree

- Example)

  Interconnect *n* pins with *n*-1 wires, each connecting two pins so that we use the least amount of wire.

- We'll look at two greedy algorithms.
  - Kruskal's algorithm
  - Prim's algorithm

Building up the solution

- Build a set A of edges.
- Initially, A is empty.
- As we add edges to A, maintain a loop invariant : A is a subset of some MST.
- Add only edges that maintain the invariant.
  - If A is a subset of some MST, an edge (u,v) is *safe* for A if and only if $A \cup \{ (u,v) \}$ is also a subset of some MST.
  - So, we will add only safe edges.

Generic MST algorithm

GENERIC-MST(*G*, *w*)

$A = \varnothing$ (Initial)

while A is not a spanning tree

  do find an edge (u,v) that is safe for A

    $A = A \cup \{ (u,v) \}$

return A

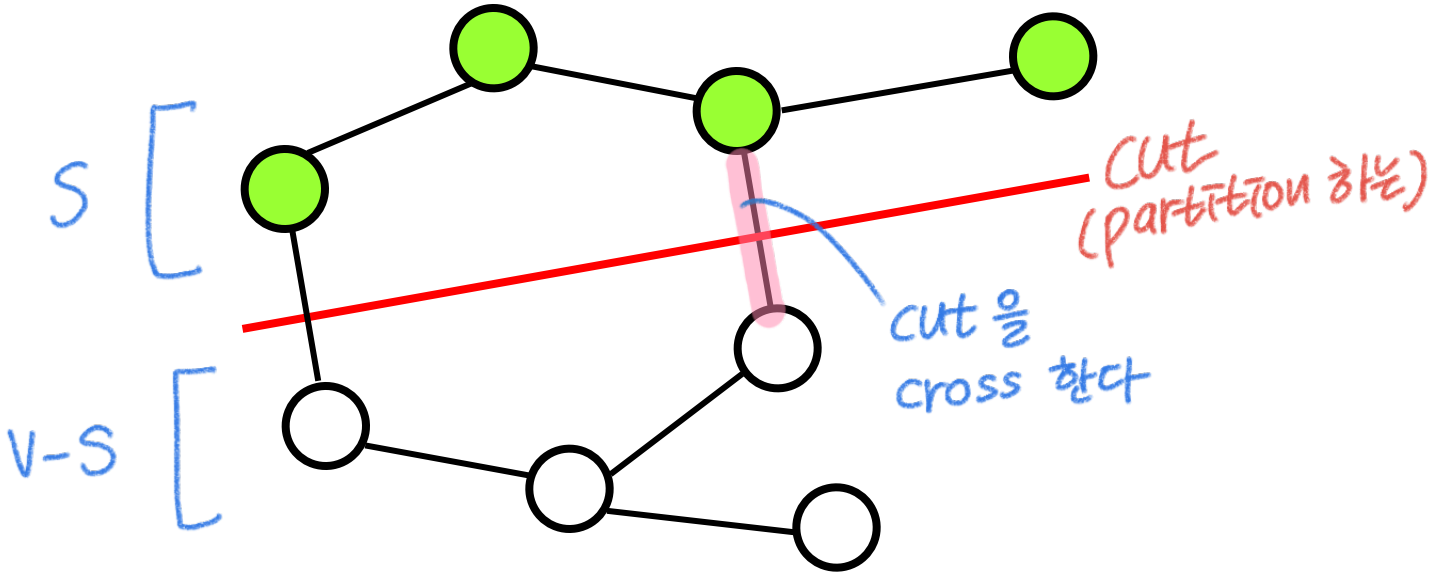
Some MST의 edge

그 순간 looks best! (smallest or largest)

e greedy

Finding a safe edge

- Let $S \subset V$ and $A \subseteq E$.
  - A *cut* $(S, V - S)$ is a partition of vertices into disjoint sets S and $V - S$.
  - Edge $(u,v) \in E$ *crosses* cut $(S, V - S)$ if one endpoint is in S and the other is in $V - S$.

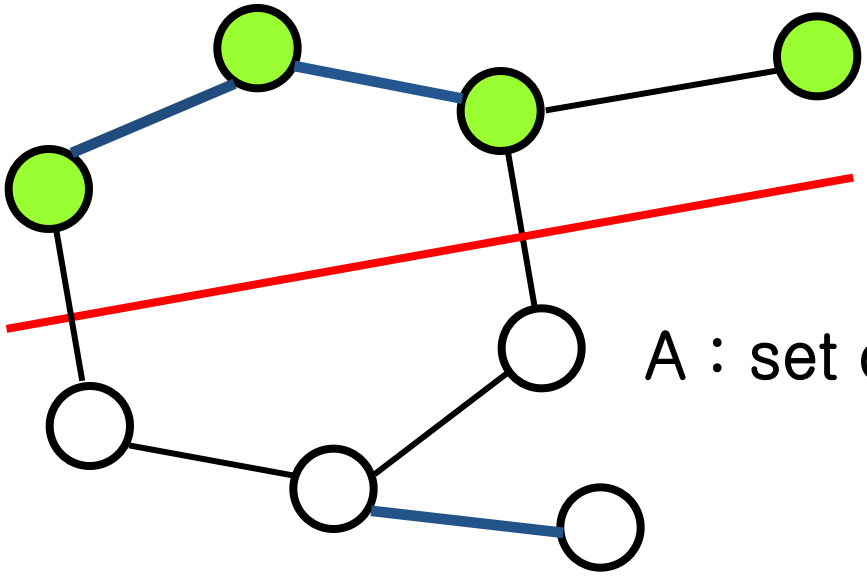


Finding a safe edge

- Let $S \subset V$ and $A \subseteq E$.
  - A cut *respects* A if and only if no edge in A crosses the cut. A의 어느 edge도 cut을 cross 하지 X
  - An edge is a *light edge* crossing a cut if and only if its weight is minimum over all edges crossing the cut. For a given cut, there can be more than one light edge crossing it.

cut cross 하는 것들 중 weight 최소인 것

(tie 이면, light edge 여러 개)



A : set of blue edges

Theorem

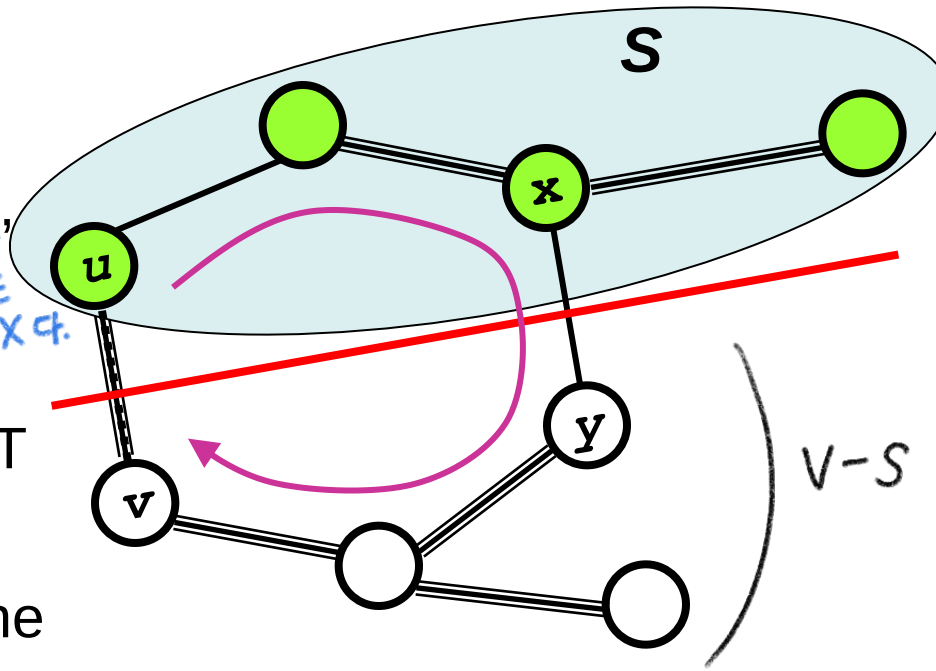
[결론] light edge = safe edge

Let A be a subset of E in some MST, $(S, V - S)$ be a cut that respects A, and (u, v) be a light edge crossing $(S, V - S)$. Then, (u, v) is safe for A.

Proof) MST

1. Let T be a MST that includes A, and assume that T does not contain the light edge (u, v). (uv는 포함X 다.)
2. We shall construct another MST T' that includes $A \cup \{ (u,v) \}$.
3. Edge (u,v) forms a cycle with the edges on the path p from u to v in T. T에 uv 포함

edge 개수 : n-1 + 1 ⇒ n ⇒ cycle



: Edges in A

Theorem

4. Since u and v are on opposite sides of the cut $(S, V - S)$ there is at least one edge in T on the path p that also crosses the cut. Let (x,y) be any such edge.

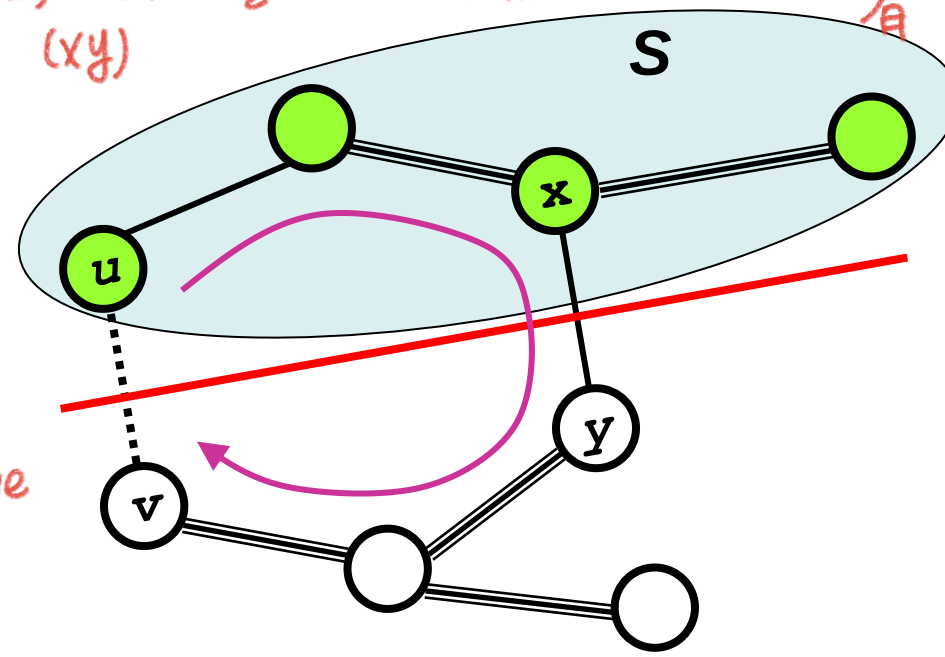
5. Removing (x,y) breaks T into two components. And adding (u,v) reconnects them to form a new spanning tree T'.

$$w(T') = w(T) - w(x,y) + w(u,v)$$

$$\leq w(T)$$

Thus, T' must be a MST.

And since $A \cup \{ (u,v) \} \subseteq T'$, (u,v) is safe for A.



Corollary

Let A be a subset of E that is included in some minimum spanning tree for G, and let $C = (V_c, E_c)$ be a connected component (tree) in the forest $G_A = (V, A)$. If (u,v) is a light edge connecting C to some other component in G_A, then (u,v) is safe for A.

Proof) The cut (Vc, $V - Vc$) respects A, and (u,v) is a light edge for this cut. Therefore, (u,v) is safe for A.

Basic idea of Kruskal's algorithm

- Sort edges into nondecreasing order by *w*.
- The algorithm maintains *A*, a forest of trees.
- Repeatedly merges two components into one by choosing the light edge that connects them.

  i.e.,

  1. Choose the light edge crossing the cut between them.
  2. (If it forms a cycle, the edge is discarded.)

- What is the design strategy of Kruskal's algorithm?

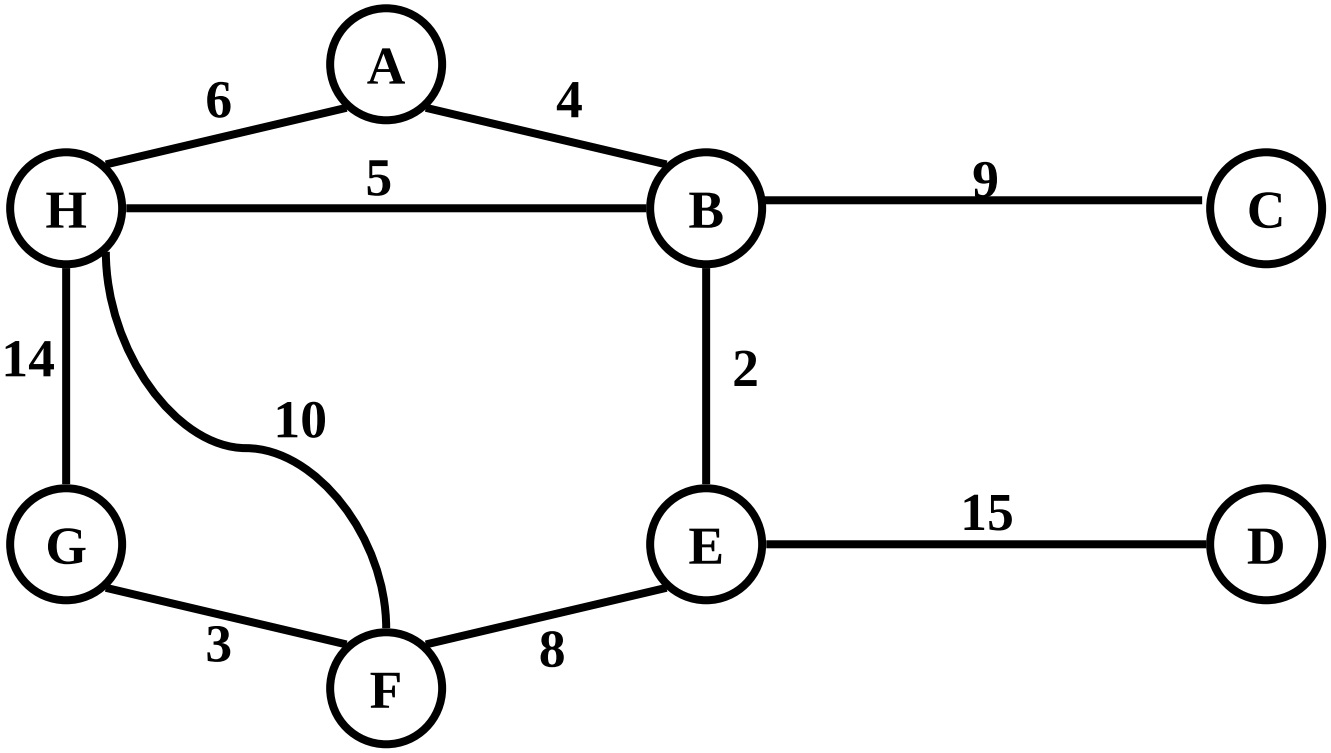
  - Greedy !!

Specific pseudo-code

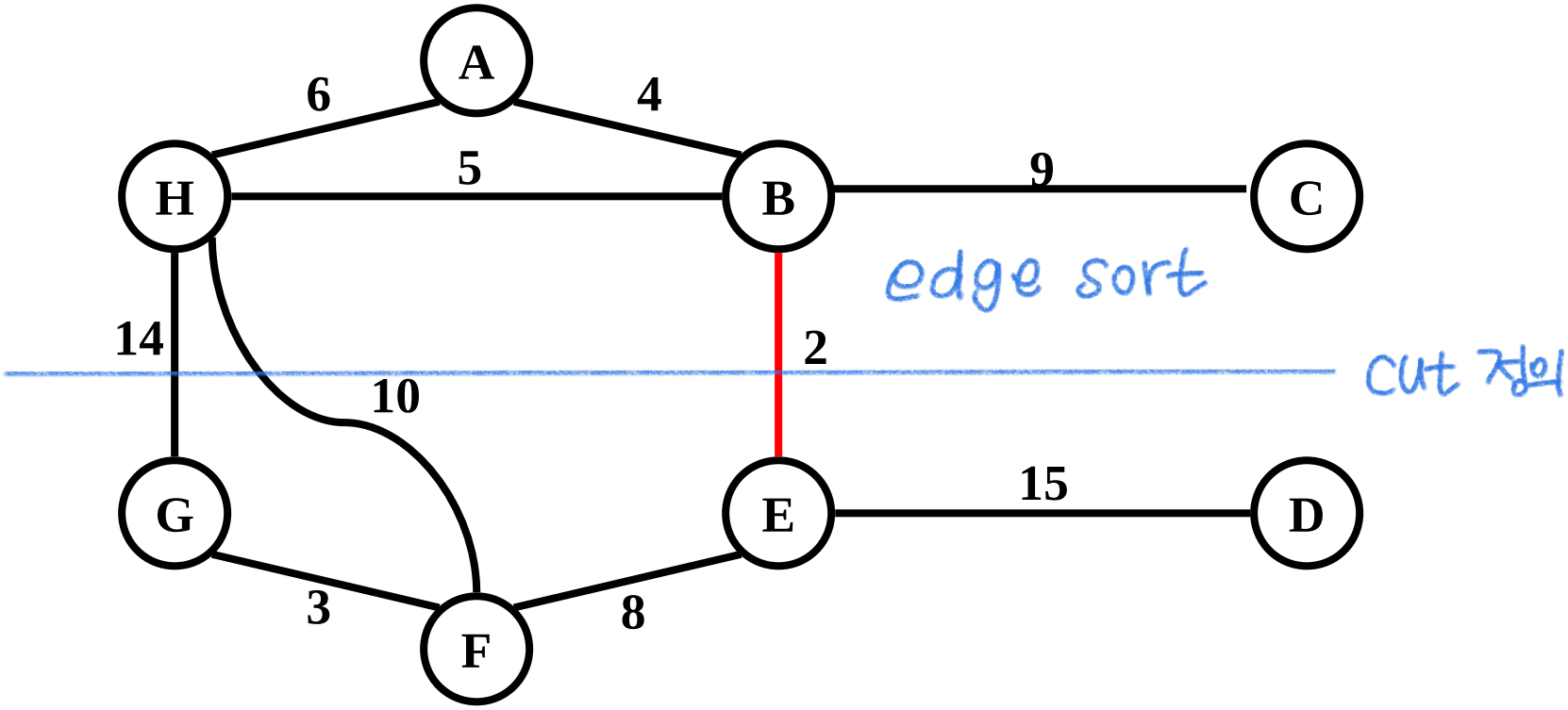
MST-Kruskal(G,w)

```
R = E;
F = 0;
While (R is not empty)
    Remove the light edge, (u,v), from R;
    if ((u,v) does not make a cycle in F)
        Add (u,v) to F;
return F;
```

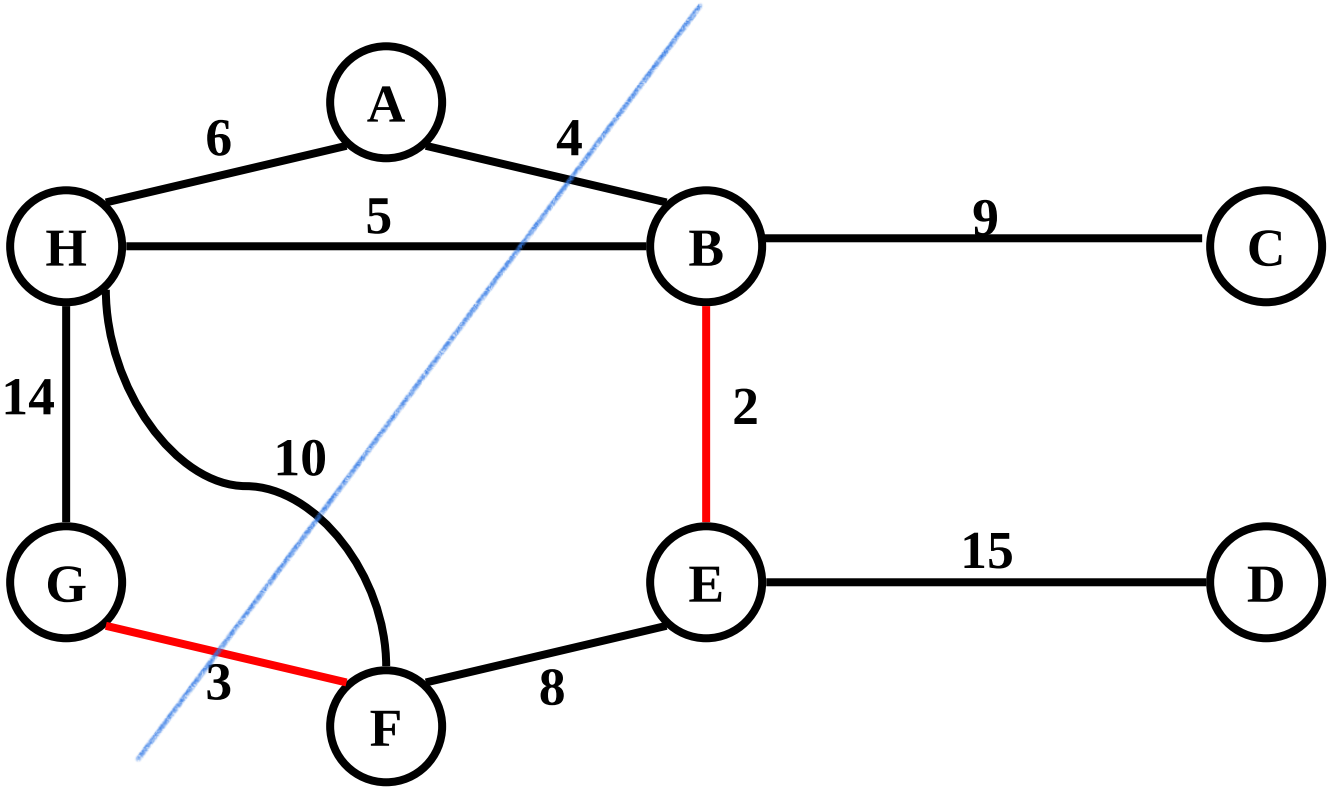
Example of Kruskal



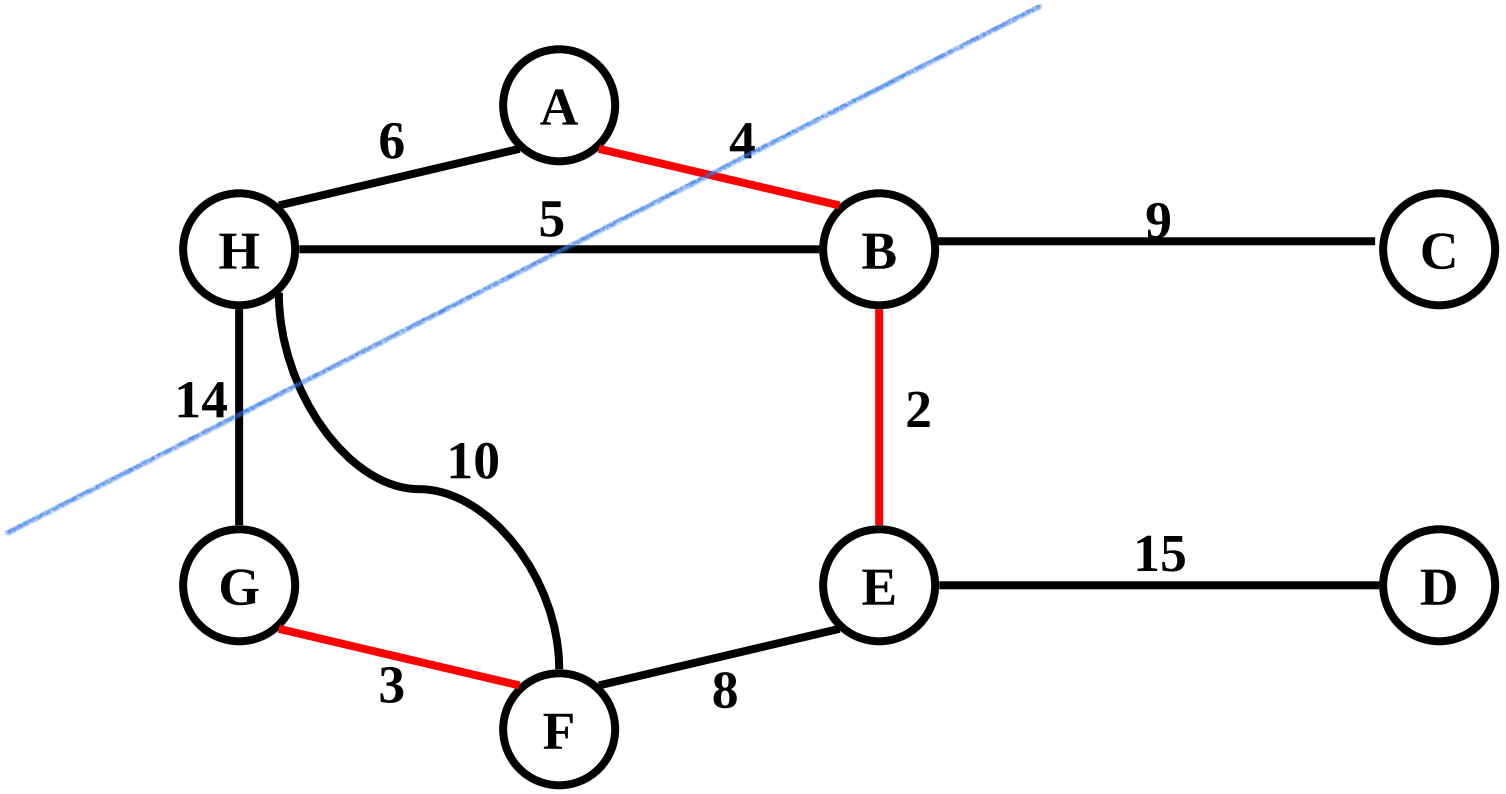
Example of Kruskal



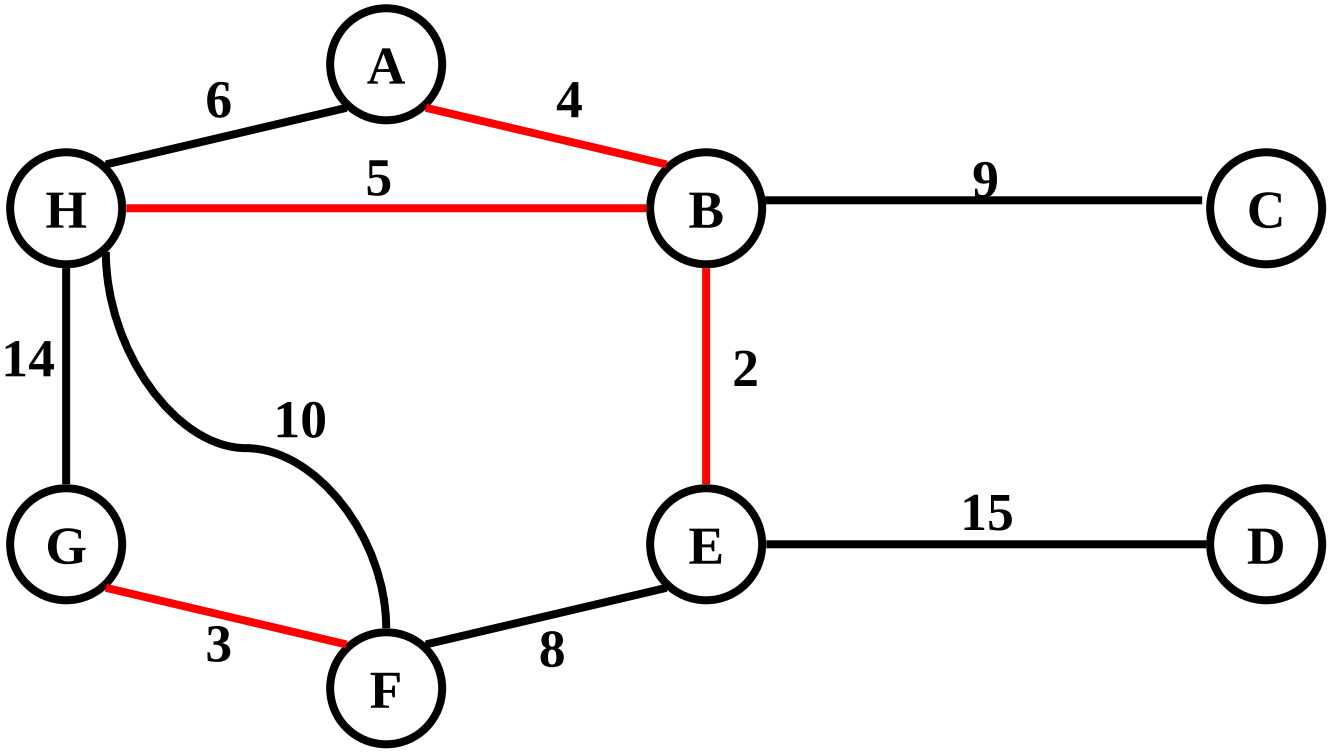
Example of Kruskal



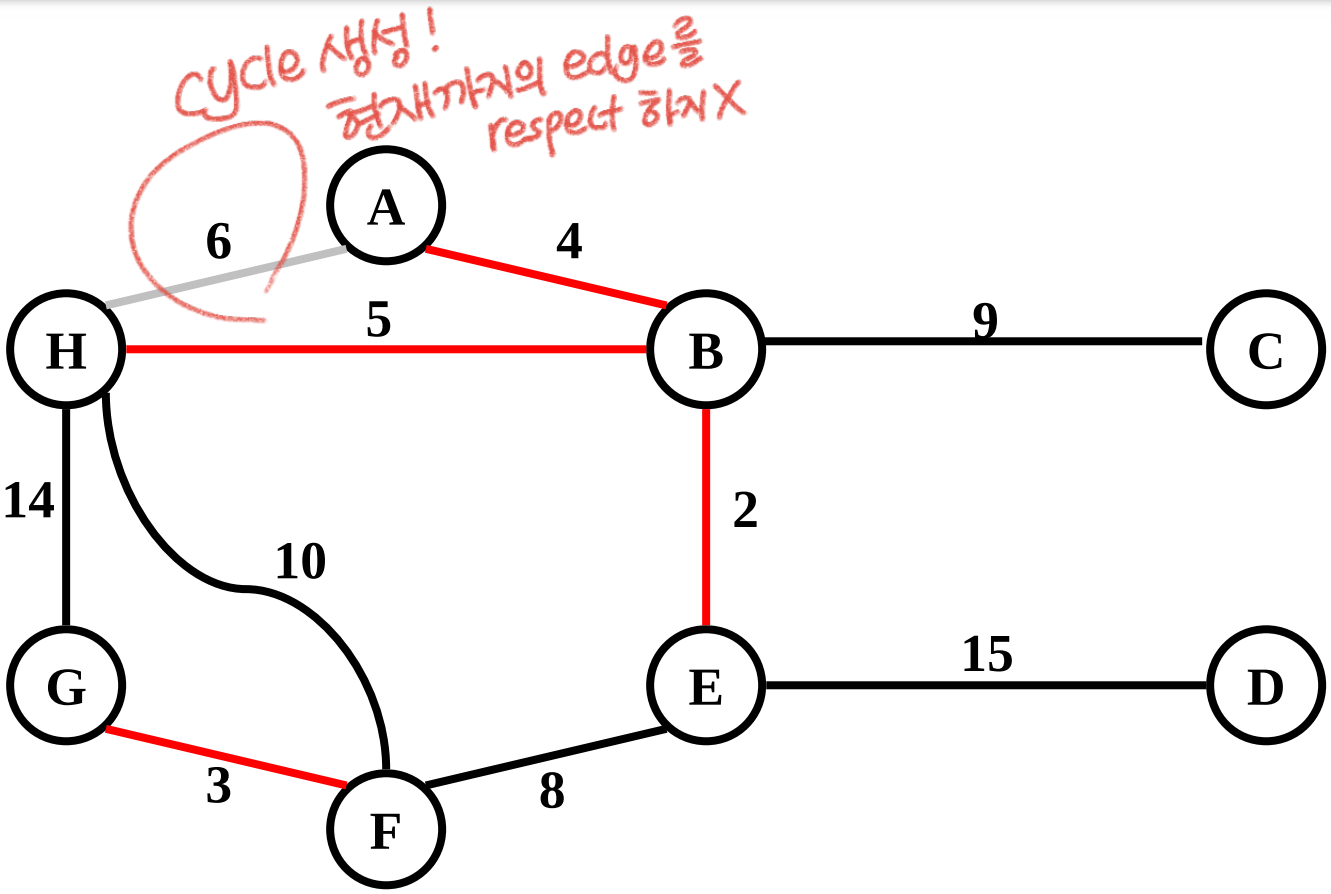
Example of Kruskal



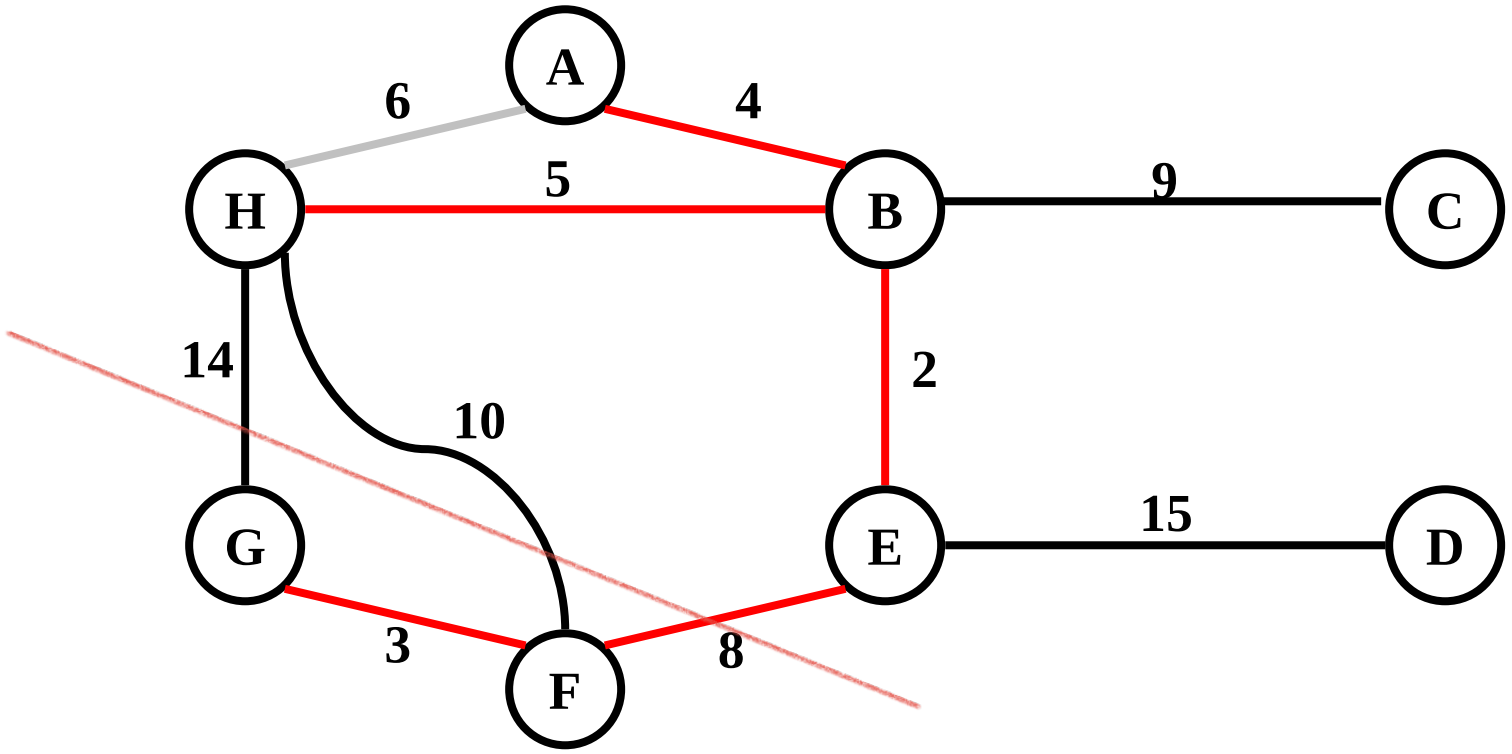
Example of Kruskal



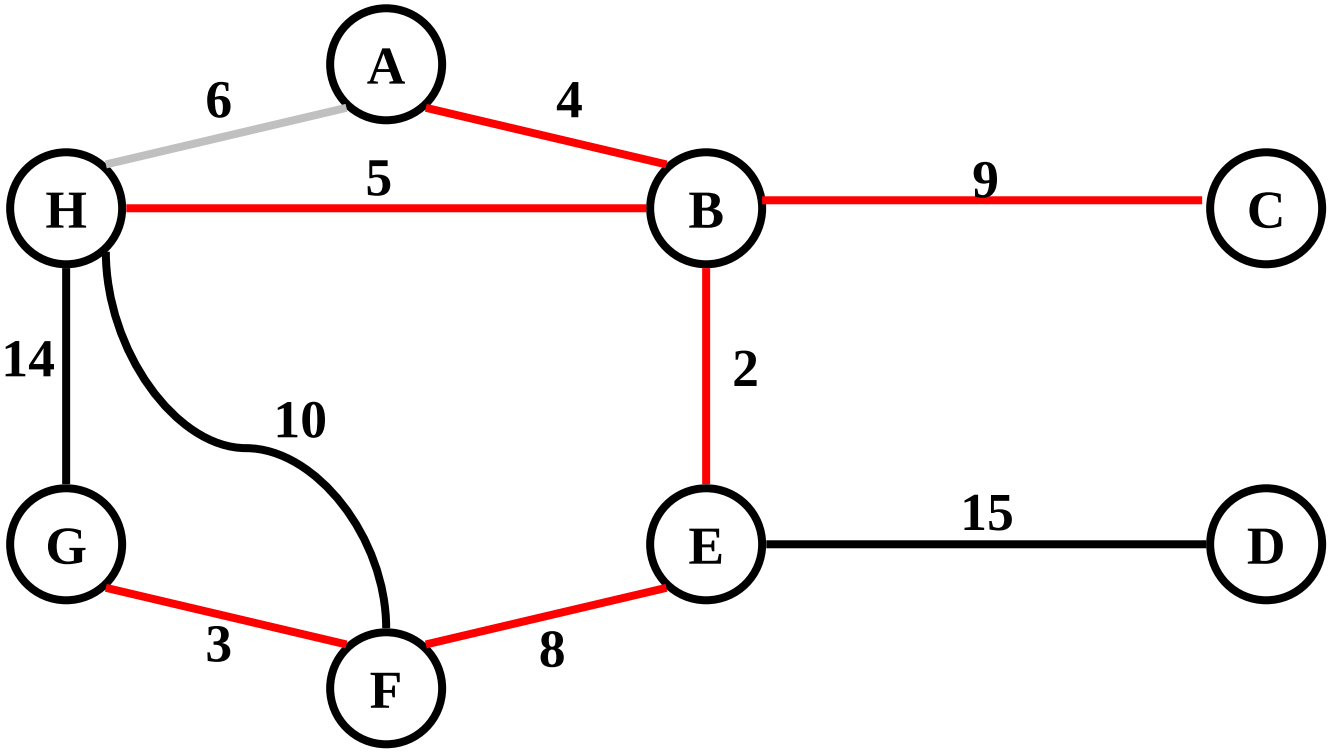
Example of Kruskal



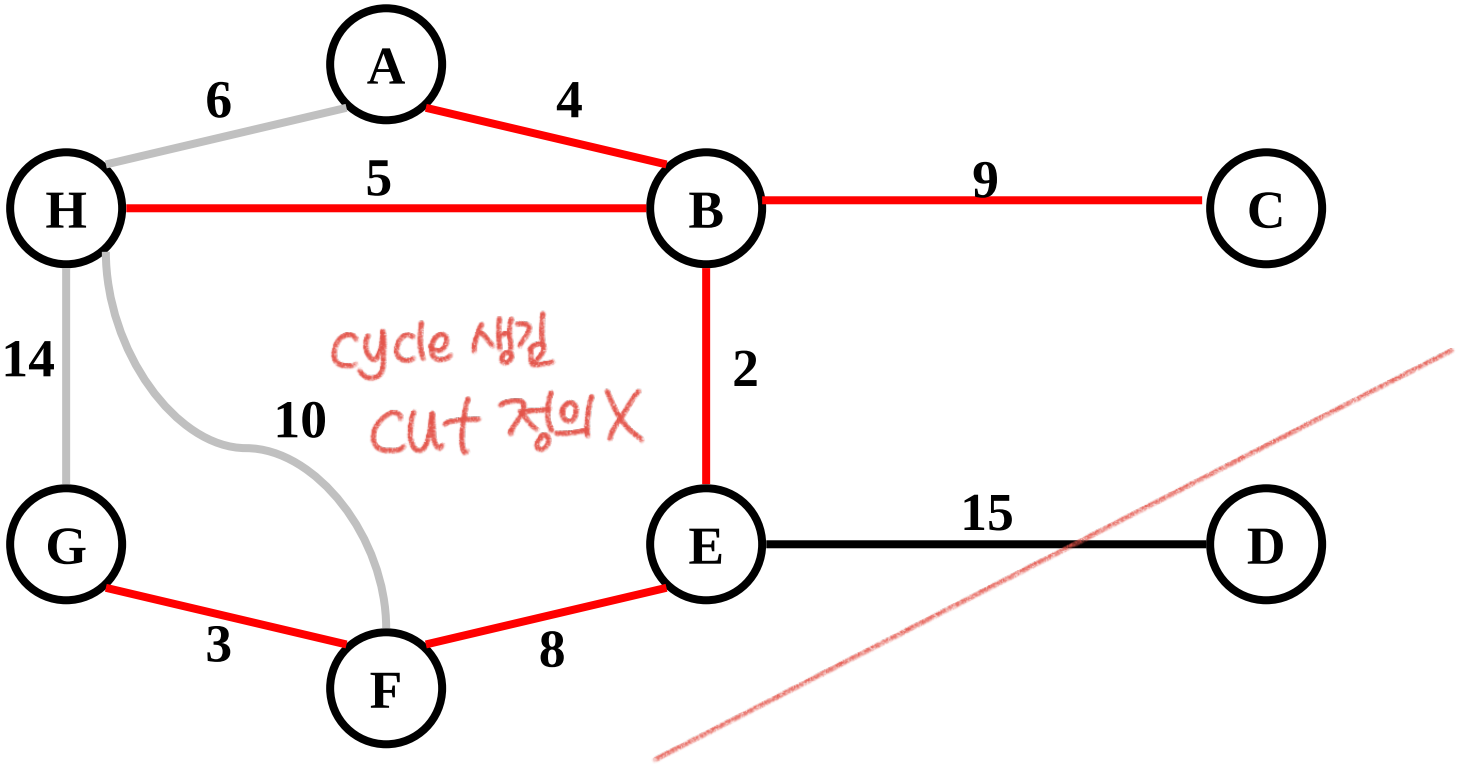
Example of Kruskal



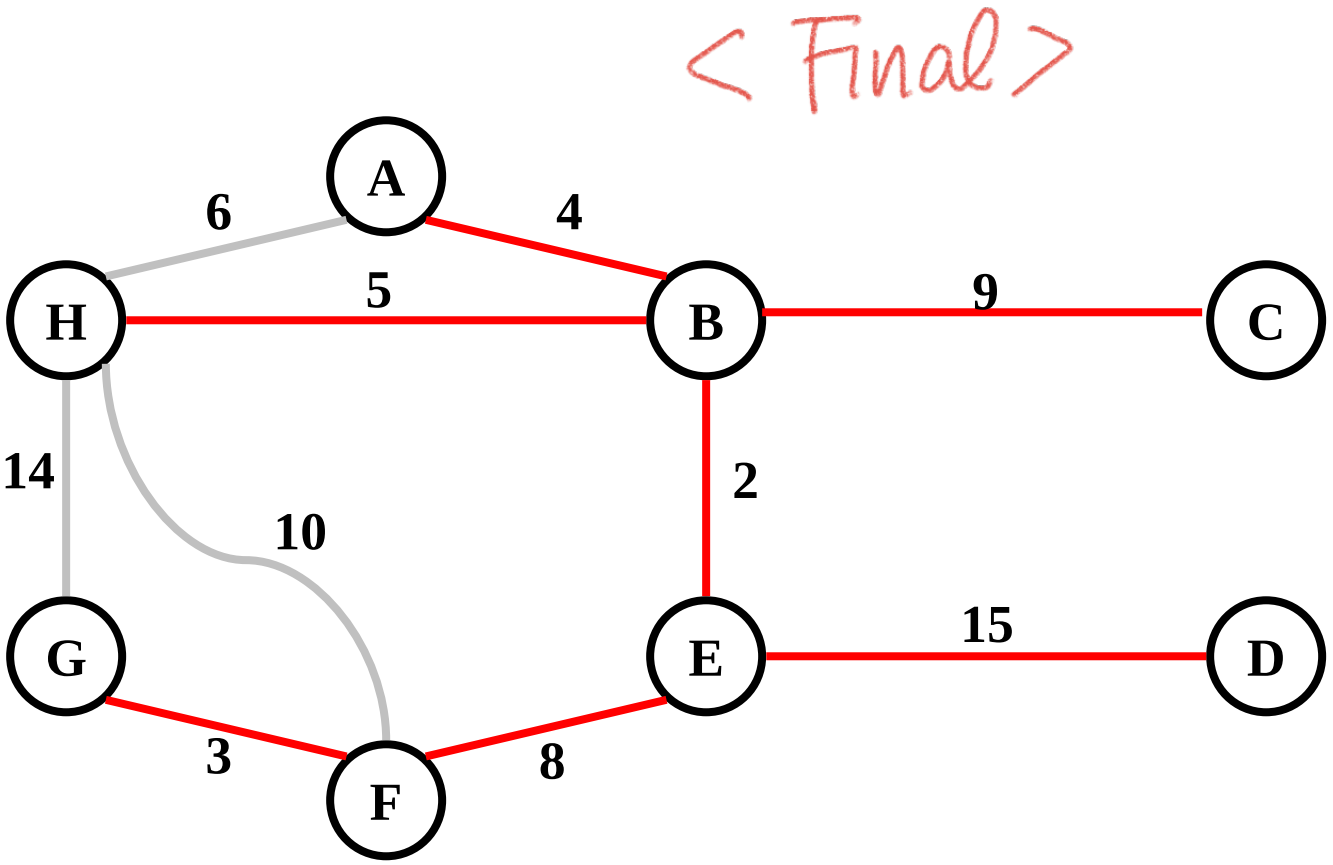
Example of Kruskal



Example of Kruskal



Example of Kruskal



Prim's algorithm

- Builds one tree, so A is always a tree.
- Start from an arbitrary "root" r.
- At each step, find a light edge crossing cut $(V_A, V - V_A)$, where V_A = vertices that A is incident on.
- $\pi[v]$ = parent of v, NIL if it has no parent or $v = r$.
- To find a light edge quickly
  - use a priority queue Q.

Prim's MST: Outline

PrimMST(G,n)

Initialize all vertices as *unseen*.

Select an arbitrary vertex *r* to start the tree; reclassify it as *tree*

Reclassify all vertices adjacent to *r* as *fringe*.

(r과 연결된)

While there are fringe vertices;

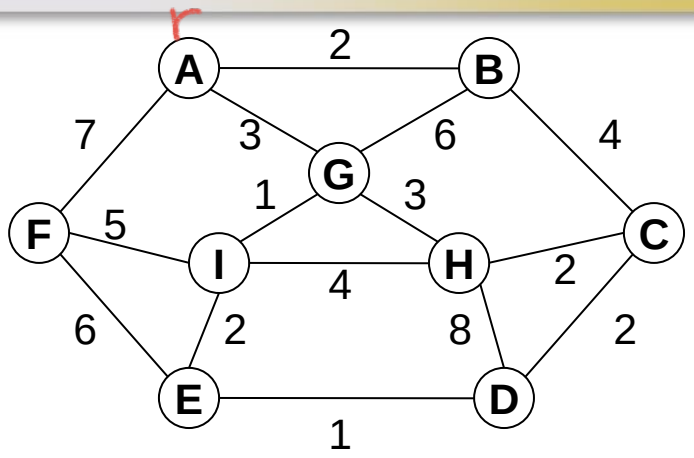
Select an edge of minimum weight between a tree vertex *t* and a fringe vertex *v*;

tree ↔ fringe 간의 out 의, light edge

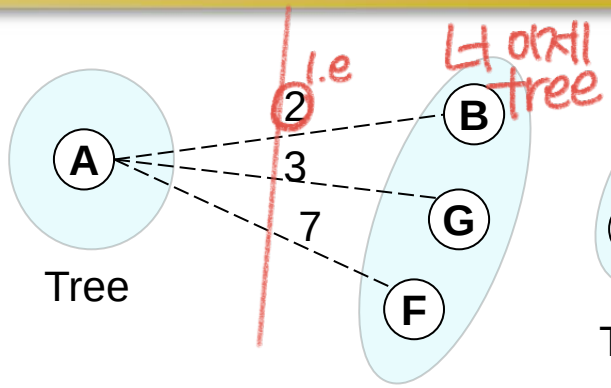
Reclassify *v* as *tree*; add edge (*t,v*) to the tree;

Reclassify all *unseen* vertices adjacent to *v* as *fringe*

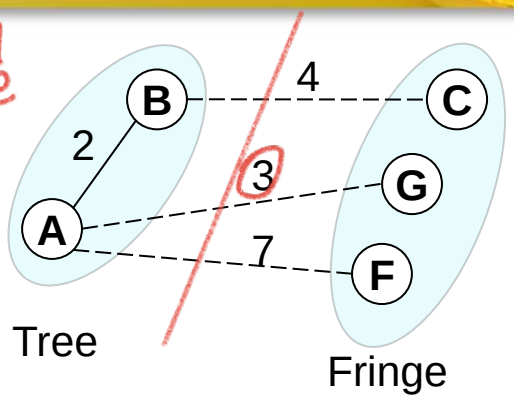
The Algorithm in action, e.g.



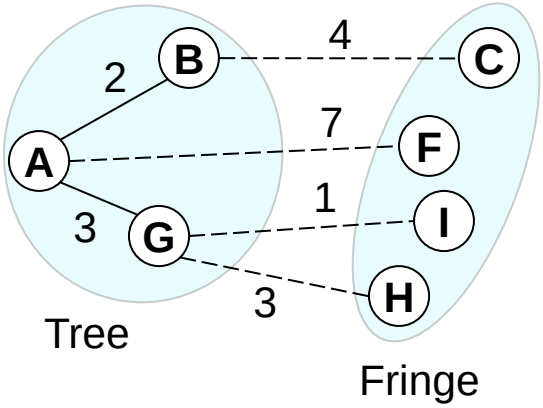
(a) A weighted graph



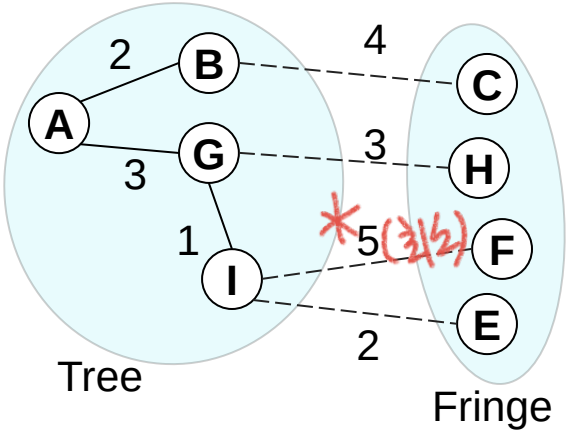
(b) After selection of the starting vertex



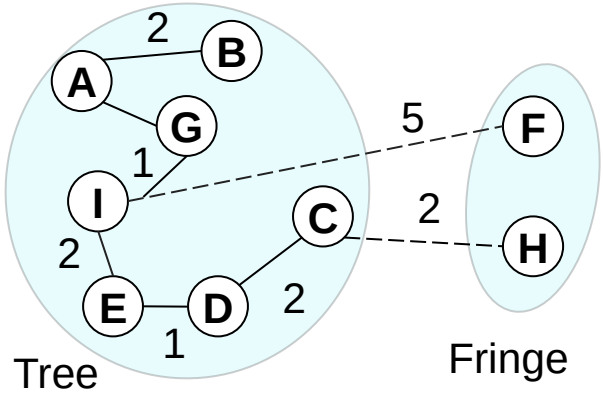
(c) *BG* was considered but did not replace *AG* as a candidate.



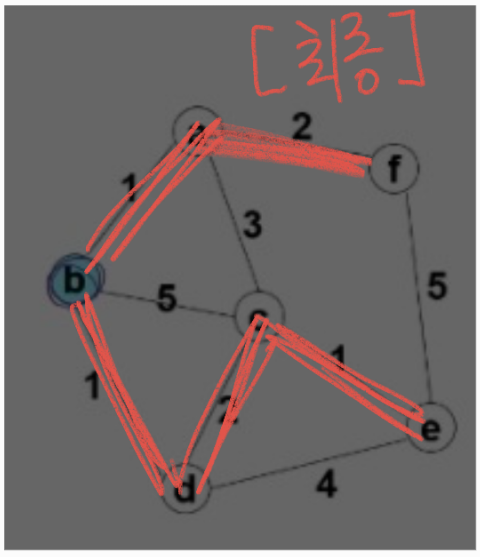
(d) After *A G* is selected and fringe and candidates are updated



(e) I F has replaced A F as a candidate.



(f) After several more passes: The two candidate edges will be put in the tree



b 1 a
  — c
1 d

tie : b 선택 (그냥)

bd — a
  — c
  — e

abd — 2 c ← c, d 중 선택
  — ~~2~~ f
  — e

abcd — f
  — e ← c, e 중 선택

abcde — f

abcdef (끝!)

Prim's Algorithm

MST-Prim(*G*, *w*, *r*)

```
Q = V[G];
for each u ∈ Q
    key[u] = ∞; π[u] = NIL;
key[r] = 0;
π[r] = NULL;
while (Q not empty)
    u = ExtractMin(Q);
    for each v ∈ Adj[u]
        if (v ∈ Q and w(u,v) < key[v])
            π[v] = u;
            key[v] = w(u,v);
```

parent

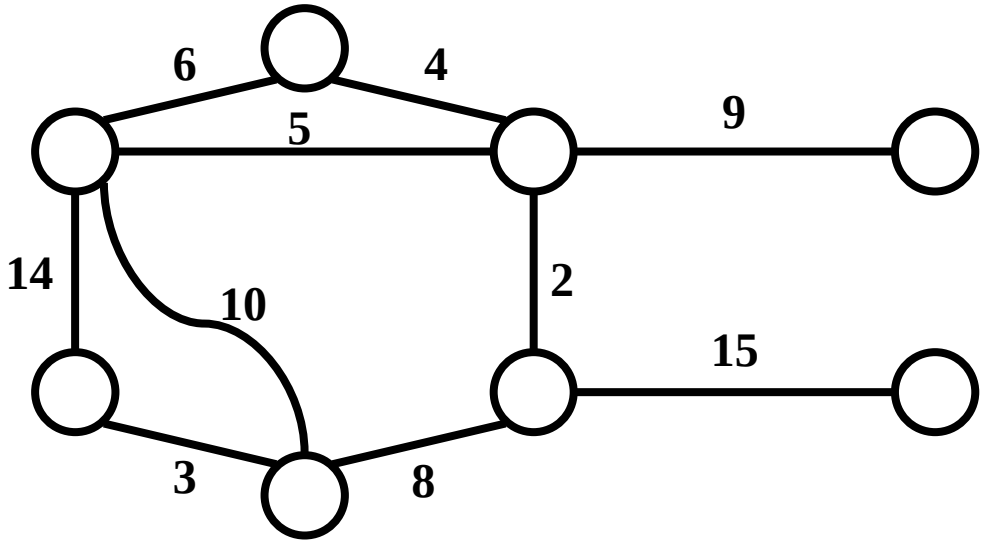
weight 검사

자으면 포함

Prim's Algorithm

MST-Prim(*G*, *w*, *r*)

```
Q = V[G];
for each u ∈ Q
    key[u] = ∞; π[u] = NIL;
key[r] = 0;
π[r] = NULL;
while (Q not empty)
    u = ExtractMin(Q);
    for each v ∈ Adj[u]
        if (v ∈ Q and w(u,v) < key[v])
            π[v] = u;
            key[v] = w(u,v);
```

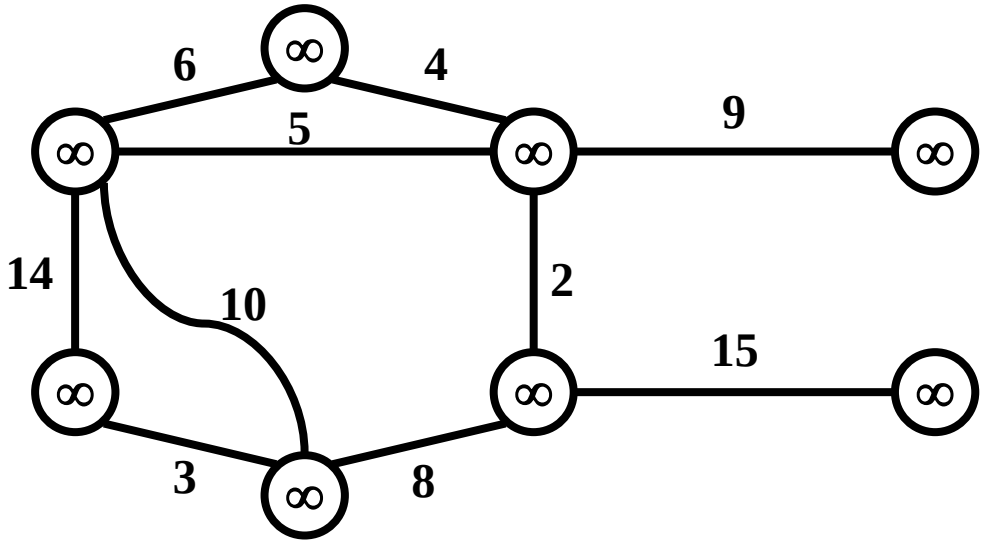


Run on example graph

Prim's Algorithm

MST-Prim(*G*, *w*, *r*)

```
Q = V[G];
for each u ∈ Q
    key[u] = ∞; π[u] = NIL;
key[r] = 0;
π[r] = NULL;
while (Q not empty)
    u = ExtractMin(Q);
    for each v ∈ Adj[u]
        if (v ∈ Q and w(u,v) < key[v])
            π[v] = u;
            key[v] = w(u,v);
```

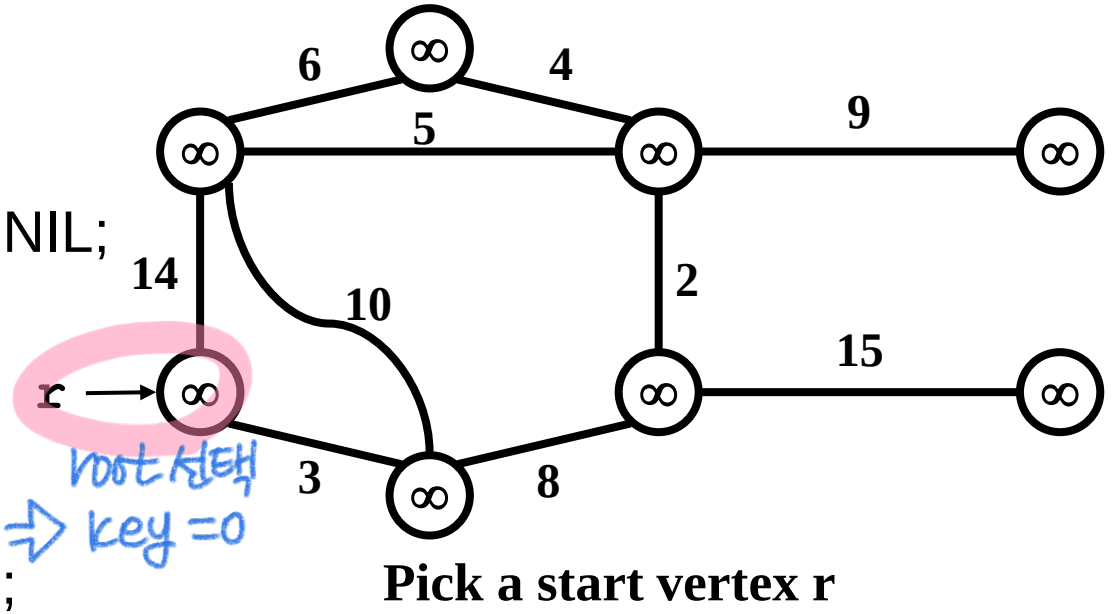


Run on example graph

Prim's Algorithm

MST-Prim(*G*, *w*, *r*)

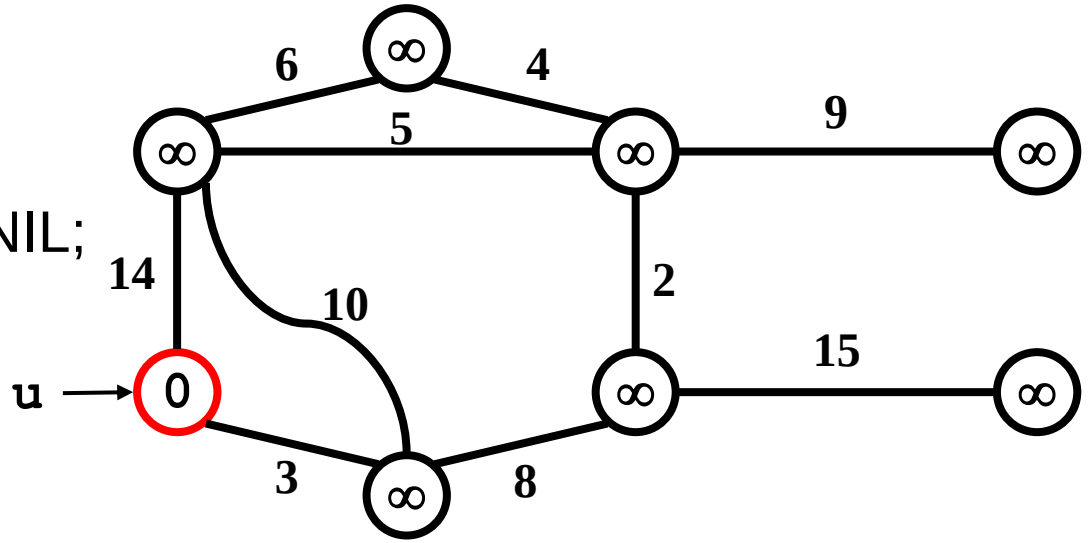
```
Q = V[G];
for each u ∈ Q
    key[u] = ∞; π[u] = NIL;
key[r] = 0;
π[r] = NULL;
while (Q not empty)
    u = ExtractMin(Q);
    for each v ∈ Adj[u]
        if (v ∈ Q and w(u,v) < key[v])
            π[v] = u;
            key[v] = w(u,v);
```



Prim's Algorithm

MST-Prim(*G*, *w*, *r*)

```
Q = V[G];
for each u ∈ Q
    key[u] = ∞; π[u] = NIL;
key[r] = 0;
π[r] = NULL;
while (Q not empty)
    u = ExtractMin(Q);
    for each v ∈ Adj[u]
        if (v ∈ Q and w(u,v) < key[v])
            π[v] = u;
            key[v] = w(u,v);
```

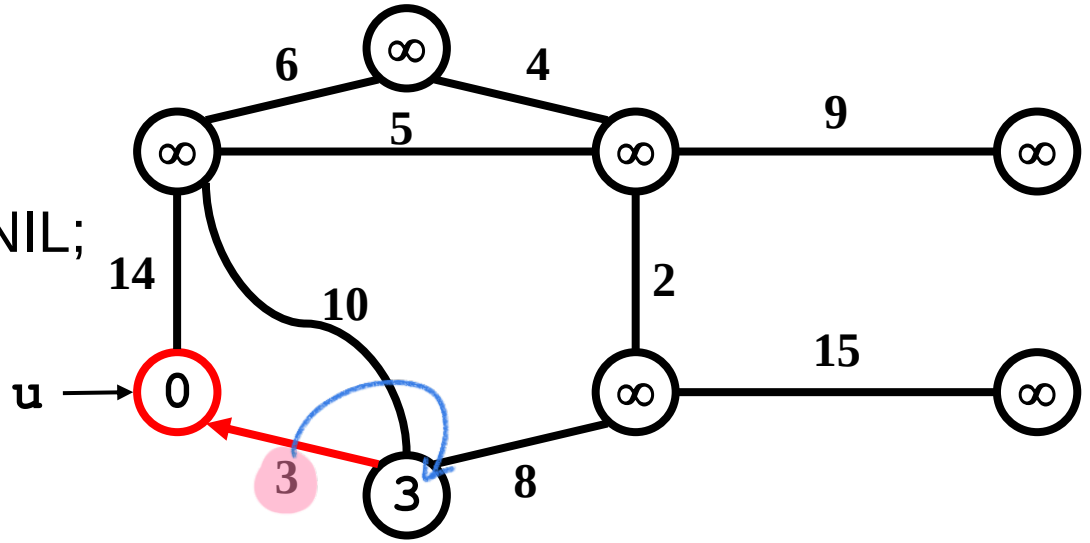


Red vertices have been removed from Q

Prim's Algorithm

MST-Prim(*G*, *w*, *r*)

```
Q = V[G];
for each u ∈ Q
    key[u] = ∞; π[u] = NIL;
key[r] = 0;
π[r] = NULL;
while (Q not empty)
    u = ExtractMin(Q);
    for each v ∈ Adj[u]
        if (v ∈ Q and w(u,v) < key[v])
            π[v] = u;
            key[v] = w(u,v);
```

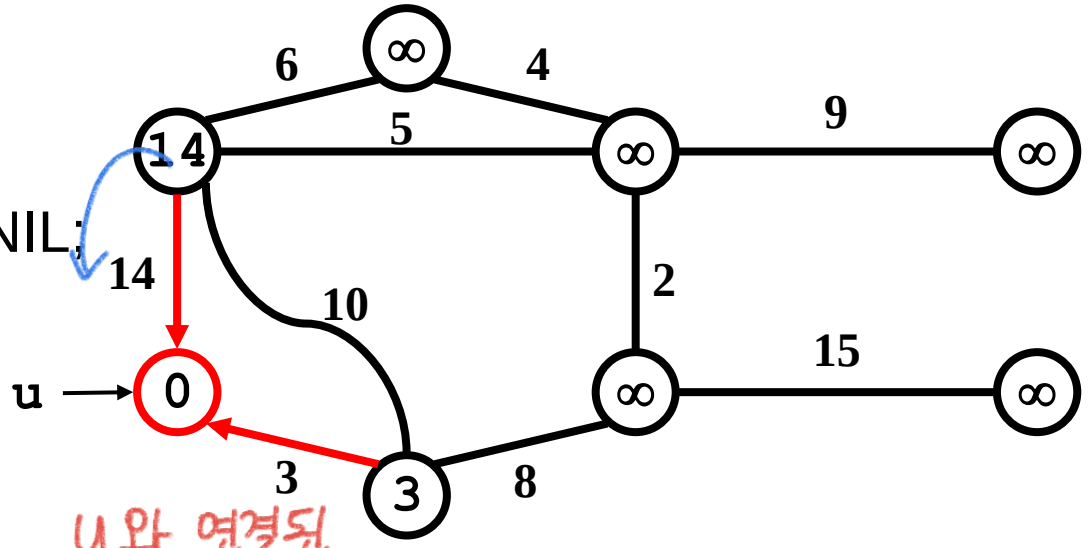


Red arrows indicate parent pointers

Prim's Algorithm

MST-Prim(*G*, *w*, *r*)

```
Q = V[G];
for each u ∈ Q
    key[u] = ∞; π[u] = NIL;
key[r] = 0;
π[r] = NULL;
while (Q not empty)
    u = ExtractMin(Q);
    for each v ∈ Adj[u]
        if (v ∈ Q and w(u,v) < key[v])
            π[v] = u;
            key[v] = w(u,v);
```



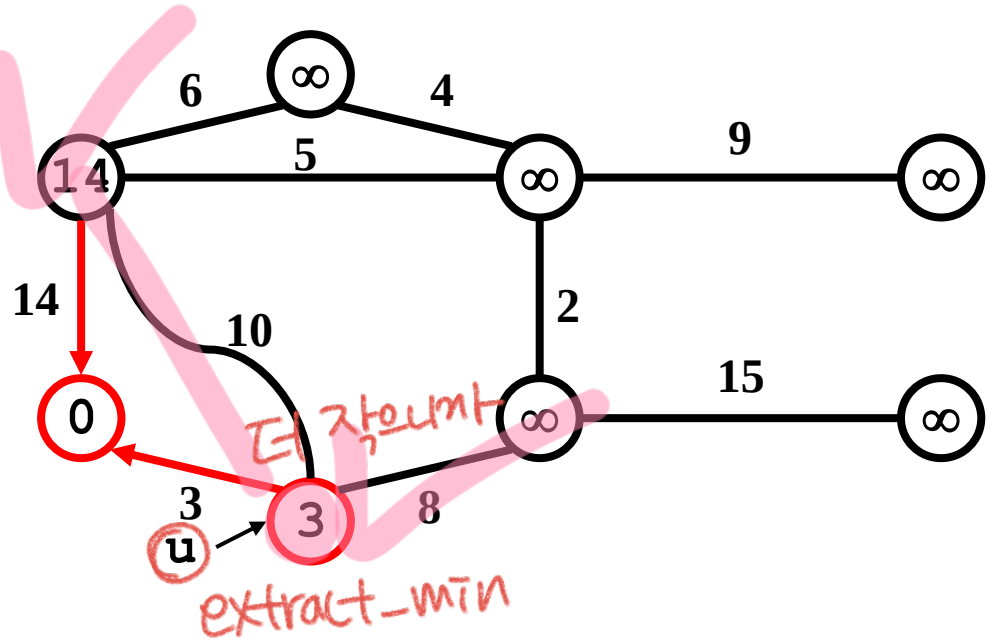
u와 연결된 node 들의 값(key)을 weight 으로 설정 (현재 key 보다 작으면)

ex) ∞ > 14

Prim's Algorithm

MST-Prim(*G*, *w*, *r*)

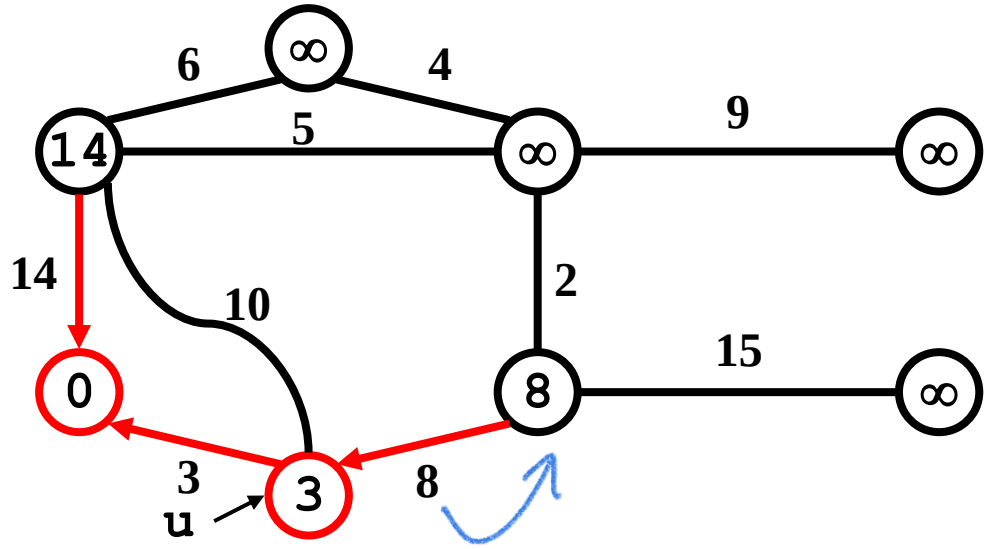
```
Q = V[G];
for each u ∈ Q
    key[u] = ∞; π[u] = NIL;
key[r] = 0;
π[r] = NULL;
while (Q not empty)
    u = ExtractMin(Q);
    for each v ∈ Adj[u]
        if (v ∈ Q and w(u,v) < key[v])
            π[v] = u;
            key[v] = w(u,v);
```



Prim's Algorithm

MST-Prim(*G*, *w*, *r*)

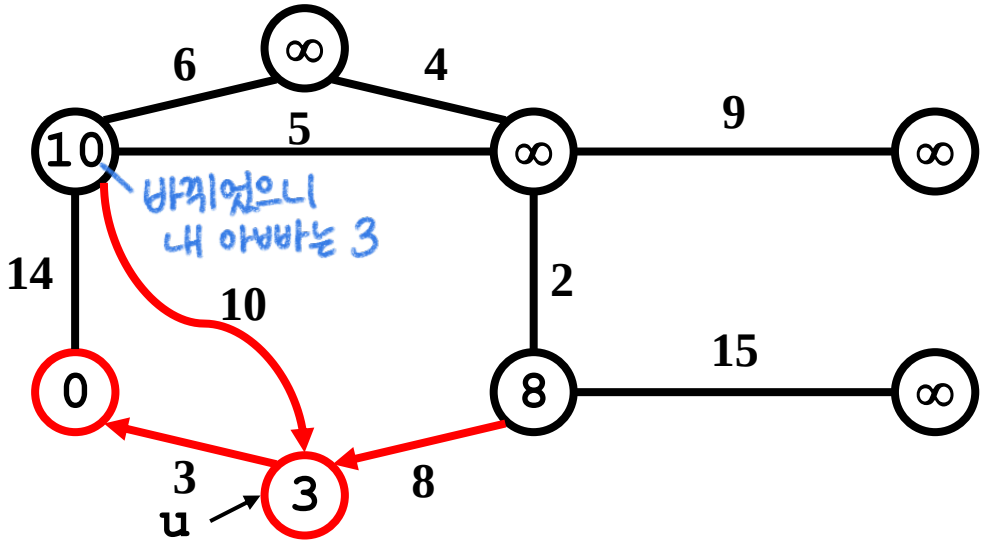
```
Q = V[G];
for each u ∈ Q
    key[u] = ∞; π[u] = NIL;
key[r] = 0;
π[r] = NULL;
while (Q not empty)
    u = ExtractMin(Q);
    for each v ∈ Adj[u]
        if (v ∈ Q and w(u,v) < key[v])
            π[v] = u;
            key[v] = w(u,v);
```



Prim's Algorithm

MST-Prim(*G*, *w*, *r*)

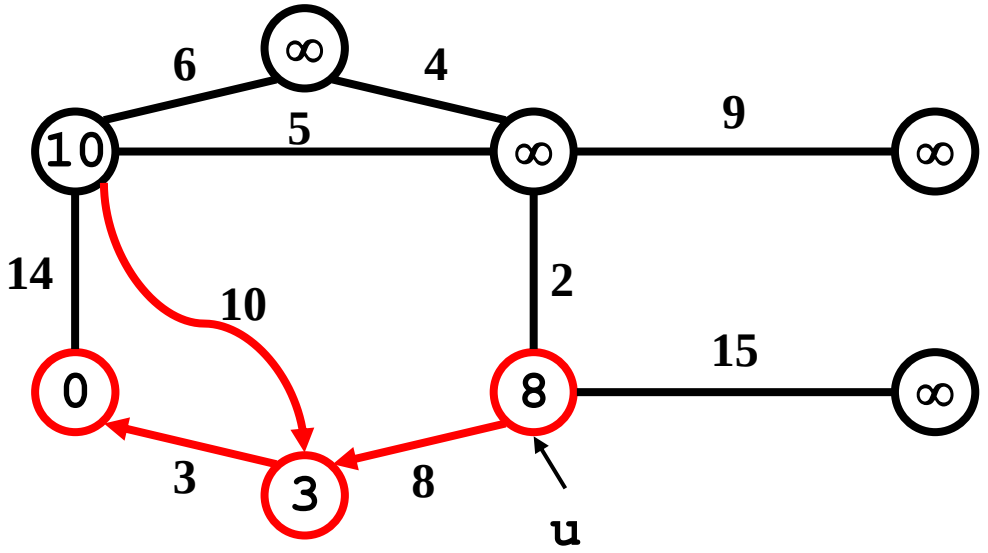
```
Q = V[G];
for each u ∈ Q
    key[u] = ∞; π[u] = NIL;
key[r] = 0;
π[r] = NULL;
while (Q not empty)
    u = ExtractMin(Q);
    for each v ∈ Adj[u]
        if (v ∈ Q and w(u,v) < key[v])
            π[v] = u;
            key[v] = w(u,v);
```



Prim's Algorithm

MST-Prim(*G*, *w*, *r*)

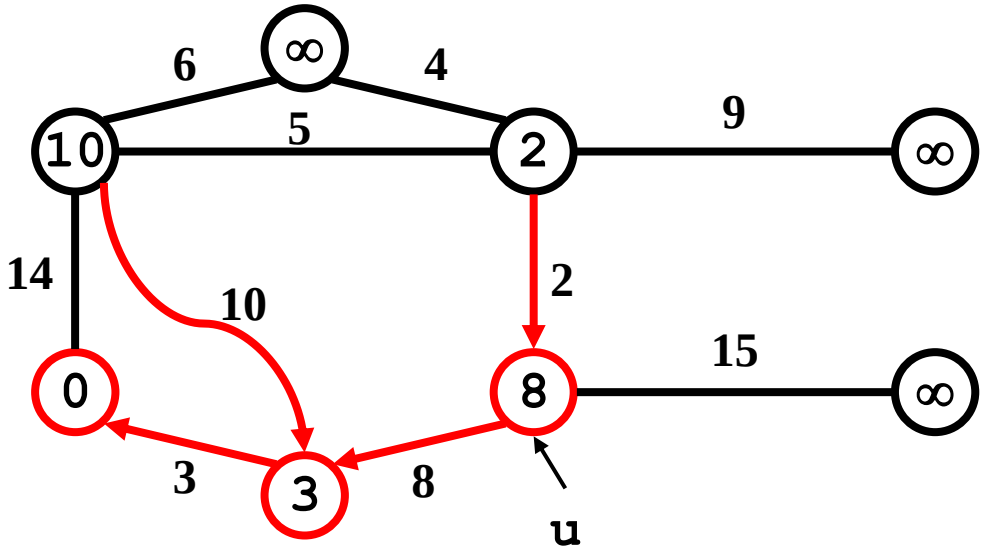
```
Q = V[G];
for each u ∈ Q
    key[u] = ∞; π[u] = NIL;
key[r] = 0;
π[r] = NULL;
while (Q not empty)
    u = ExtractMin(Q);
    for each v ∈ Adj[u]
        if (v ∈ Q and w(u,v) < key[v])
            π[v] = u;
            key[v] = w(u,v);
```



Prim's Algorithm

MST-Prim(*G*, *w*, *r*)

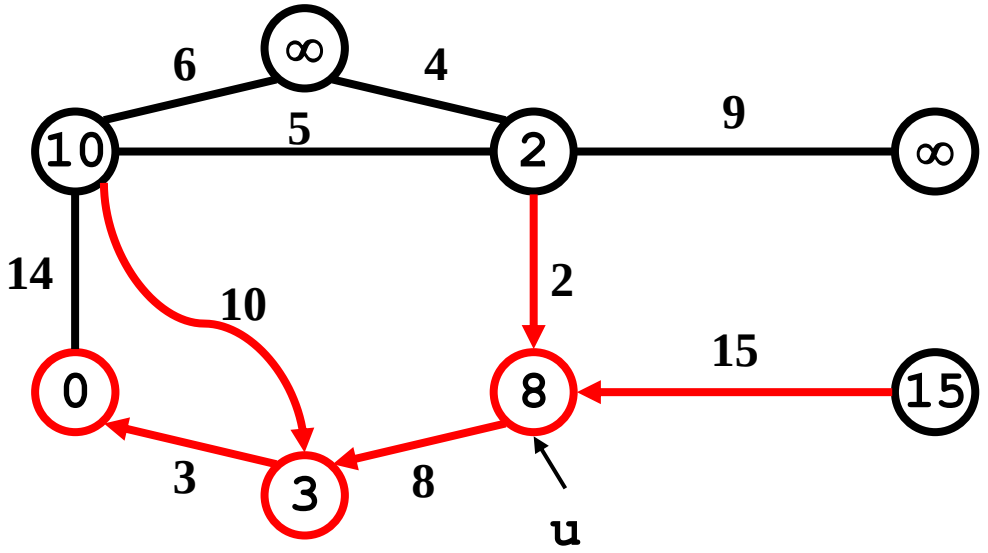
```
Q = V[G];
for each u ∈ Q
    key[u] = ∞; π[u] = NIL;
key[r] = 0;
π[r] = NULL;
while (Q not empty)
    u = ExtractMin(Q);
    for each v ∈ Adj[u]
        if (v ∈ Q and w(u,v) < key[v])
            π[v] = u;
            key[v] = w(u,v);
```



Prim's Algorithm

MST-Prim(*G*, *w*, *r*)

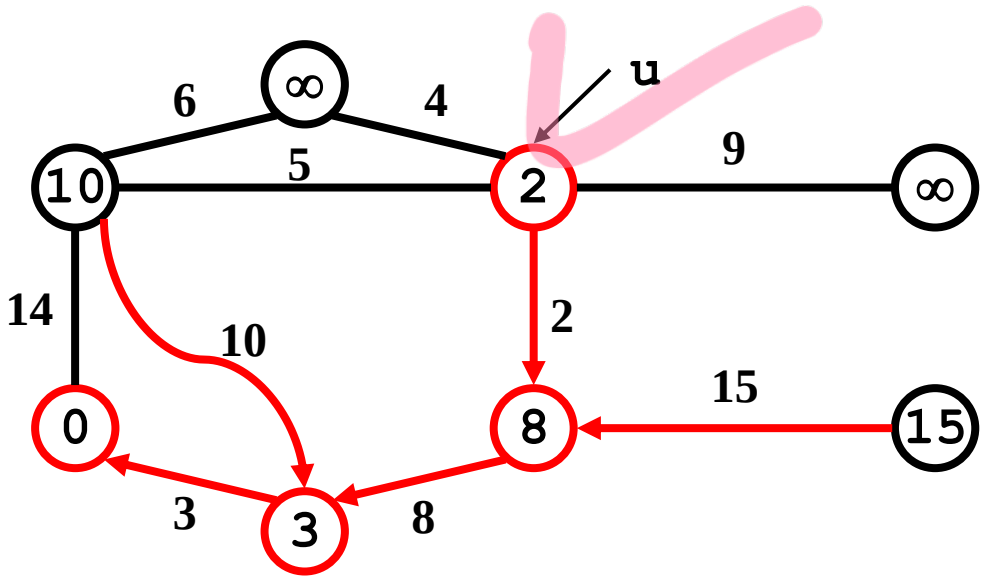
```
Q = V[G];
for each u ∈ Q
    key[u] = ∞; π[u] = NIL;
key[r] = 0;
π[r] = NULL;
while (Q not empty)
    u = ExtractMin(Q);
    for each v ∈ Adj[u]
        if (v ∈ Q and w(u,v) < key[v])
            π[v] = u;
            key[v] = w(u,v);
```



Prim's Algorithm

MST-Prim(*G*, *w*, *r*)

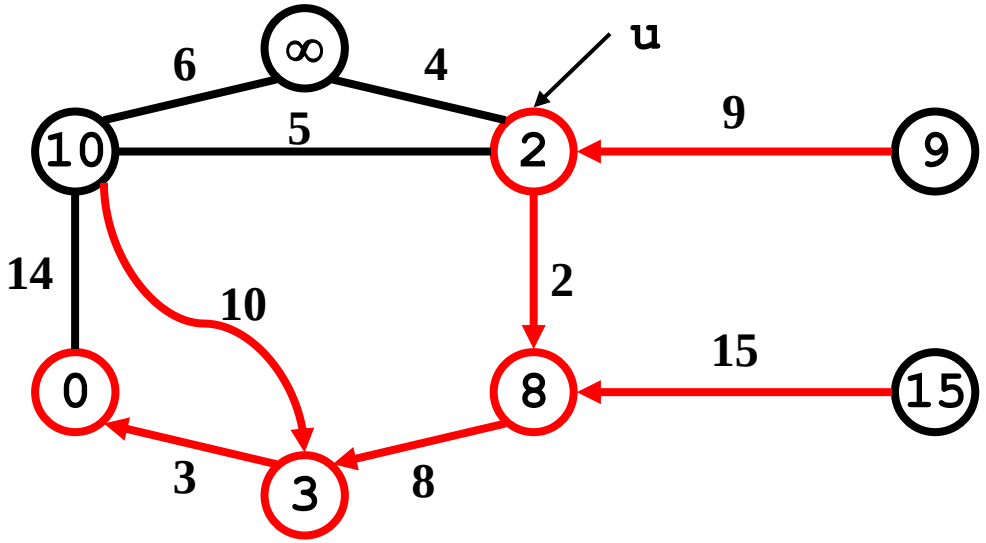
```
Q = V[G];
for each u ∈ Q
    key[u] = ∞; π[u] = NIL;
key[r] = 0;
π[r] = NULL;
while (Q not empty)
    u = ExtractMin(Q);
    for each v ∈ Adj[u]
        if (v ∈ Q and w(u,v) < key[v])
            π[v] = u;
            key[v] = w(u,v);
```



Prim's Algorithm

MST-Prim(*G*, *w*, *r*)

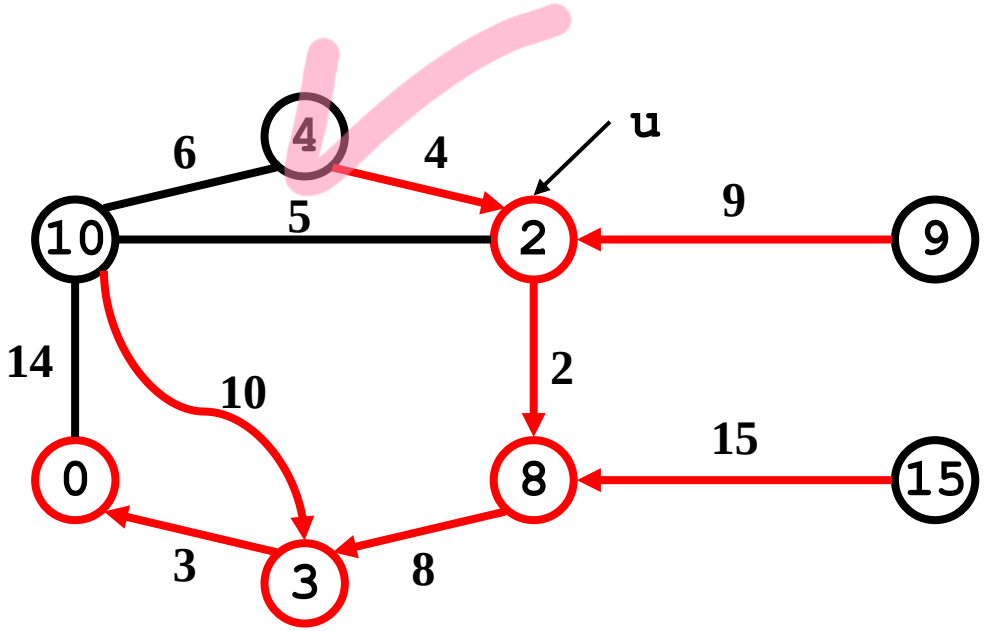
```
Q = V[G];
for each u ∈ Q
    key[u] = ∞; π[u] = NIL;
key[r] = 0;
π[r] = NULL;
while (Q not empty)
    u = ExtractMin(Q);
    for each v ∈ Adj[u]
        if (v ∈ Q and w(u,v) < key[v])
            π[v] = u;
            key[v] = w(u,v);
```



Prim's Algorithm

MST-Prim(*G*, *w*, *r*)

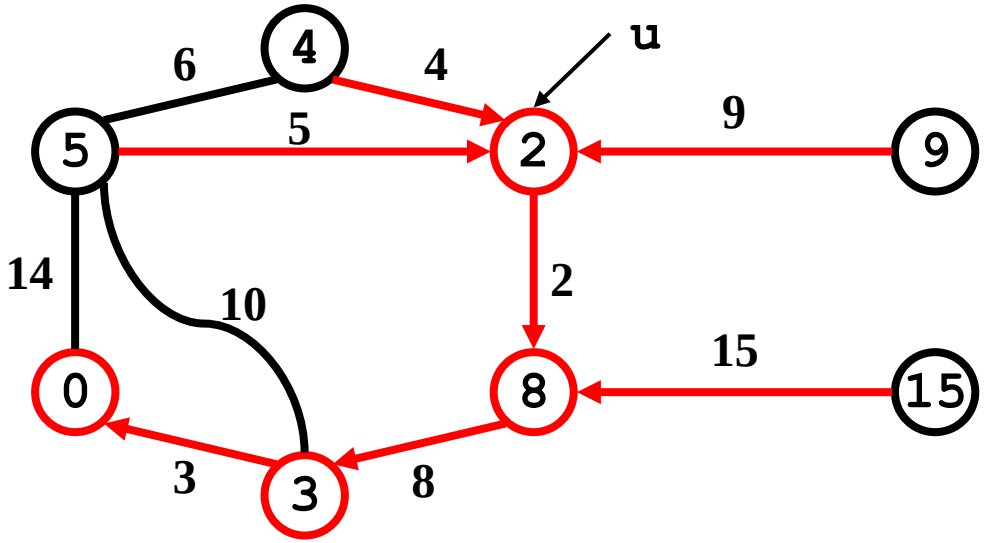
```
Q = V[G];
for each u ∈ Q
    key[u] = ∞; π[u] = NIL;
key[r] = 0;
π[r] = NULL;
while (Q not empty)
    u = ExtractMin(Q);
    for each v ∈ Adj[u]
        if (v ∈ Q and w(u,v) < key[v])
            π[v] = u;
            key[v] = w(u,v);
```



Prim's Algorithm

MST-Prim(*G*, *w*, *r*)

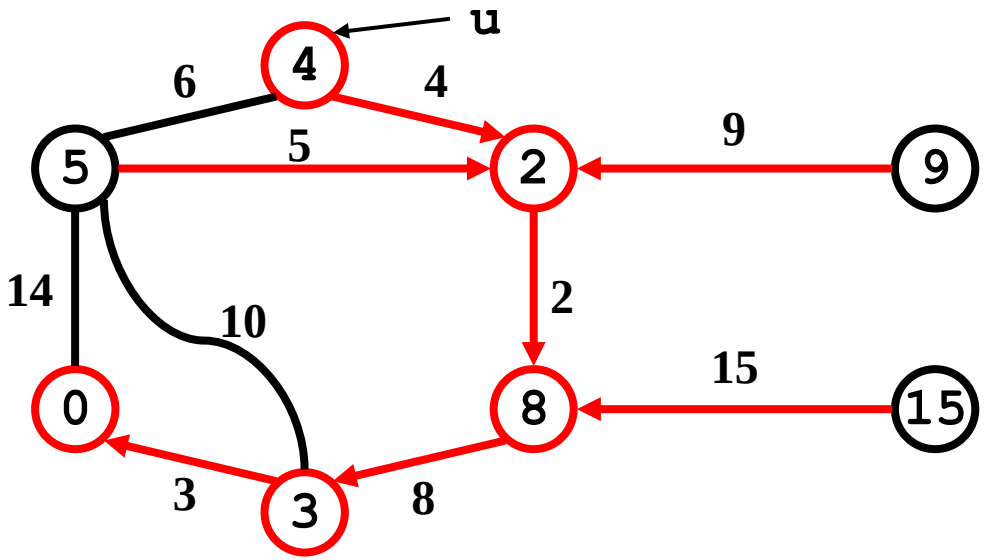
```
Q = V[G];
for each u ∈ Q
    key[u] = ∞; π[u] = NIL;
key[r] = 0;
π[r] = NULL;
while (Q not empty)
    u = ExtractMin(Q);
    for each v ∈ Adj[u]
        if (v ∈ Q and w(u,v) < key[v])
            π[v] = u;
            key[v] = w(u,v);
```



Prim's Algorithm

MST-Prim(*G*, *w*, *r*)

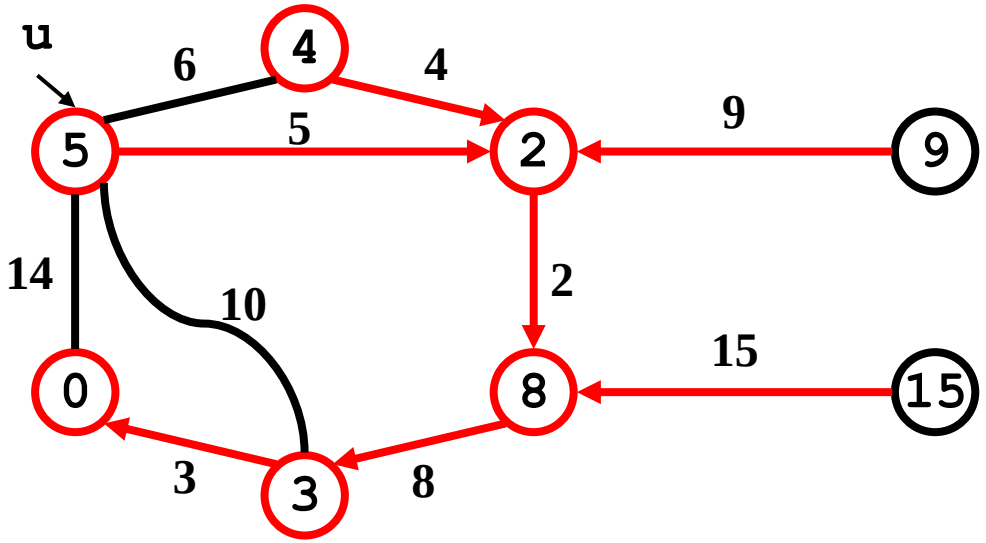
```
Q = V[G];
for each u ∈ Q
    key[u] = ∞; π[u] = NIL;
key[r] = 0;
π[r] = NULL;
while (Q not empty)
    u = ExtractMin(Q);
    for each v ∈ Adj[u]
        if (v ∈ Q and w(u,v) < key[v])
            π[v] = u;
            key[v] = w(u,v);
```



Prim's Algorithm

MST-Prim(*G*, *w*, *r*)

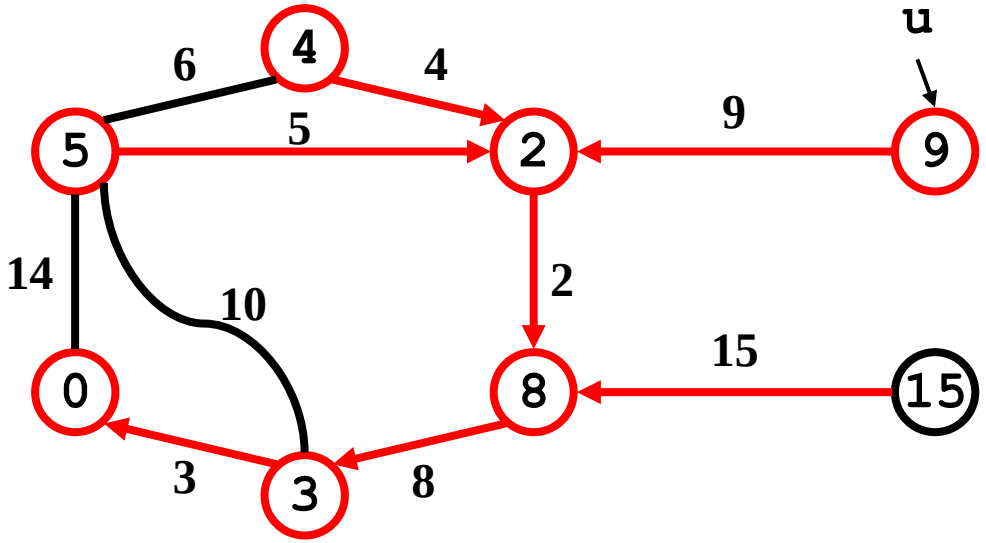
```
Q = V[G];
for each u ∈ Q
    key[u] = ∞; π[u] = NIL;
key[r] = 0;
π[r] = NULL;
while (Q not empty)
    u = ExtractMin(Q);
    for each v ∈ Adj[u]
        if (v ∈ Q and w(u,v) < key[v])
            π[v] = u;
            key[v] = w(u,v);
```



Prim's Algorithm

MST-Prim(*G*, *w*, *r*)

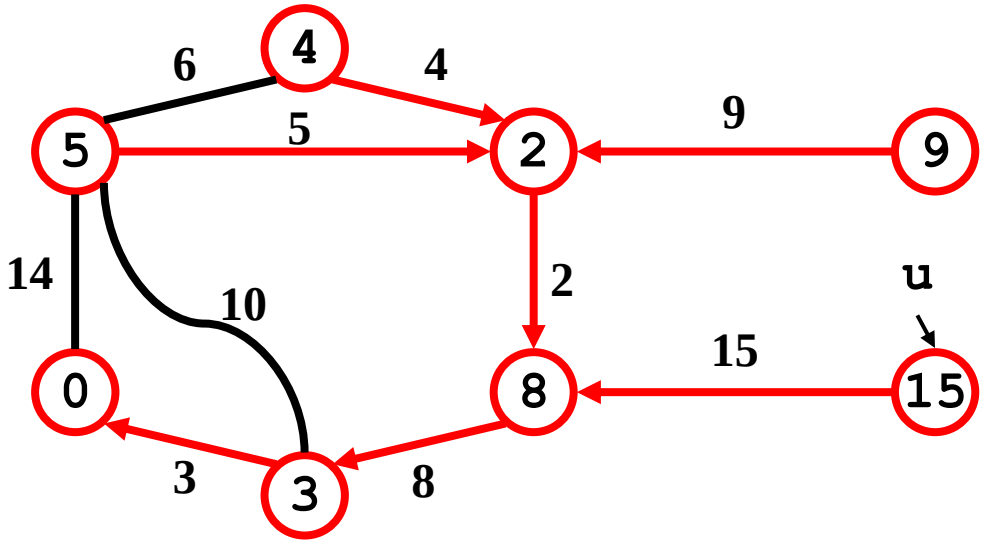
```
Q = V[G];
for each u ∈ Q
    key[u] = ∞; π[u] = NIL;
key[r] = 0;
π[r] = NULL;
while (Q not empty)
    u = ExtractMin(Q);
    for each v ∈ Adj[u]
        if (v ∈ Q and w(u,v) < key[v])
            π[v] = u;
            key[v] = w(u,v);
```



Prim's Algorithm

MST-Prim(*G*, *w*, *r*)

```
Q = V[G];
for each u ∈ Q
    key[u] = ∞; π[u] = NIL;
key[r] = 0;
π[r] = NULL;
while (Q not empty)
    u = ExtractMin(Q);
    for each v ∈ Adj[u]
        if (v ∈ Q and w(u,v) < key[v])
            π[v] = u;
            key[v] = w(u,v);
```



[QUIZ]
CH 22, 23, 24
13주차 화요일

(Exercise 2) Prim's algorithm의 time complexity
항상 하나의 tree가 유지되고,
edge들이 추가됨

✓ 1-3번째 줄 $\Theta(V \lg V)$ or $\Theta(V)$
Q_Queue에 V개의 node 넣는 것 (첫줄)

✓ while loop의 실행 ; vertex 갯수만큼, $\Theta(V)$ (횟수 |V|)) $O(V \lg V)$
u = ExtractMin(Q) : $O(\lg V)$
(5번 line)

✓ 6 - for loop : edge 갯수, $\Theta(E)$) $\Theta(E)$
1
8 $\Theta(\lg V)$

모든 edge 들에 대해 key가 restruct 되어야 함 : $\Theta(\lg V)$

total running time) $O(V \log V) + E \log V = O(E \log V)$

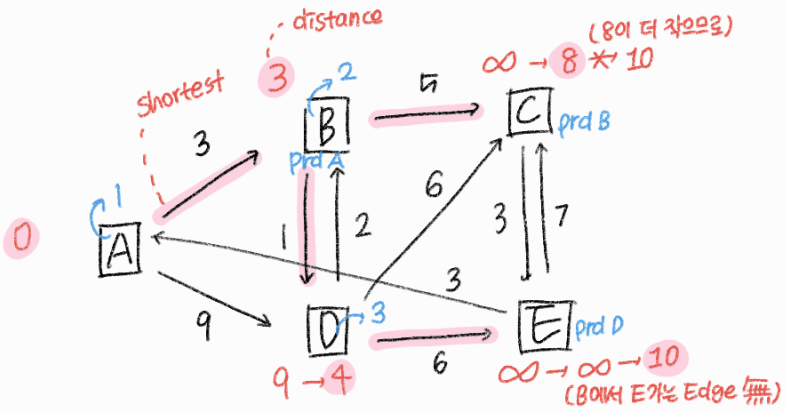
(Exercise 3) Prim, kruskal의 design strategy

=> greedy (light한 edge)

Shortest Path

(Exercise 1) Run Dijkstra's algorithm. A is source.
After D is extracted from p-queue and all edges incident from D is relaxed, distance of B,C,E 구하라.

n.e, cycle 여부에 따라 적절히 골라야 한다



* data structure : p-queue
design strategy : greedy
$\Theta(VE+V^2 \lg V) = \Theta(V^3)$

(cf) Bellman
$\Theta(V^2E) = \Theta(V^4)$
negative edge 有

(Exercise 2) Floyd Warshall algorithm
Show the last matrix D

* recursive solution 가지고 구할 것

D^0 = (vertex간 direct edge)

$$\begin{bmatrix} 0 & \infty & \infty & \infty & -1 & \infty \\ 1 & 0 & \infty & 2 & \cancel{\infty}\,10 & \infty \\ \infty & 2 & 0 & \infty & \infty & -8 \\ -4 & \infty & \infty & 0 & \cancel{3}\,-5 & \infty \\ \infty & -7 & \infty & \infty & 0 & \infty \\ \infty & \square & 10 & \infty & \infty & 0 \end{bmatrix}$$

(V2→V6)
V2→V1→V5

D^1 : 1번 node를 거쳐서 가는

⋮

⇓

D^6 까지 완성

* design strategy : DP
$\Theta(V^3)$, 메모리 ↑
⊕ 구현이 쉽다. simple
(data 개수 적을 때 용이)
negative edge도 deal 가능.
(Dijkstra 는 불가)

HANDONG GLOBAL UNIVERSITY

Chapter 24, 25
Shortest Paths

Algorithm Analysis

School of CSEE

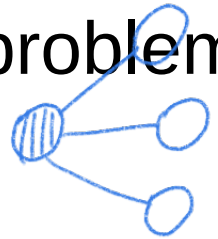
Shortest path

- Input : directed graph $G = (V, E)$ with weight function $w : E \rightarrow R$
- Determine the path p with minimum weight from source to destination.
- Weight $w(p)$ of path p : sum of edge weights on path p
- *shortest-path weight u to v :*

$$\delta(u,v) = \begin{cases} \min \{ w(p) : \text{path } p \text{ from } u \text{ to } v \} \\ \qquad \text{if there exists a path from } u \text{ to } v. \\ \infty, \text{ otherwise.} \end{cases}$$

Variants

- Single-source shortest path : find shortest paths from a given source vertex $s \in V$ to every vertex $v \in V$
- Single-destinations : find shortest paths to a given destination vertex. By reversing the direction of the each edge in the graph, the problem is reduced to single-source problem.



- Single-pair : find shortest path from u to v. No easier than single-source problem.

- All-pairs shortest-paths : find shortest path from u to v for all u, $v \in V$ 각 node가 전부 source

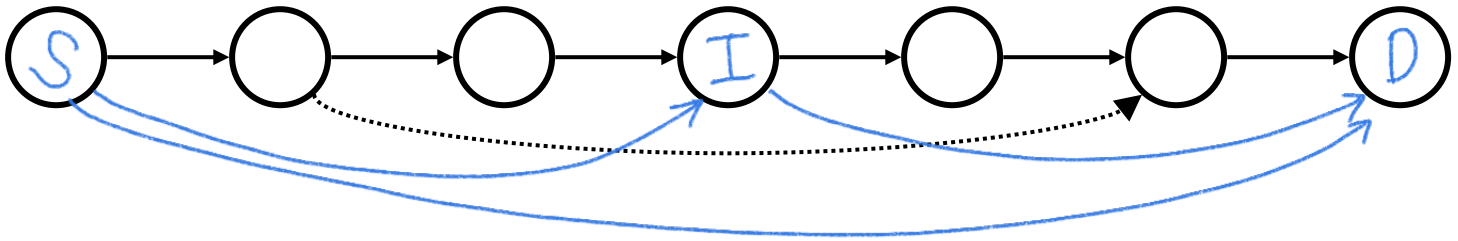
Variants

- Single-source shortest path
  - Bellman-Ford algorithm
  - In DAG
  - Dijkstra's algorithm

- All-pairs shortest-paths
  - Floyd-Warshall algorithm

Shortest Path Properties : Lemma 24.1

- Again, we have *optimal substructure*: the shortest path consists of shortest subpaths:



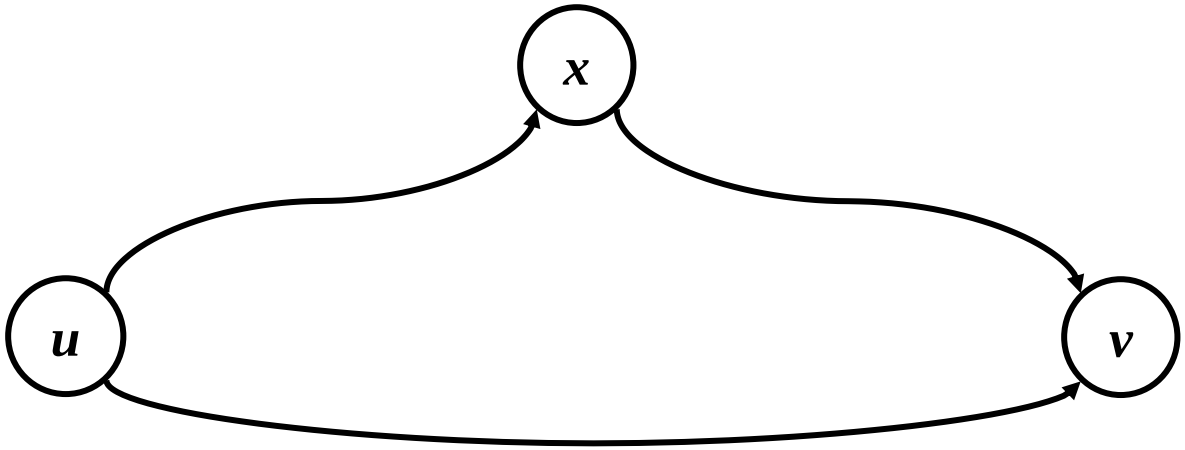
  - Proof: suppose some subpath is not a shortest path.

Proof

- There must then exist a shorter subpath.
- We could substitute the shorter subpath for a shortest path.
- But then overall path is not shortest path. Contradiction!!

Shortest Path Properties

- Define $\delta(u,v)$ to be the weight of the shortest path from u to v. Shortest paths satisfy the *triangle inequality*: $\delta(u,v) \leq \delta(u,x) + \delta(x,v)$



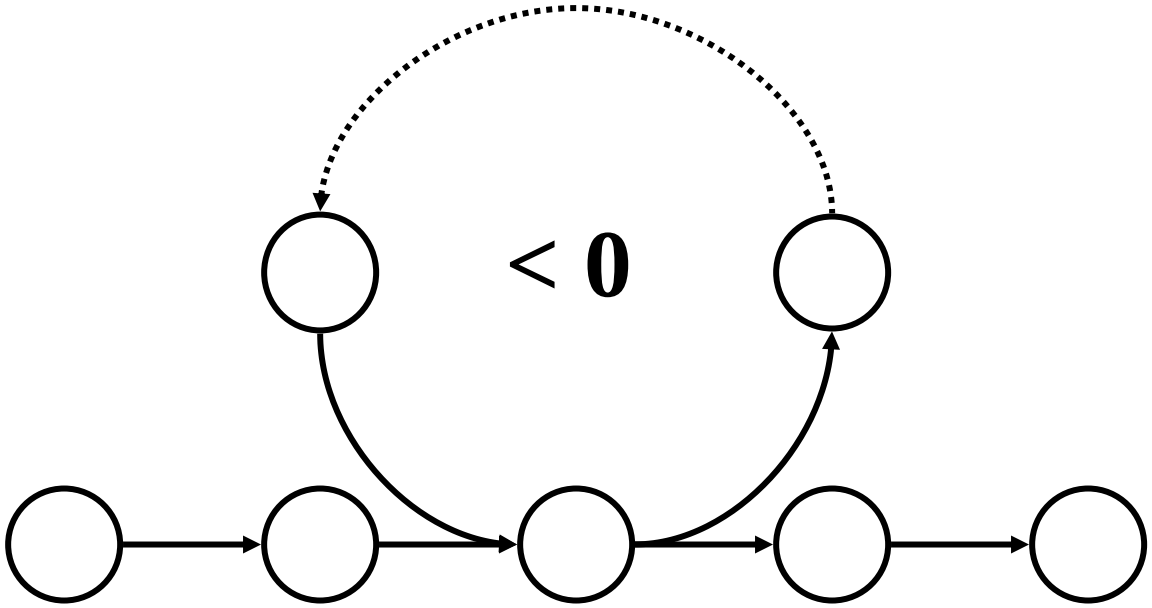
This path is no longer than any other path

Negative-weight edges

- OK, as long as no negative-weight cycle are reachable from the source.

거쳐가지 않으면 OK

- If we reach a negative-weight cycle, we can just keep going around it and get $w(s, v) = -\infty$ for all v in the cycle.
- Some algorithms work only if there are no negative-weight edges.

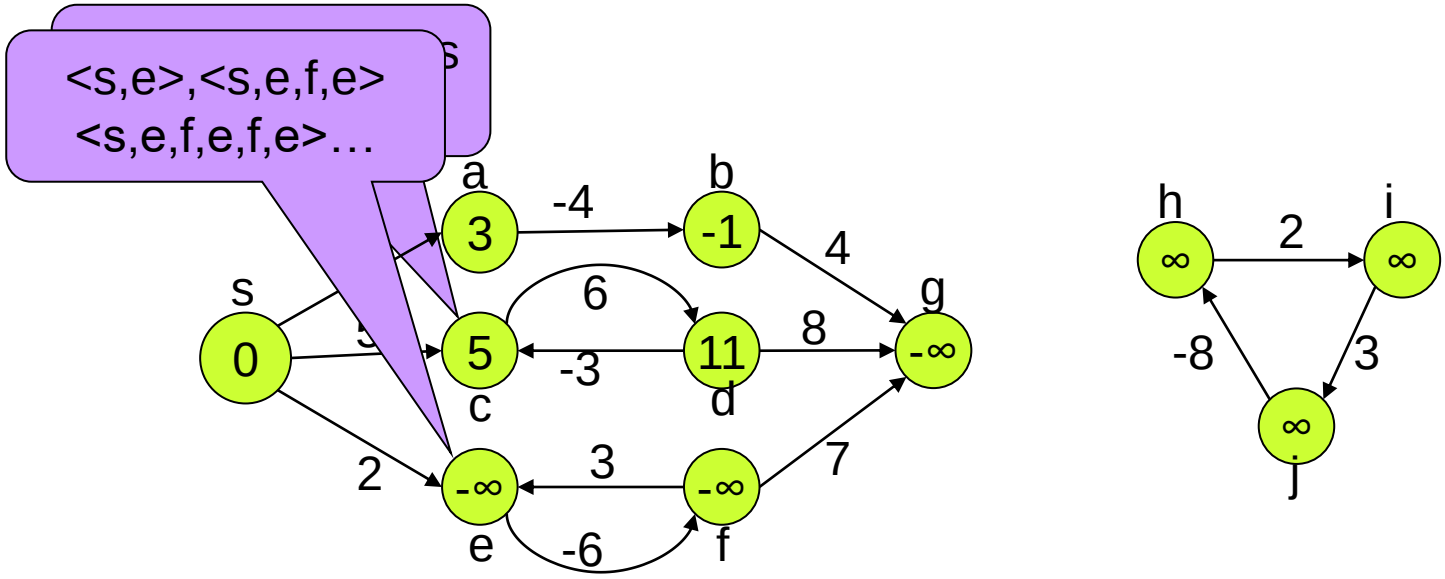


Cycles

- Shortest paths can't contain cycles.
  - We assumed that there is no negative-weight cycles.
  - Positive-weight cycle : Just omit it to get a shorter path.
  - Zero-weight cycle : There is no reason to use them. Assume that our solutions won't use them.

Cycles

- If there is a cycle reachable from s, shortest-path weights are not well defined.
- $\delta(s,v) = -\infty$ ($v \in V$)



Single Source Shortest Path

SS shortest-path algorithm

- Find shortest paths from a given source vertex $s \in V$ to every vertex $v \in V$.
- Output for each vertex $v \in V$:
  - $d[v] = \delta(s,v)$ distance
    - Initially, $d[v] = \infty$
    - Reduces $d[v]$ as algorithm progress. But always maintain $d[v] \geq \delta(s,v)$
    - $d[v]$: *shortest-path estimate*
  - $\pi[v]$ = predecessor of v on a shorteat path from s.
    - If no predecessor, $\pi[v]$ = NIL
    - π induces a tree – *shortest-path tree*

Initialization

INIT-SINGLE-SOURCE(V, s)

for each $v \in V$

do $d[v] = \infty$ distance

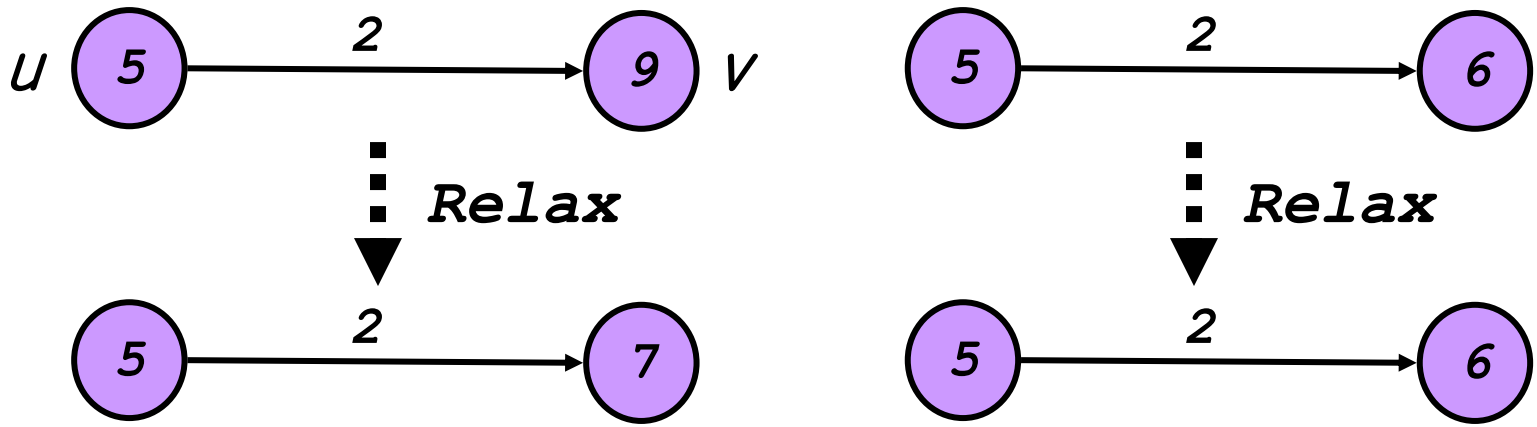
$\pi[v] =$ NIL predecessor

$d[s] = 0$

Relaxation

- A key technique in shortest path algorithms is *relaxation*.
  - Idea: for all v, maintain upper bound $d[v]$ on $\delta(s,v)$

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



[1] Bellman-Ford algorithm

- Solves the single-source shortest-paths problem in general case in which edge weights may be negative.
- Allow negative-weight edges and produce a correct answer as long as no negative-weight cycles are reachable from the source.
- Returns a boolean value
  - negative-weight cycle – FALSE
  - No such cycle – TRUE

Pseudocode

```
BELLMAN-FORD(V, E, w, s)
1.  INIT-SINGLE-SOURCE(V, s)
2.  for i = 1 to |V| - 1
3.      do for each edge (u,v) ∈ E
4.          do RELAX(u, v, w)
5.  for each edge (u,v) ∈ E
6.      do if d[v] > d[u] + w(u,v)
7.          then return FALSE
8.  return TRUE
```

Initialize d[], which will converge to shortest-path value δ

Relaxation: Make |V|-1 passes, relaxing each edge

Test for solution Under what condition do we get a solution?

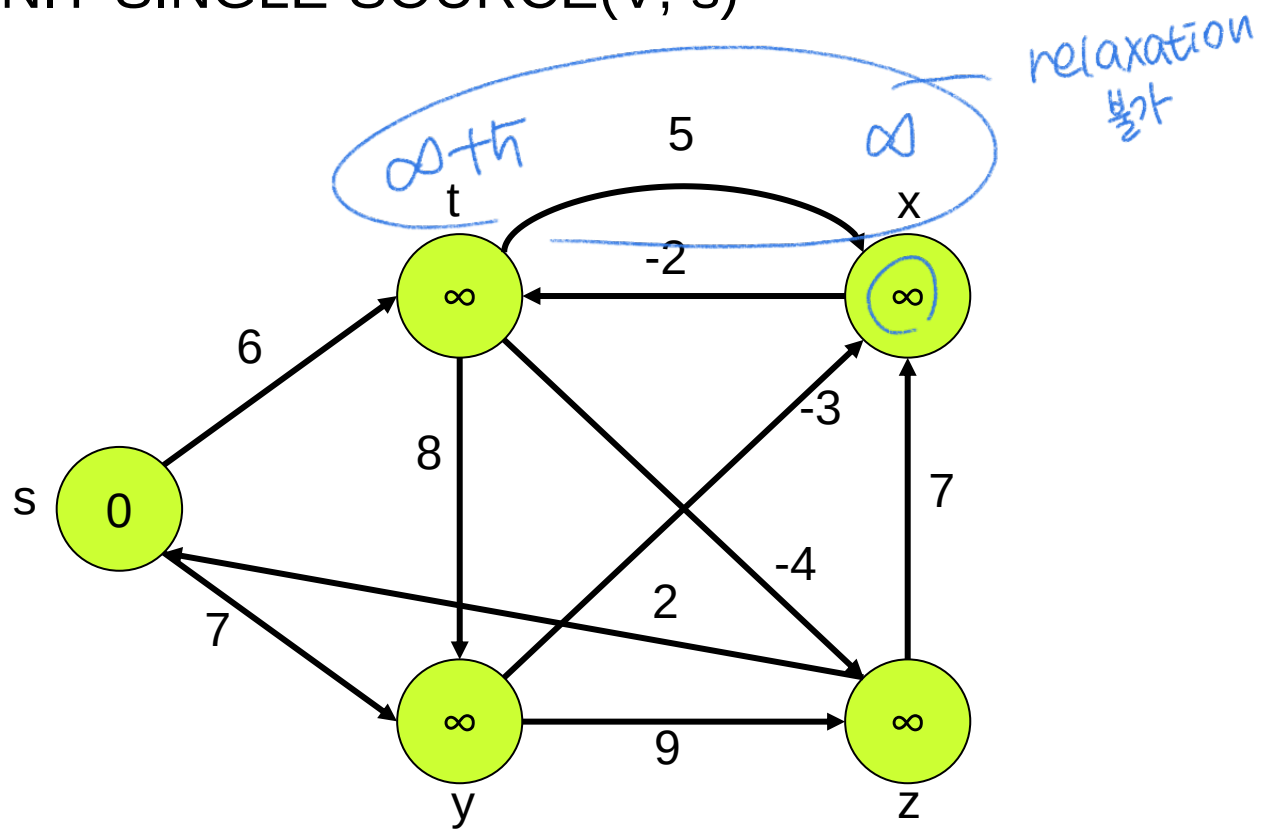
The first for loop relaxes all edges $|V| - 1$ times.

Time : $\Theta(VE)$

Example

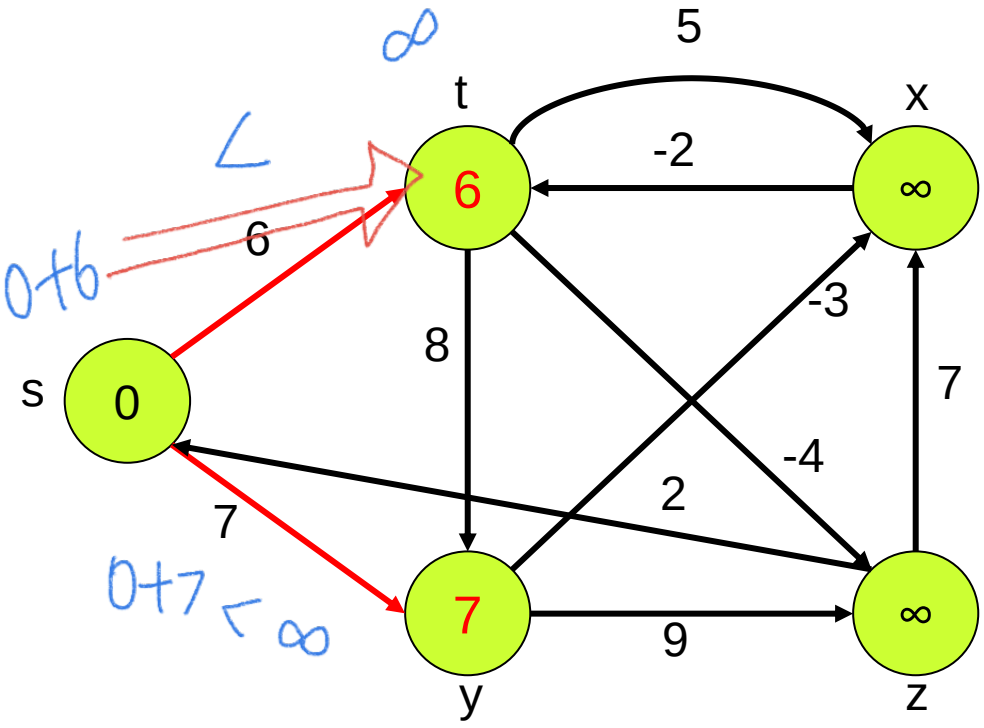
relax 순서 (무관)

- Order : (t,x), (t,y), (t,z), (x,t), (y,x), (y,z), (z,x), (z,s), (s,t), (s,y)
- After INIT-SINGLE-SOURCE(V, s)



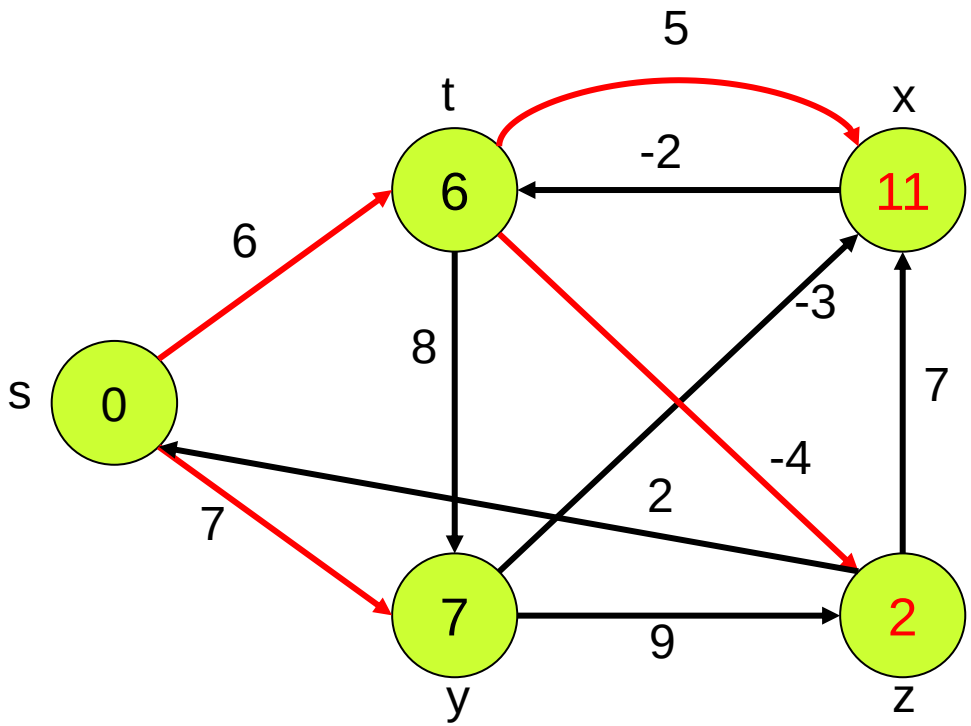
Example

- Order : (t,x), (t,y), (t,z), (x,t), (y,x), (y,z), (z,x), (z,s), (s,t), (s,y)
- At first pass



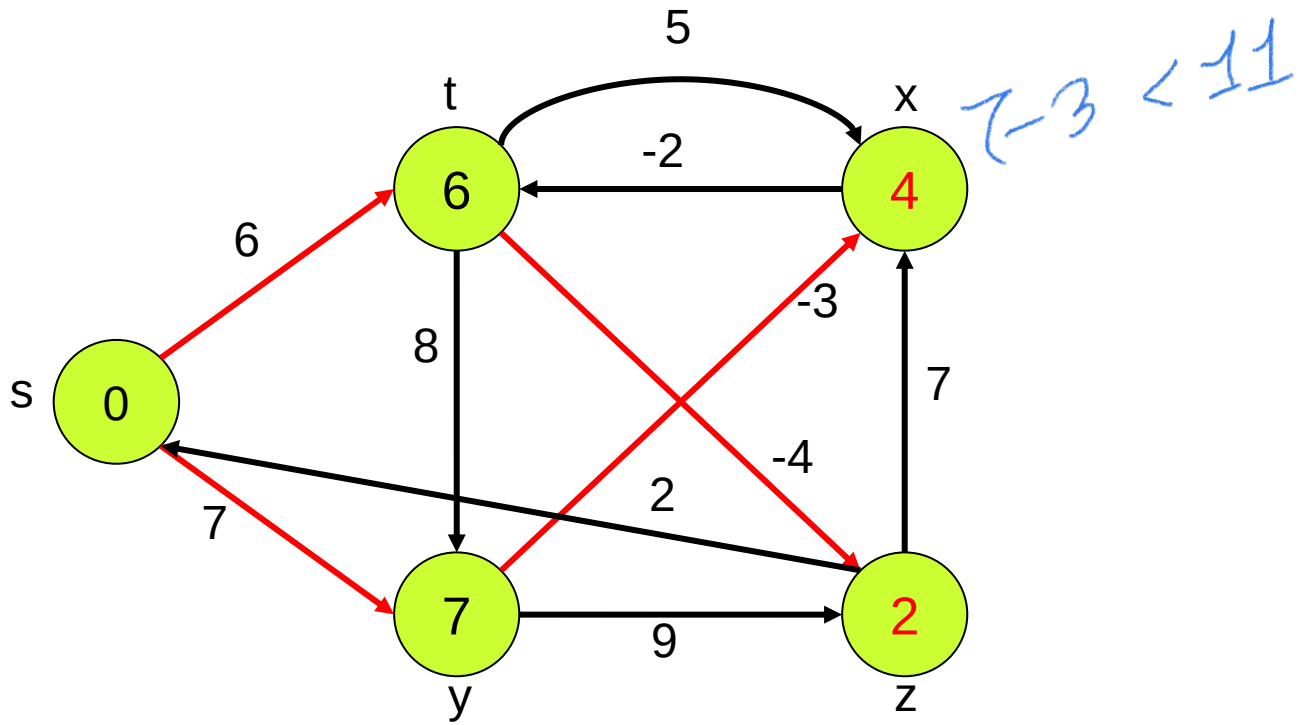
Example

- Order : (t,x), (t,y), (t,z), (x,t), (y,x), (y,z), (z,x), (z,s), (s,t), (s,y)
- At second pass



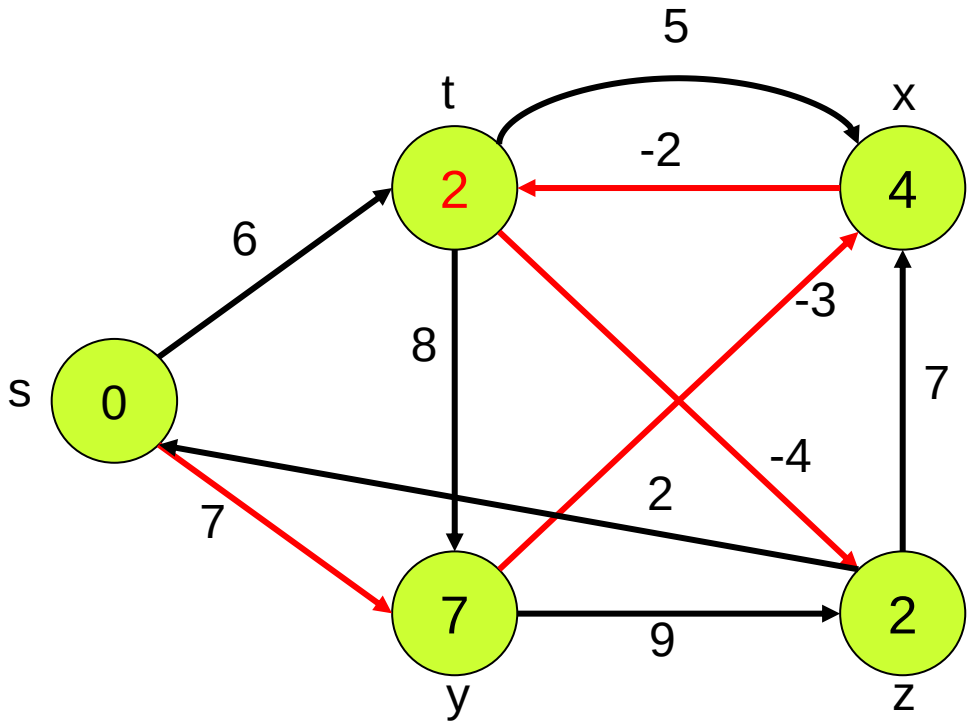
Example

- Order : (t,x), (t,y), (t,z), (x,t), (y,x), (y,z), (z,x), (z,s), (s,t), (s,y)
- At second pass



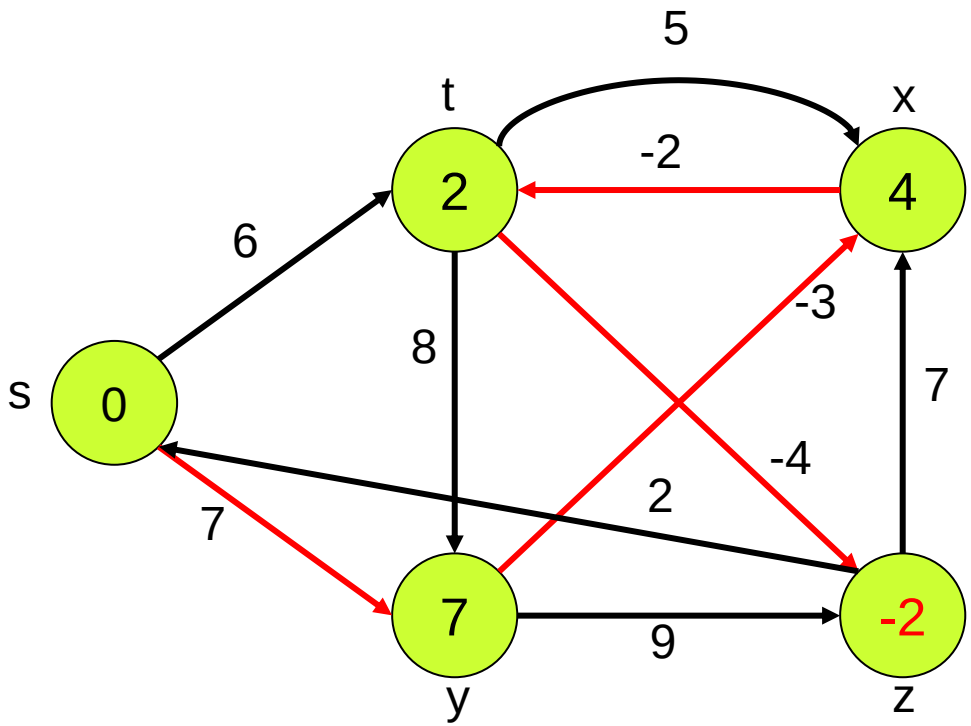
Example

- Order : (t,x), (t,y), (t,z), (x,t), (y,x), (y,z), (z,x), (z,s), (s,t), (s,y)
- At third pass



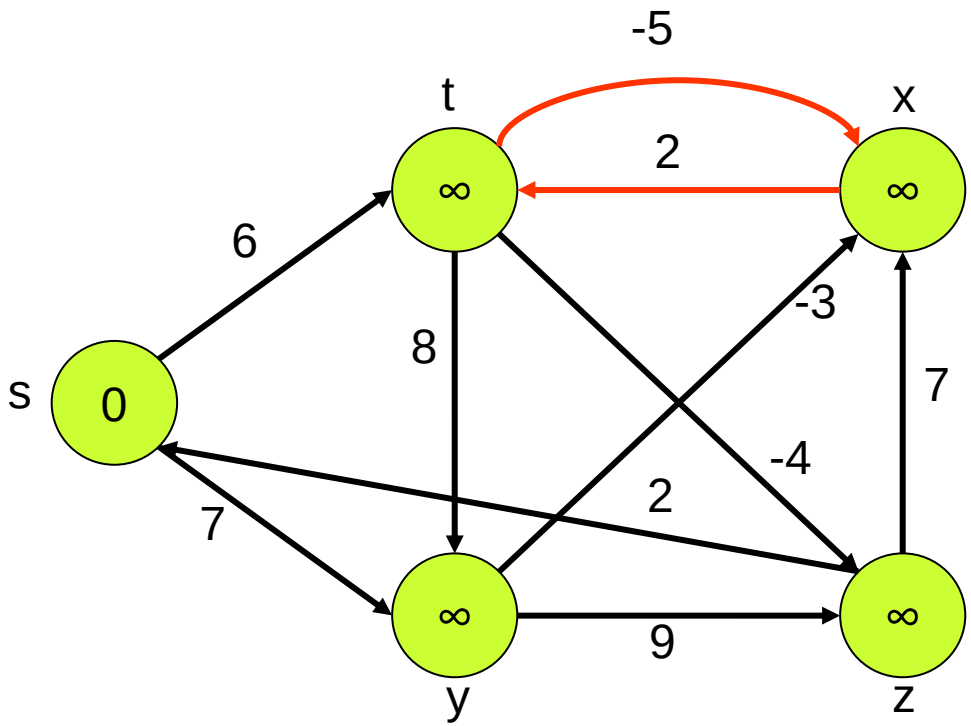
Example

- Order : (t,x), (t,y), (t,z), (x,t), (y,x), (y,z), (z,x), (z,s), (s,t), (s,y)
- At fourth pass



Neg. Weight Cycle Example

- Order : (t,x), (t,y), (t,z), (x,t), (y,x), (y,z), (z,x), (z,s), (s,t), (s,y)
- After INIT-SINGLE-SOURCE(V, s)



Correctness

- Values you get on each pass and how quickly it converges depends on order of relaxation.
- But guaranteed to converge after $|V|$-1 passes, assuming no negative-weight cycles.

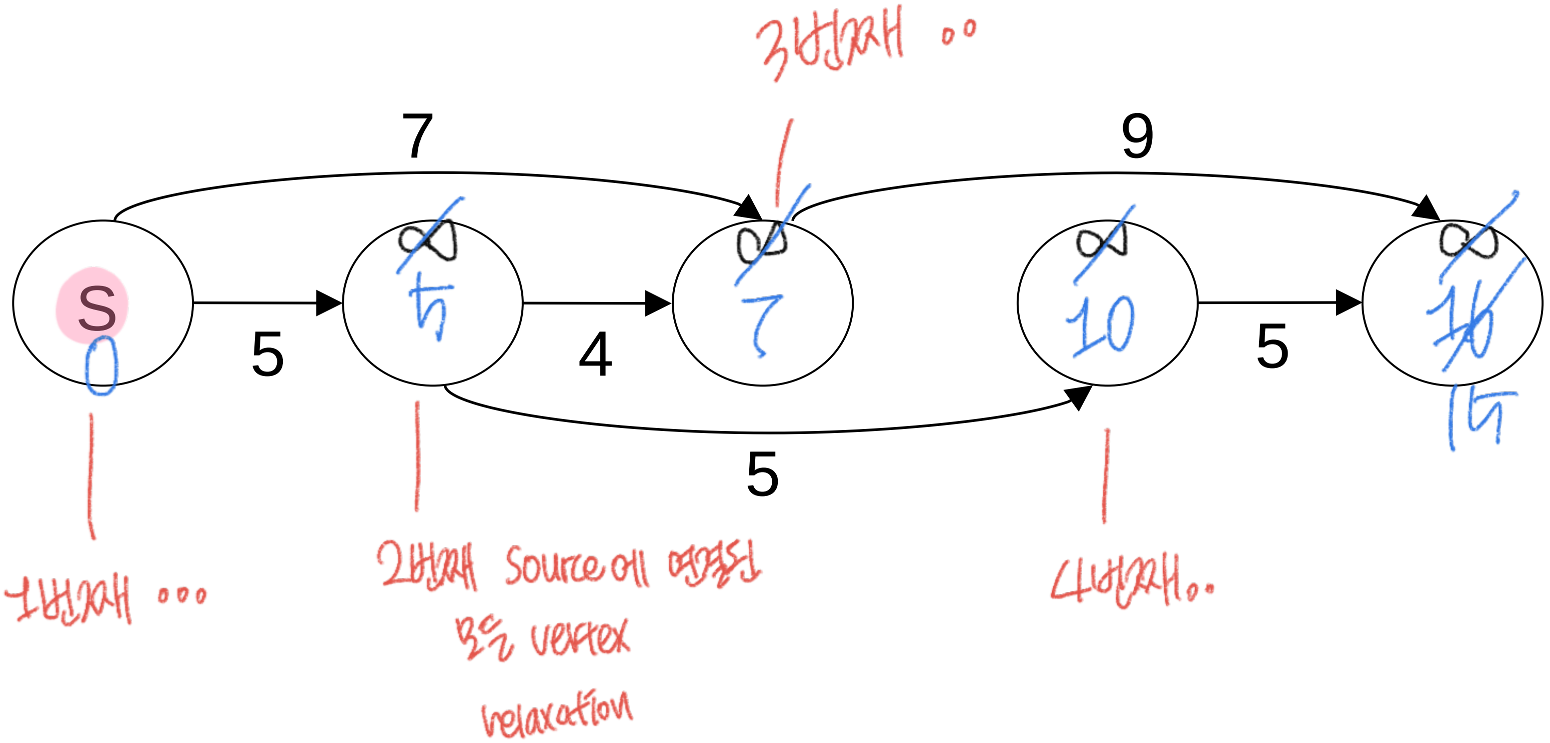
- Proof : use path-relaxation property.

  Theorem 24.4

[2] SSP in DAG

- Problem: finding shortest paths in DAG
  - Bellman-Ford takes Θ(VE) time.
  - *How can we do better?*
  - Idea: use topological sort
    - Since it is a DAG, there are no cycles.
    - Every path in a DAG is subsequence of topologically sorted, so processes vertices on each shortest path from left to right, then it would be done in one pass.
    - *What will be the running time?*

Example


3번째 ..
7
9
S
0
5
5
4
7
10
5
16
15
5
1번째 ...
2번째 Source에 연결된 모든 vertex relaxation
4번째..

SSP in DAG

DAG-SHORTEST-PATHS(V, E, w, s)

topologically sort the vertices $\Theta(V+E)$

INIT-SINGLE-SOURCE(V, s) $\Theta(|V|)$

for each vertex u, take in topologically sorted order

do for each vertex $v \in Adj[u]$ $\Theta(|E|)$

do RELAX(u, v, w)

Time : $\Theta(V+E)$

[3] Dijkstra's Algorithm

- If there are no negative weight edge, we can beat Bellman-Ford.
- Similar to breadth-first search : weighted version of breadth-first search.
  - Grow a tree gradually, advancing from vertices taken from a queue.
  - Instead of a FIFO queue, uses a priority queue.
  - Keys are shortest-path weights ($d[v]$).

Basic Idea

- Have two sets of vertices :
  - S = vertices whose final shortest-path weights are determined.
  - Q = priority queue = $V - S$.
- For the graph $G=(V,E)$, maintains a set S of vertices for which the shortest paths are known.
- Repeatedly selects the vertex u ($u \in V\text{-}S$), with the minimum shortest-path estimated, adds u to S, and relaxes all edges leaving u.

greedy

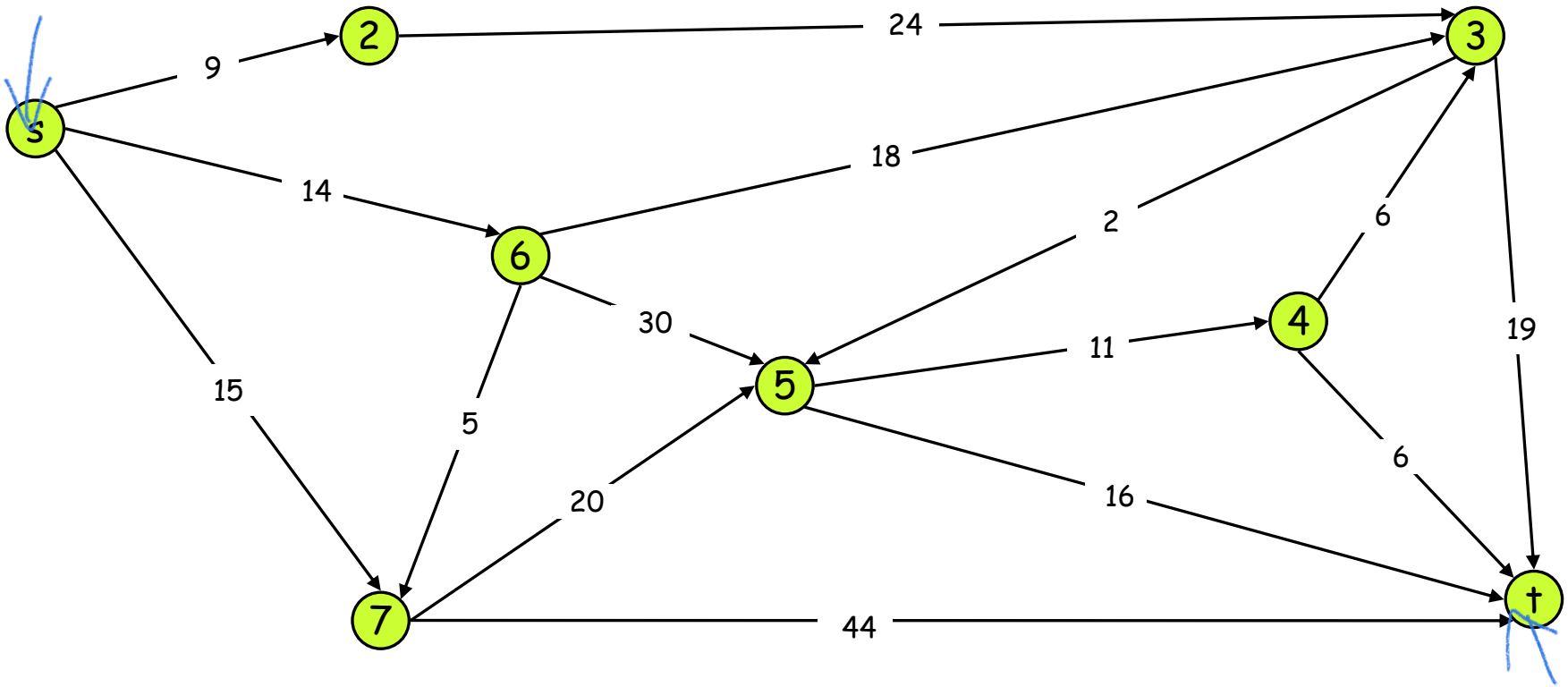
Pseudo-code

```
DIJKSTRA(G,w,s)

1 INITIALIZE-SINGLE-SOURCE(G, s)
2 S ← Ø
3 Q ← V[G]   모든 node 넣기 !
4 while Q ≠ Ø
5     do u ← EXTRACT-MIN(Q)
6        S ← S U {u}
7        for each vertex v ∈ Adj[u]   연결된 모든 edge
8            do RELAX(u, v, w)        relaxation
```

Example

- Find shortest path from s to t.



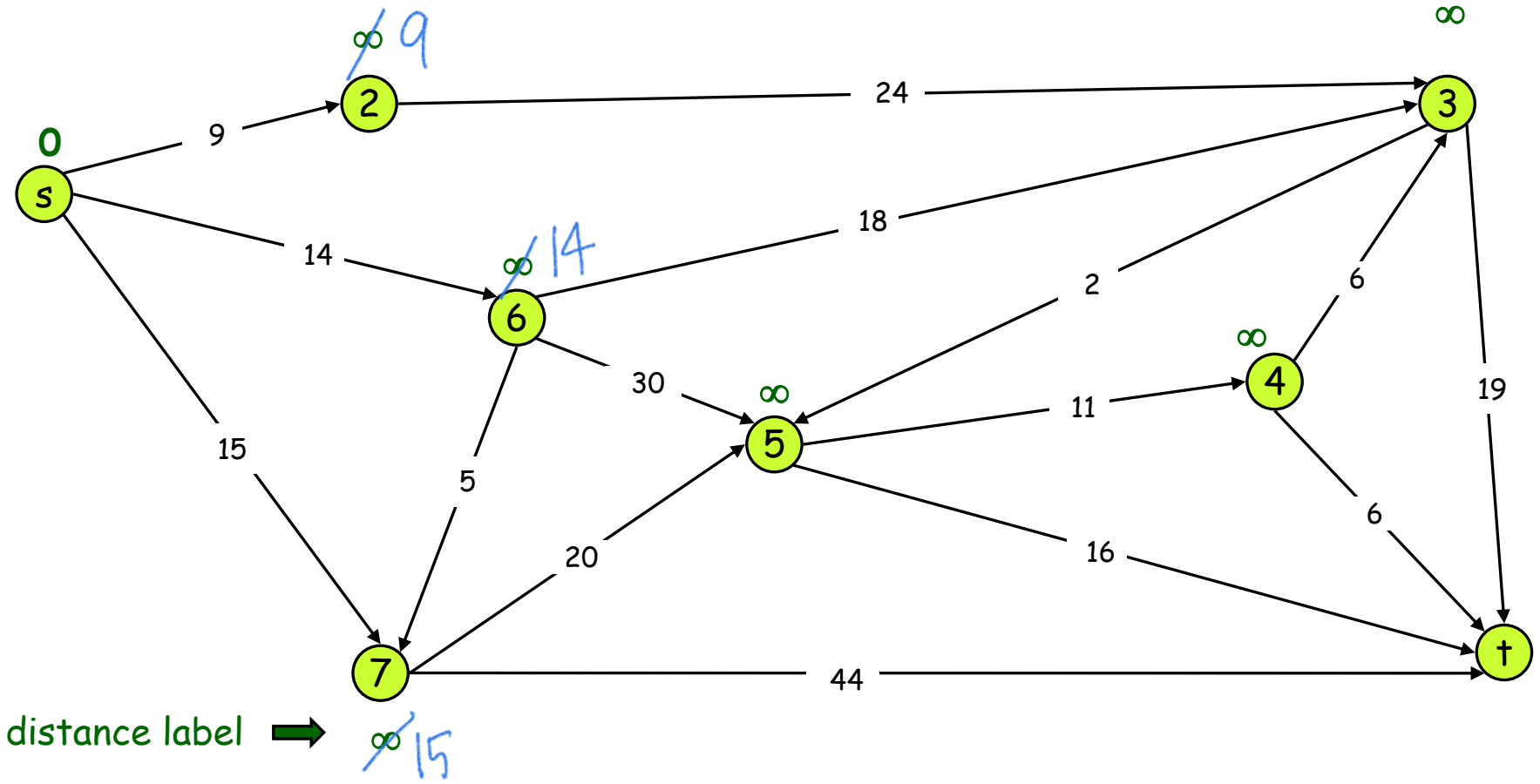
Example

S = { }

Q = { s, 2, 3, 4, 5, 6, 7, t }

INITIALIZE-SINGLE-SOURCE(G, s)

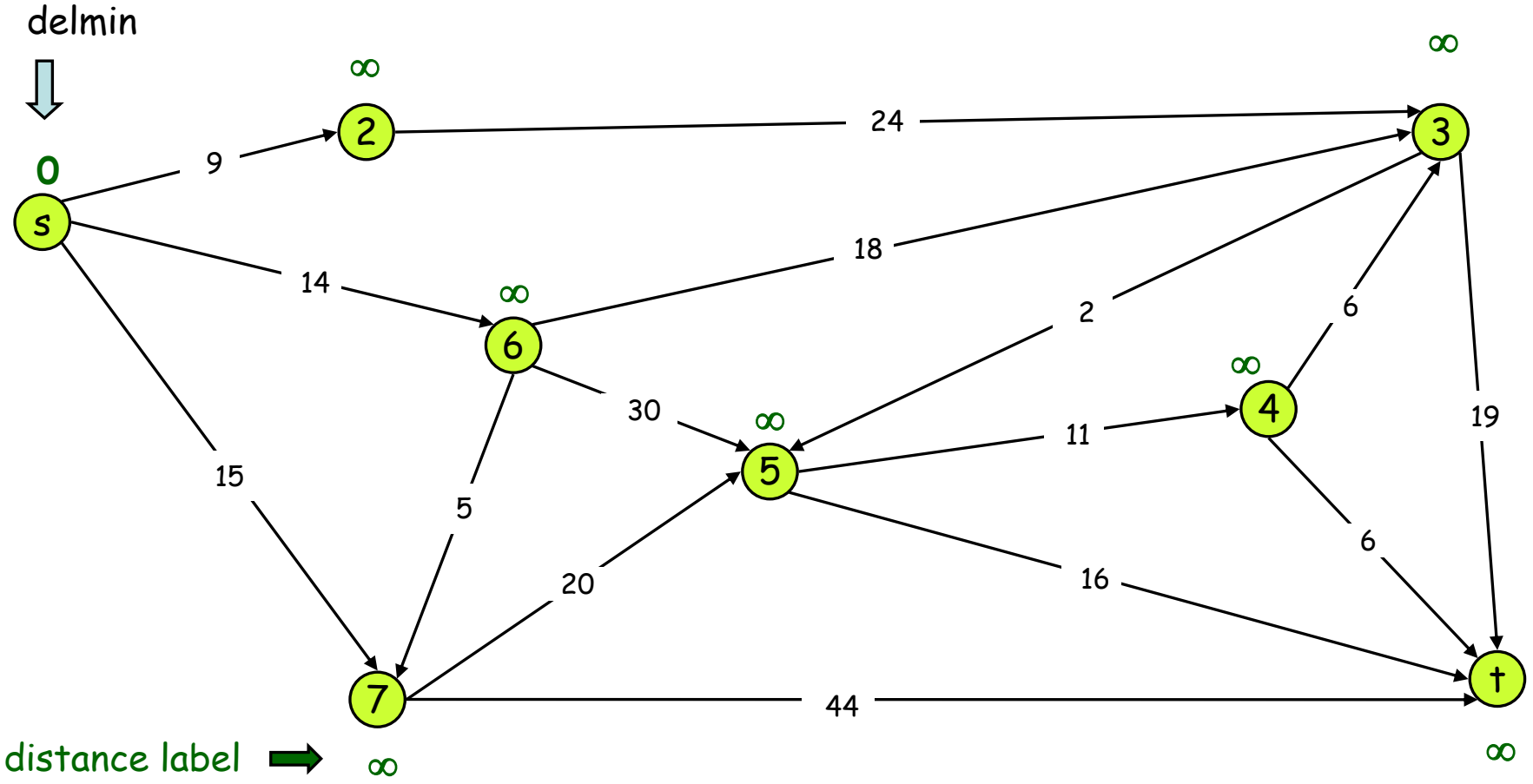
S ← ∅ , Q ← V[G]



Example

S = { }
Q = { s, 2, 3, 4, 5, 6, 7, t }

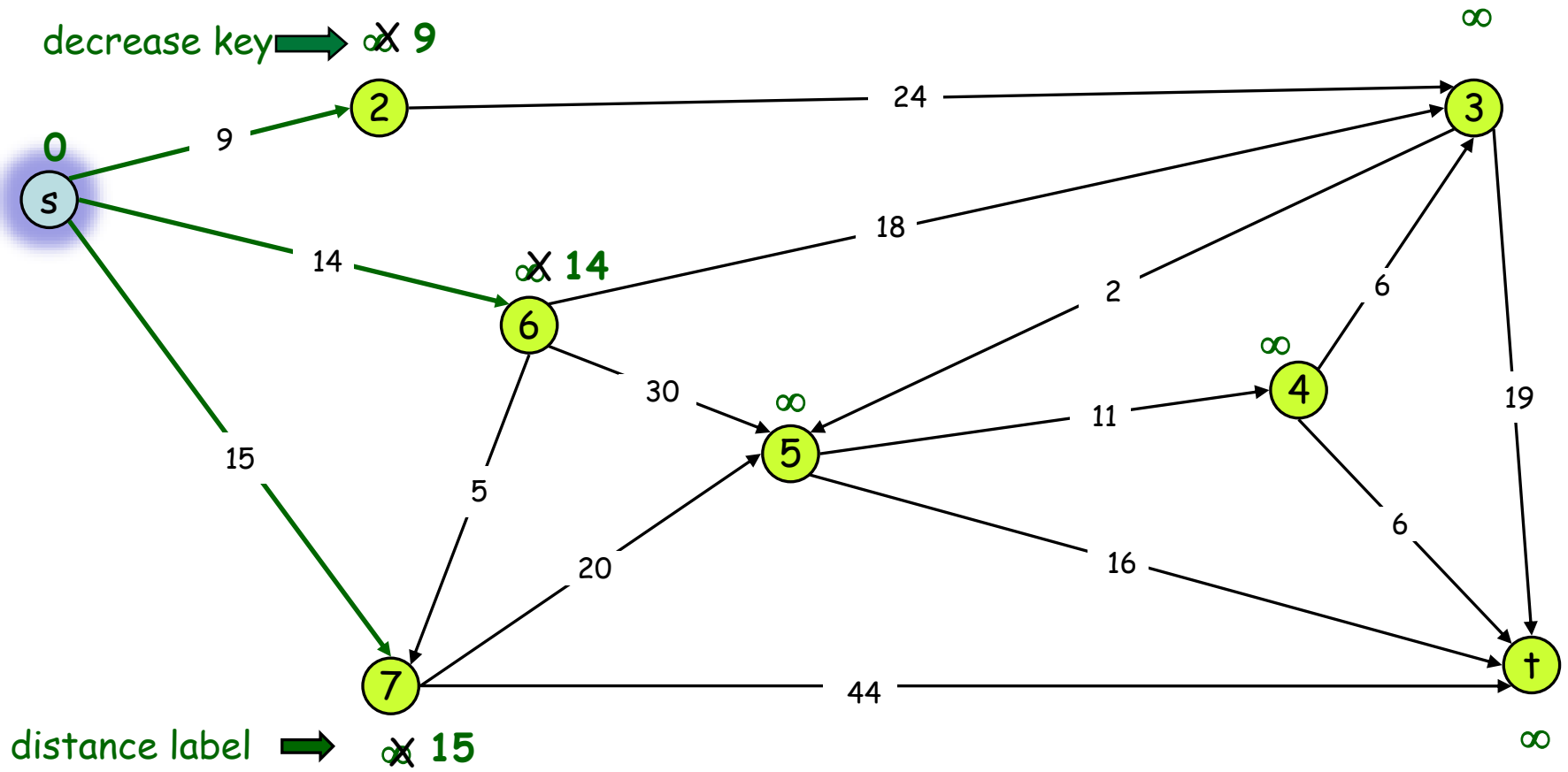
While Q ≠ ∅
do u ← EXTRACT-MIN(Q)



Example

S = { s }
Q = { 2, 6, 7, 3, 4, 5, t }

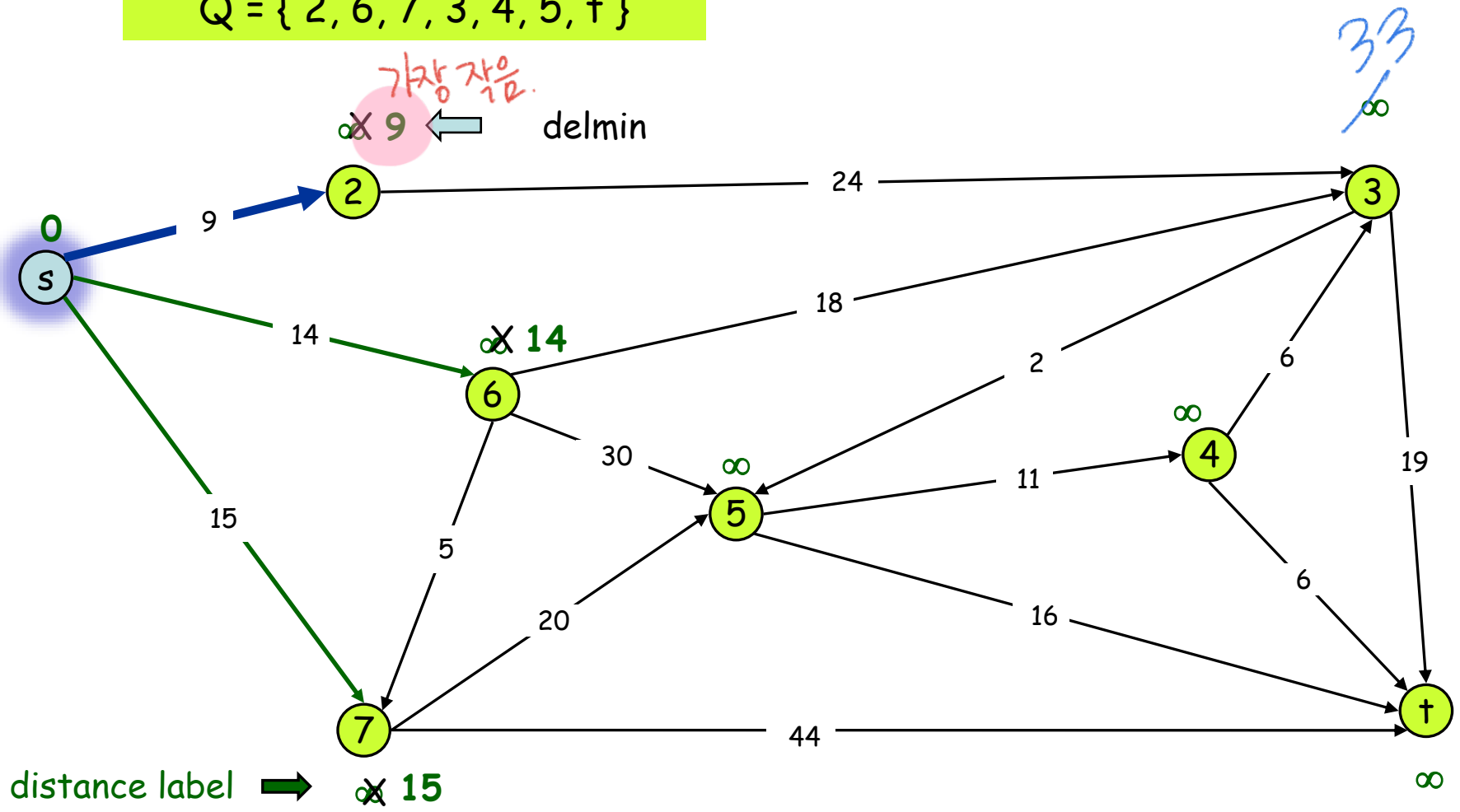
S ← S U {u}
for each vertex v ∈ Adj[u]
do RELAX(u, v, w)



Example

S = { s }
Q = { 2, 6, 7, 3, 4, 5, t }

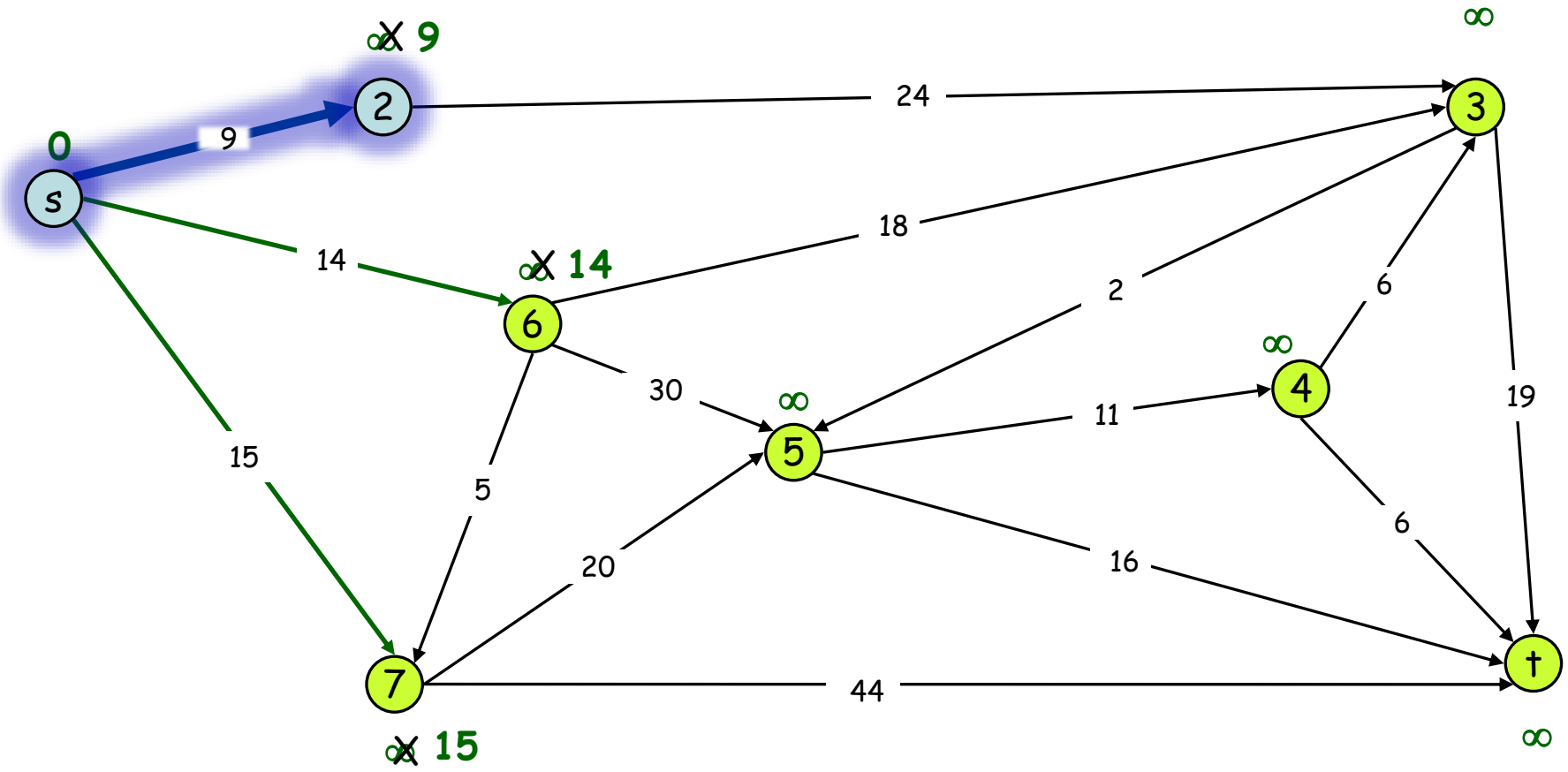
do u ← EXTRACT-MIN(Q)



Example

S = { s, 2 }
Q = { 6, 7, 3, 4, 5, t }

S ← S U {u}

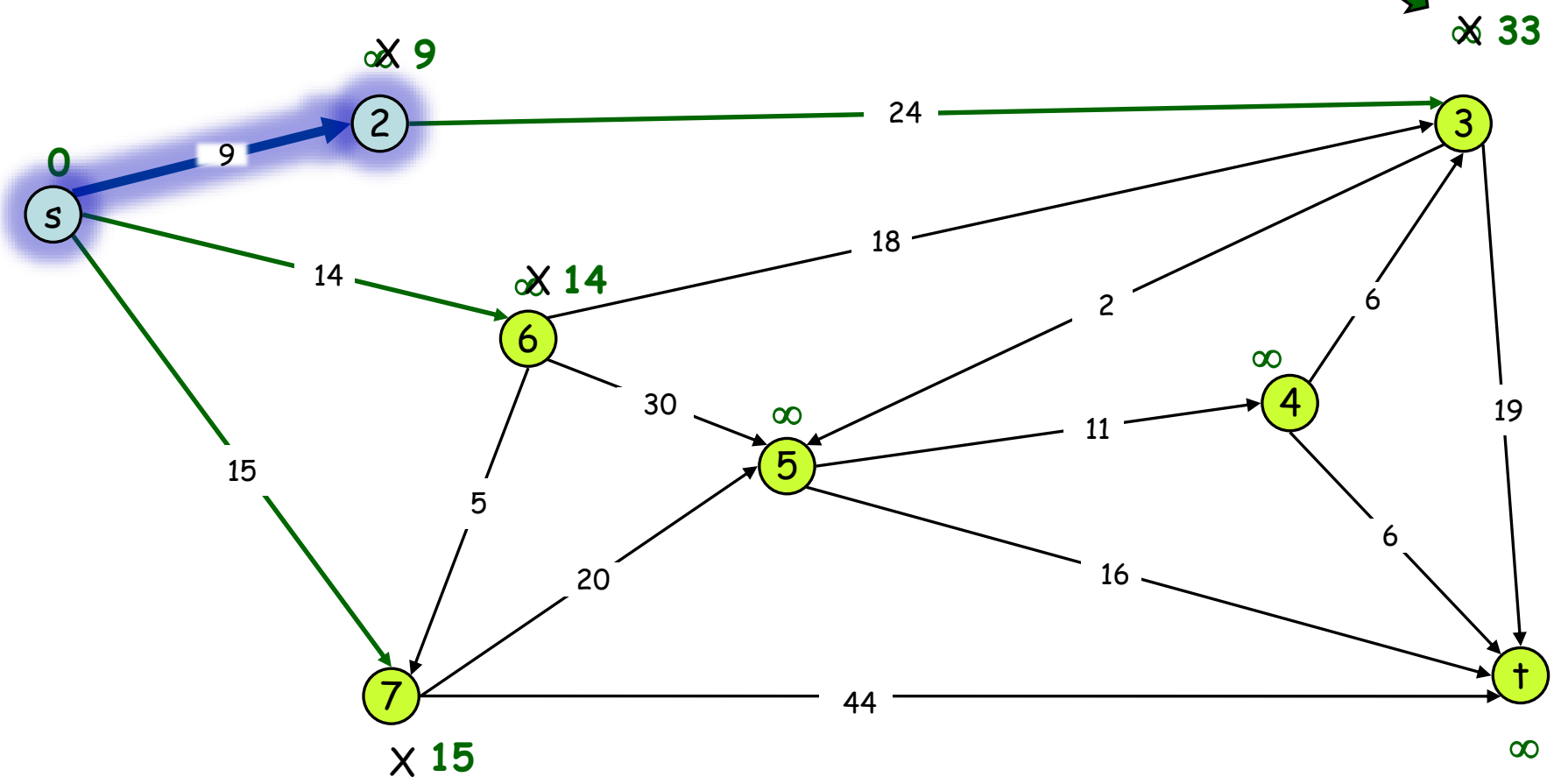


Example

S = { s, 2 }
Q = { 6, 7, 3, 4, 5, t }

for each vertex v ∈ Adj[u]
do RELAX(u, v, w)

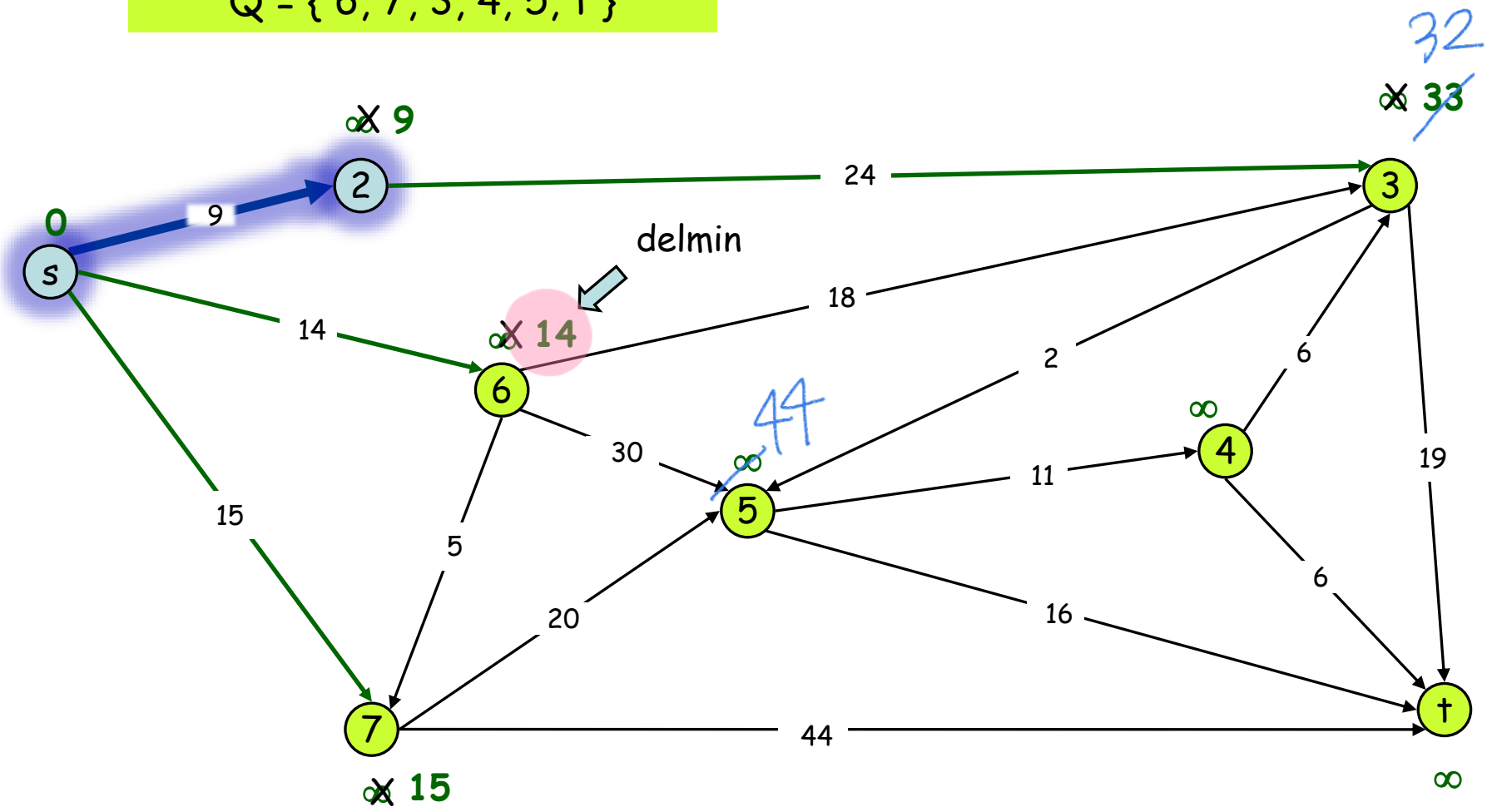
decrease key



Example

S = { s, 2 }
Q = { 6, 7, 3, 4, 5, t }

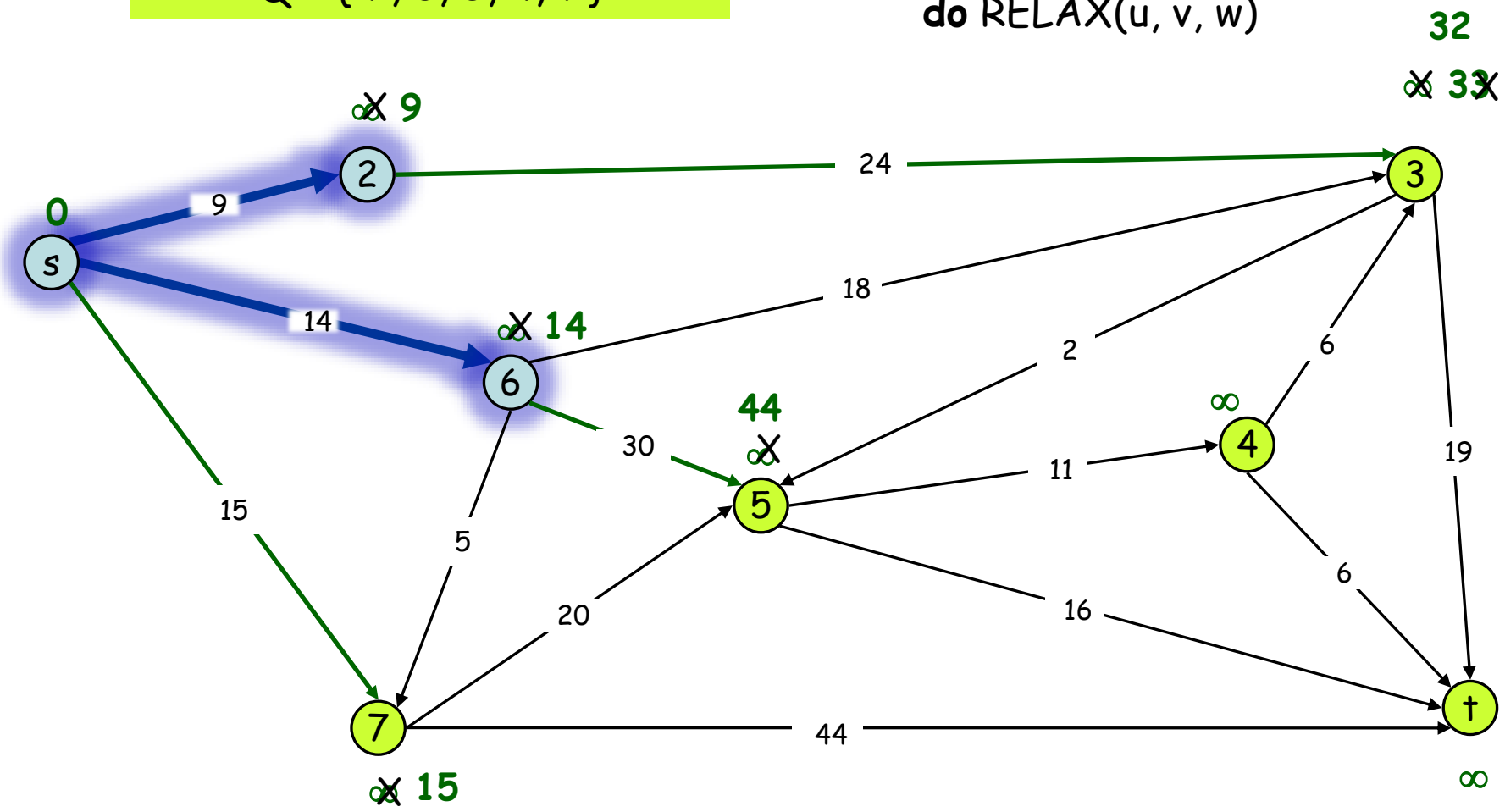
do u ← EXTRACT-MIN(Q)



Example

S = { s, 2, 6 }
Q = { 7, 3, 5, 4, t }

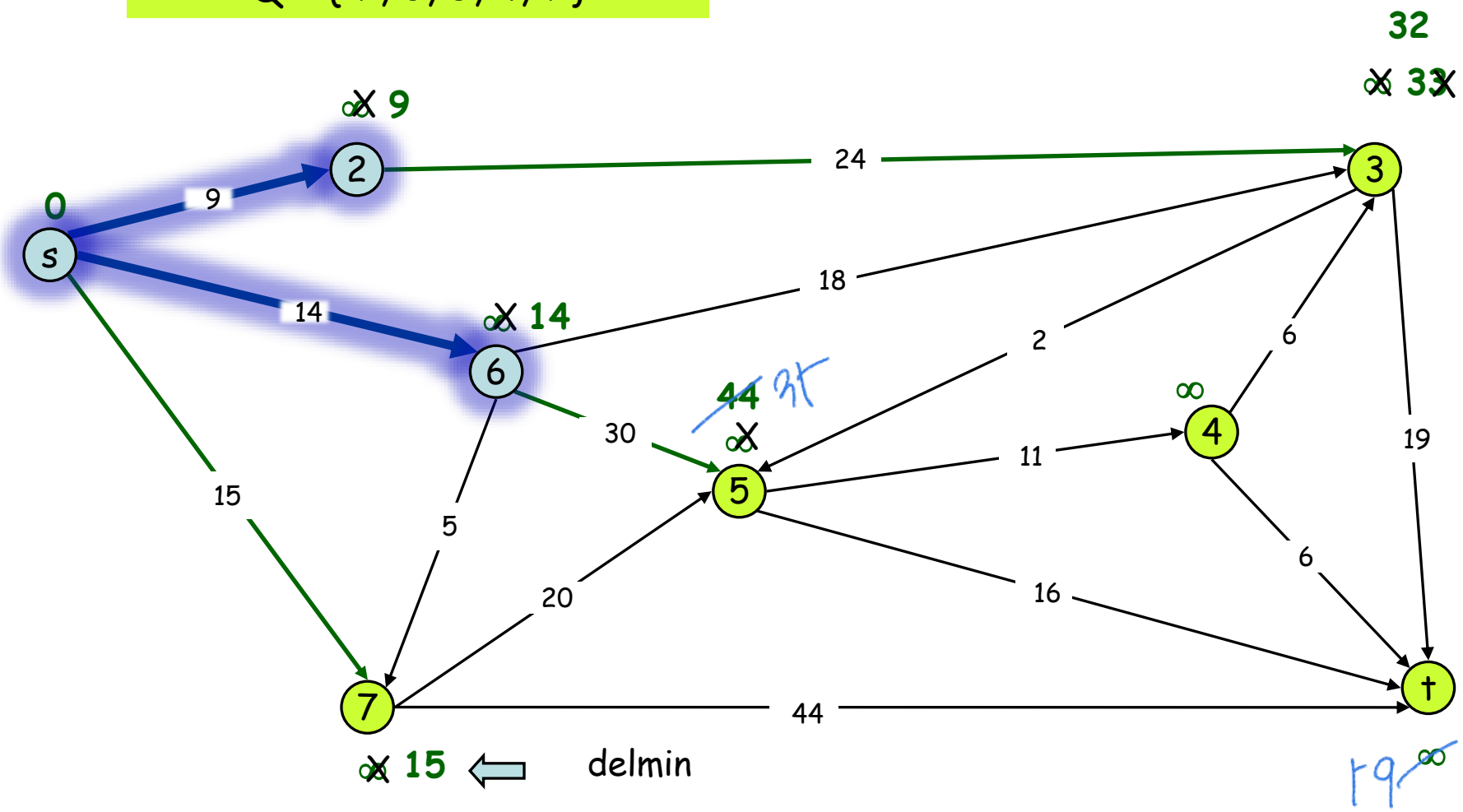
S ← S U {u}
for each vertex v ∈ Adj[u]
do RELAX(u, v, w)



Example

S = { s, 2, 6 }
Q = { 7, 3, 5, 4, t }

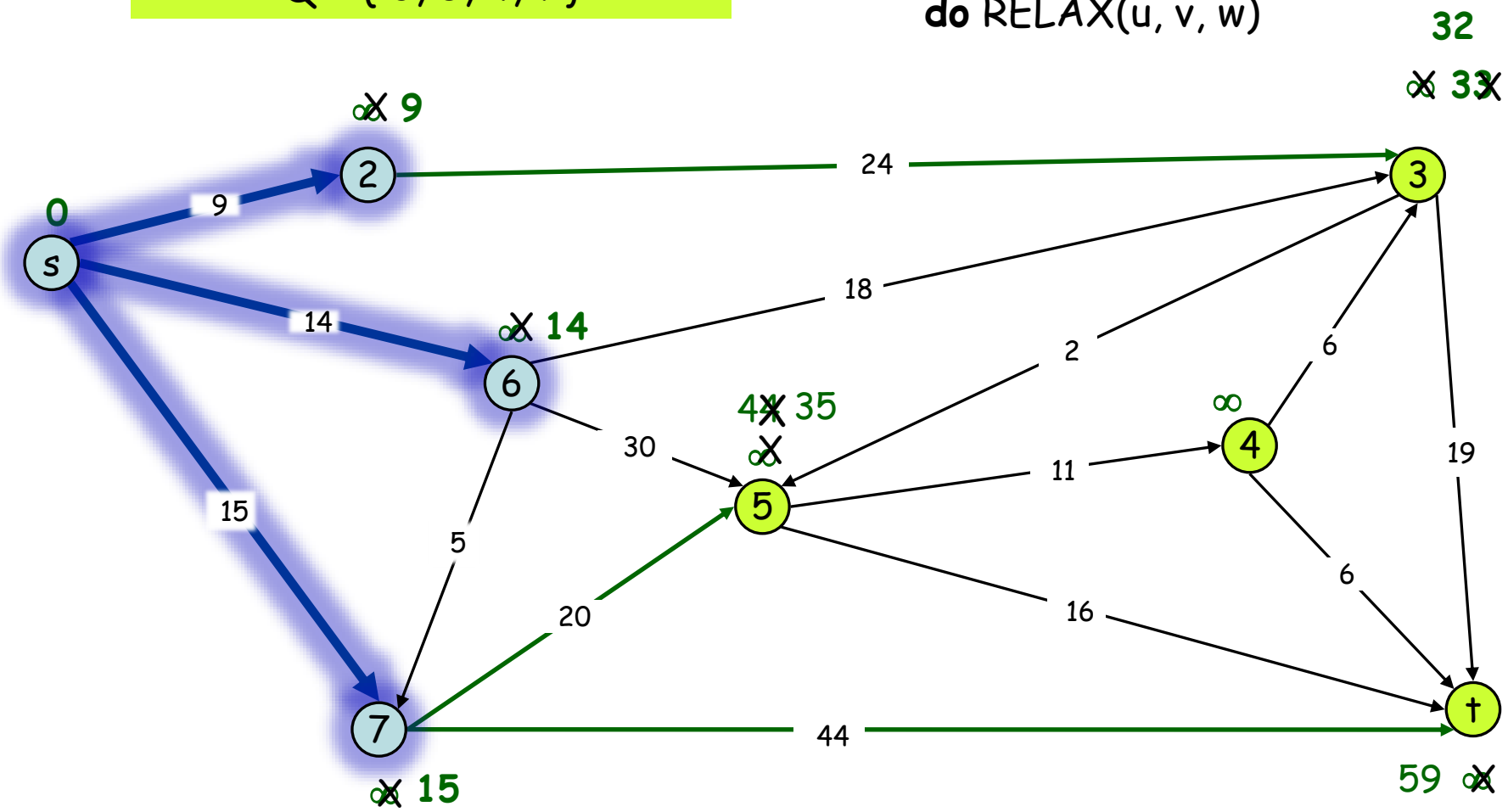
do u ← EXTRACT-MIN(Q)



Example

S = { s, 2, 6, 7 }
Q = { 3, 5, 4, t }

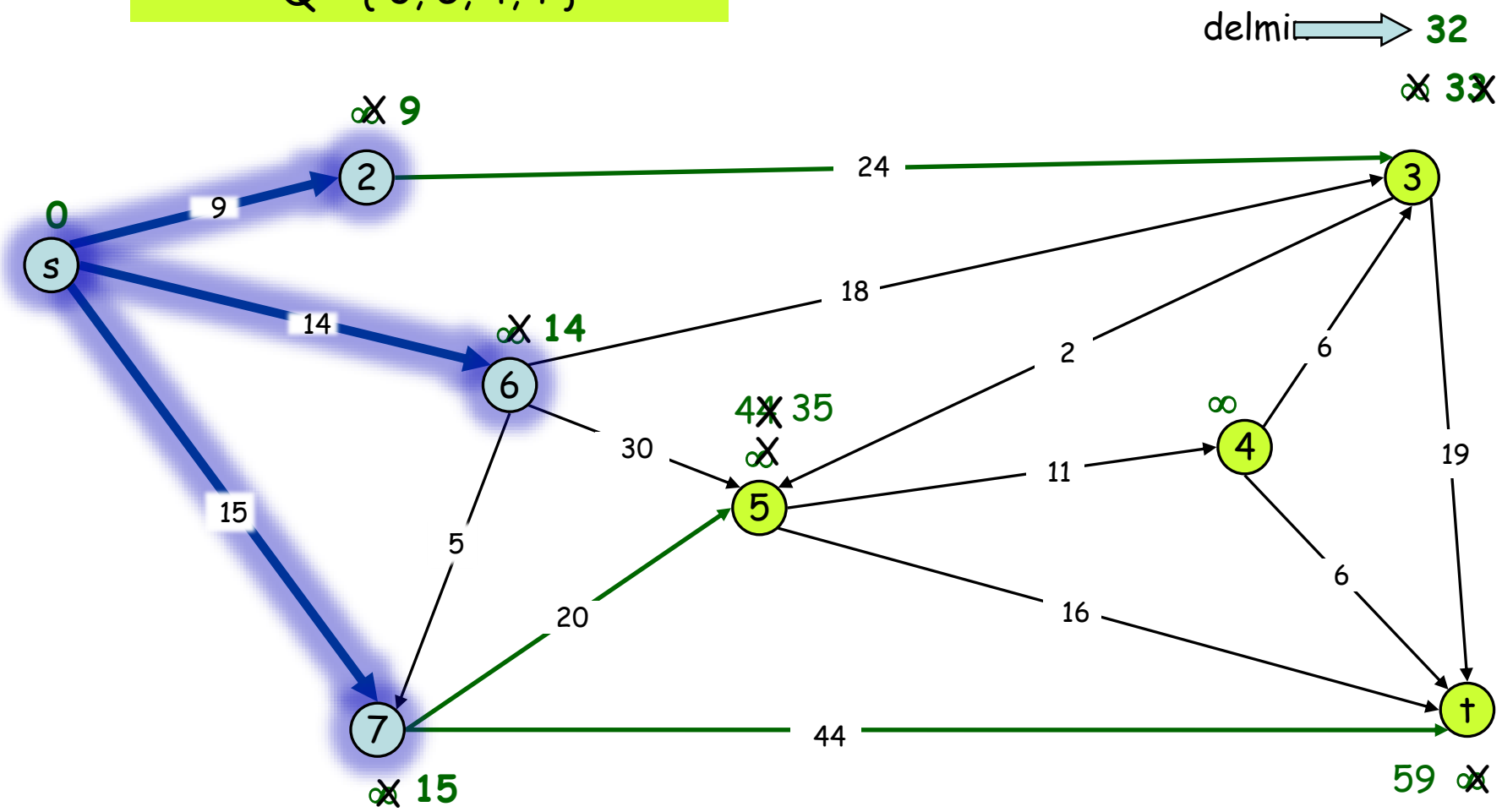
S ← S U {u}
for each vertex v ∈ Adj[u]
do RELAX(u, v, w)



Example

S = { s, 2, 6, 7 }
Q = { 3, 5, 4, t }

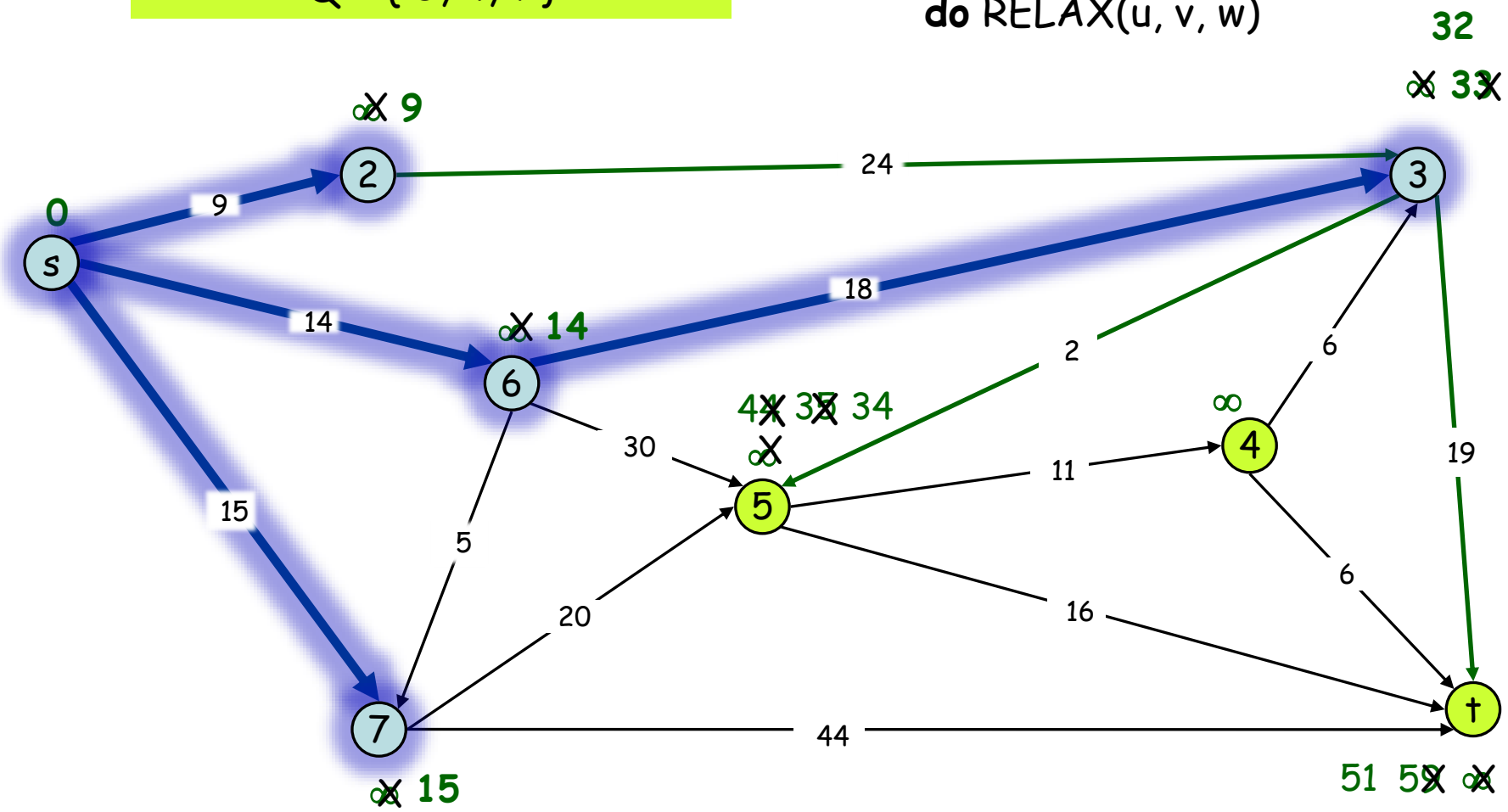
do u ← EXTRACT-MIN(Q)



Example

S = { s, 2, 3, 6, 7 }
Q = { 5, 4, t }

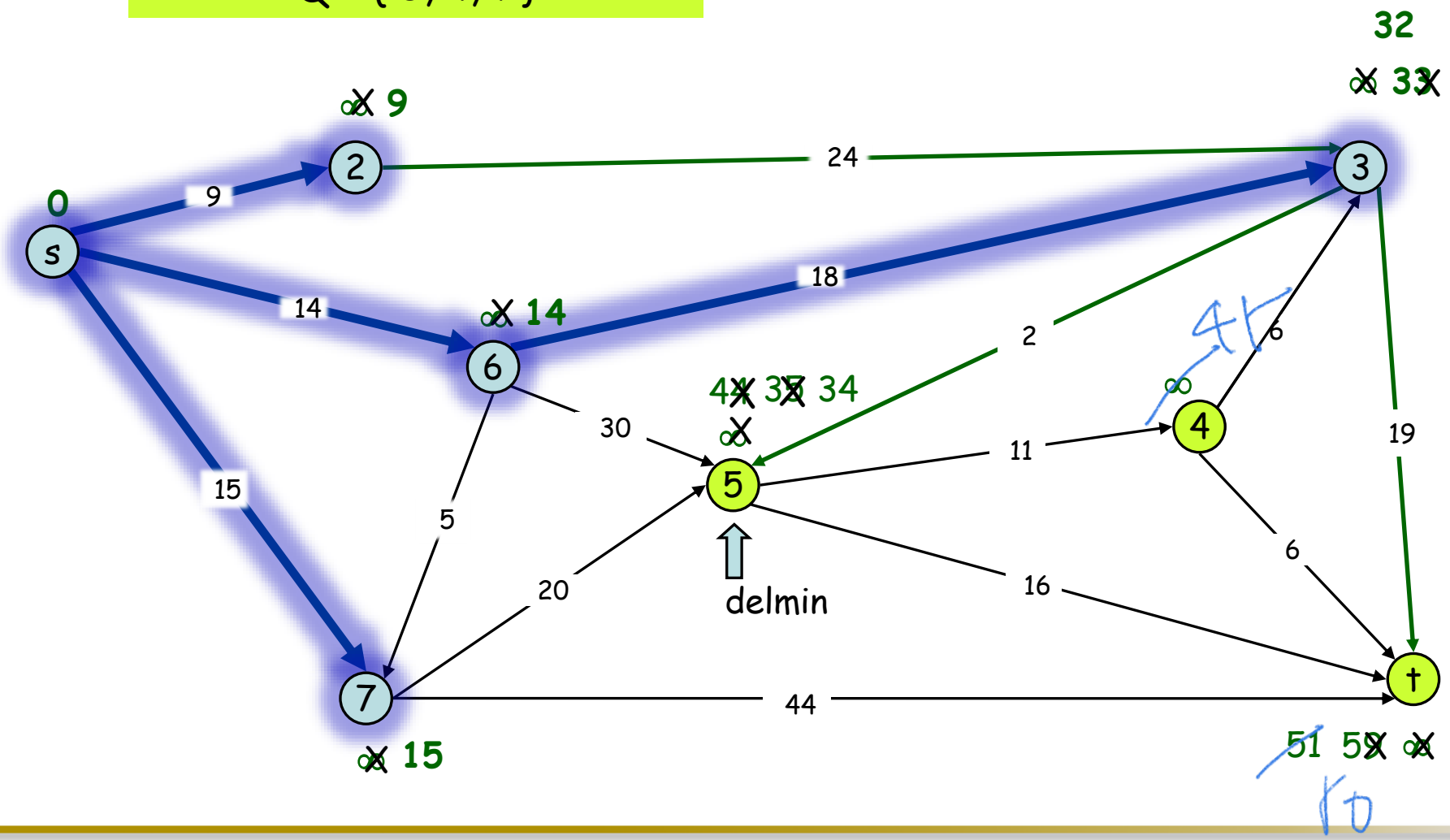
S ← S U {u}
for each vertex v ∈ Adj[u]
do RELAX(u, v, w)



Example

S = { s, 2, 3, 6, 7 }
Q = { 5, 4, t }

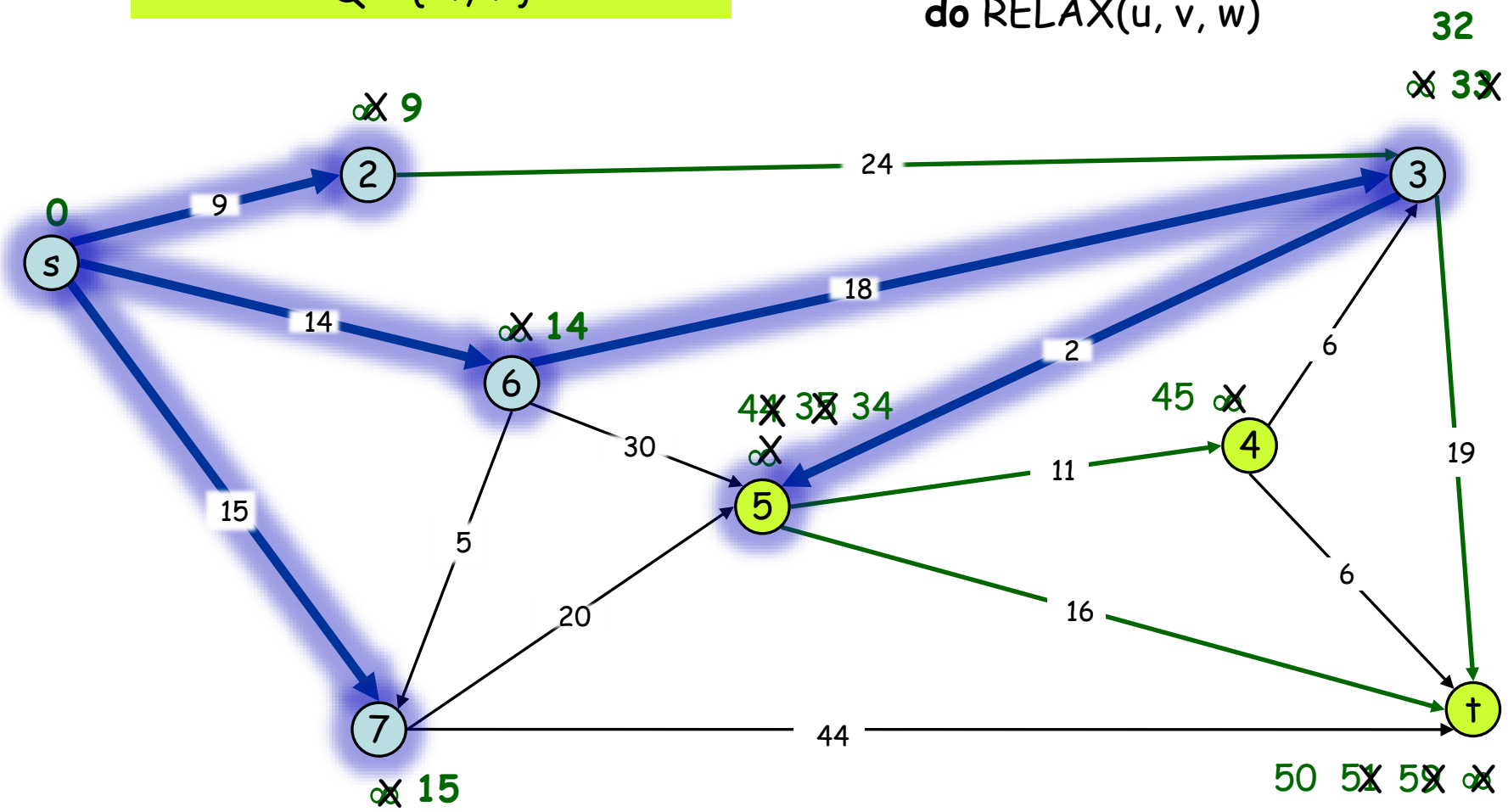
do u ← EXTRACT-MIN(Q)



Example

S = { s, 2, 3, 5, 6, 7 }
Q = { 4, t }

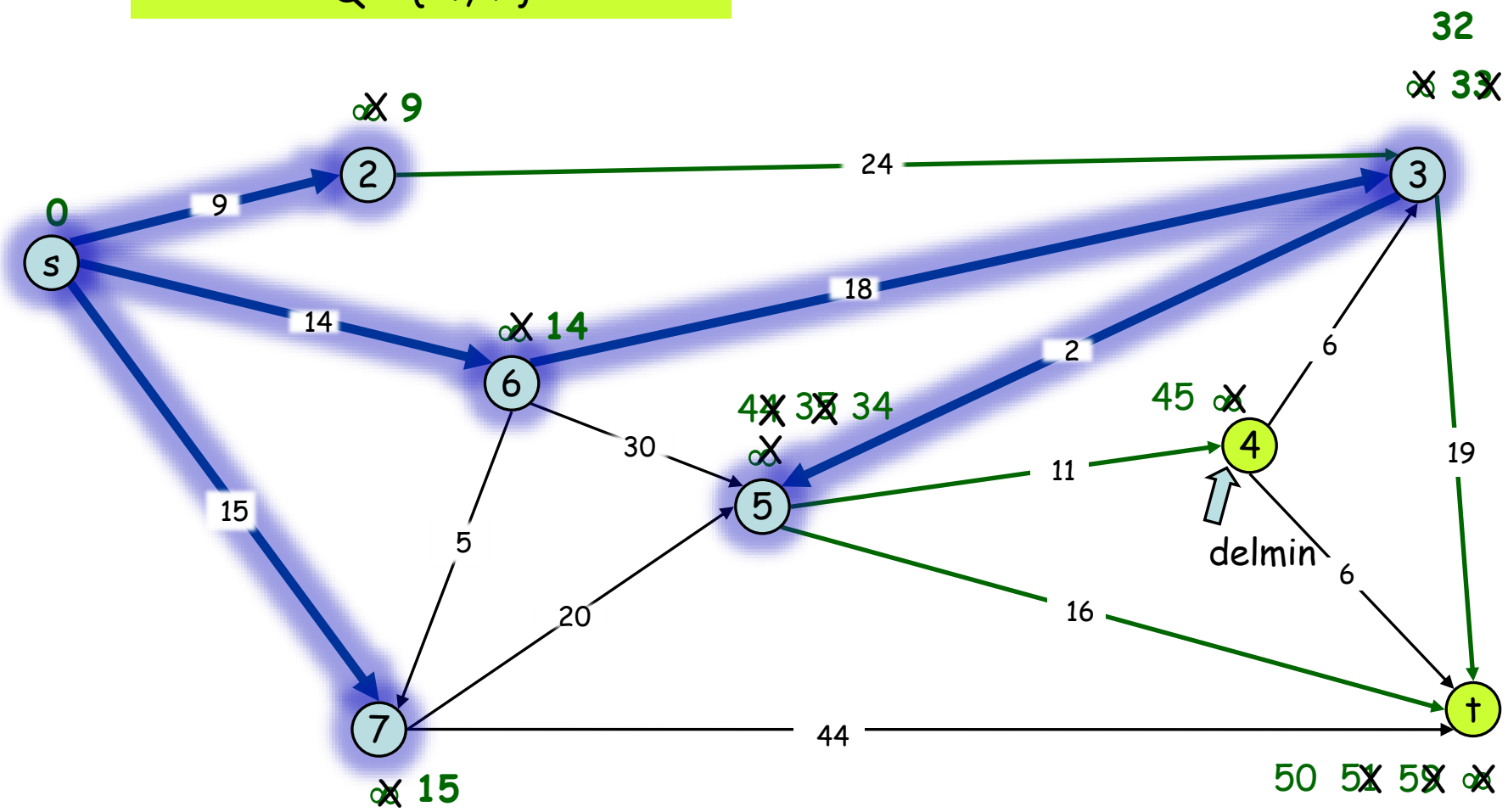
S ← S U {u}
for each vertex v ∈ Adj[u]
do RELAX(u, v, w)



Example

S = { s, 2, 3, 5, 6, 7 }
Q = { 4, t }

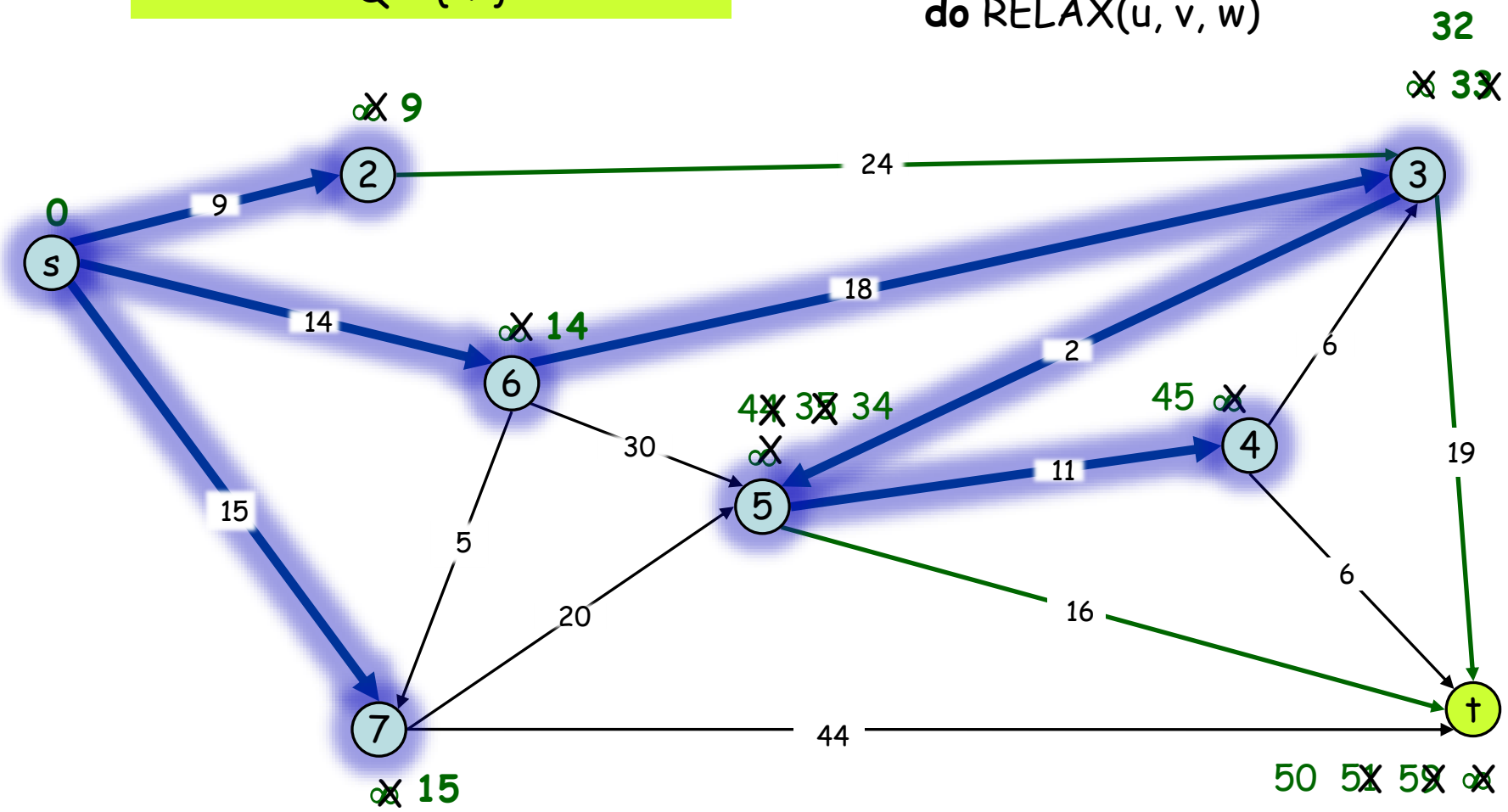
do u ← EXTRACT-MIN(Q)



Example

S = { s, 2, 3, 4, 5, 6, 7 }
Q = { t }

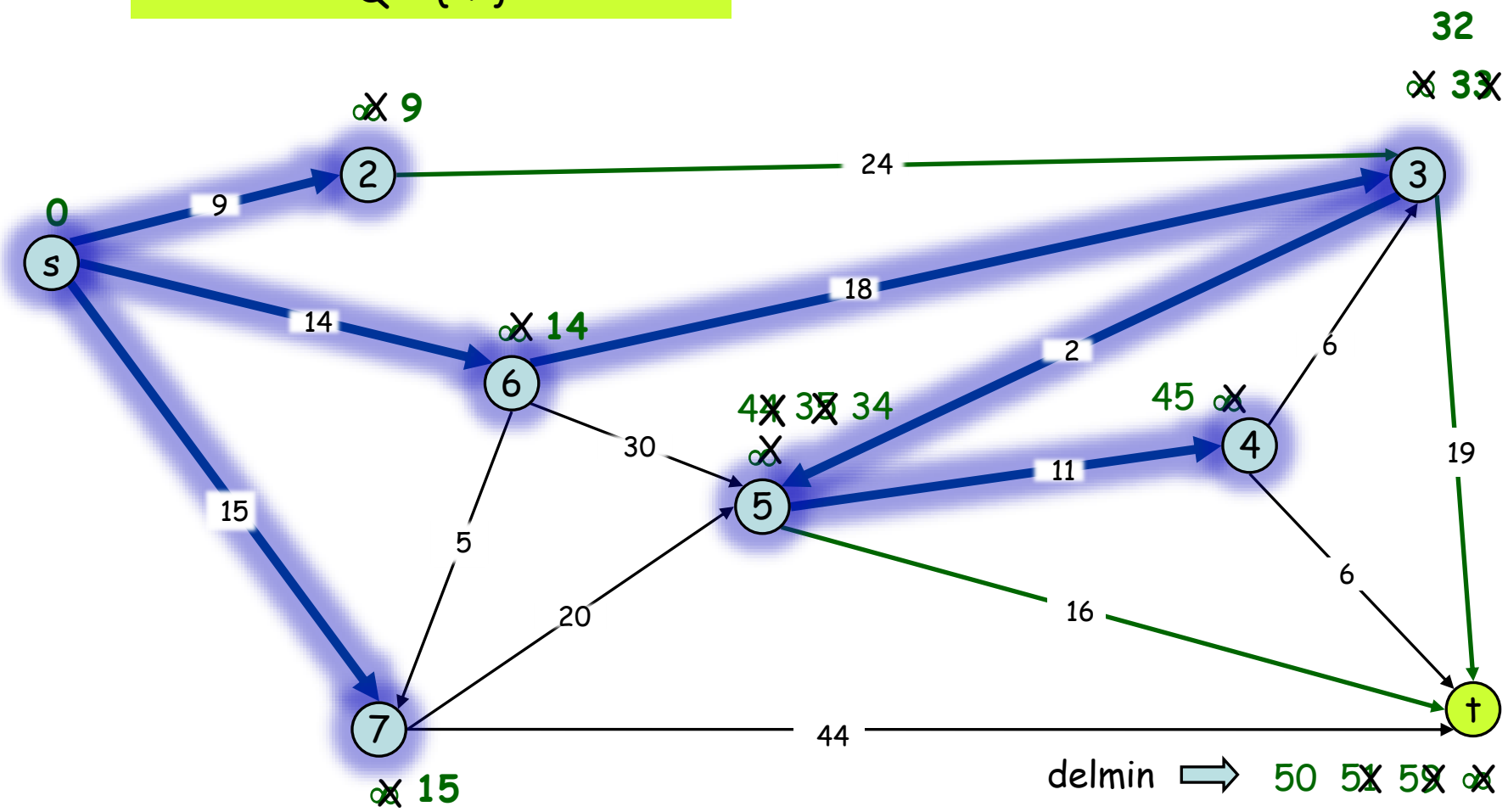
S ← S U {u}
for each vertex v ∈ Adj[u]
do RELAX(u, v, w)



Example

S = { s, 2, 3, 4, 5, 6, 7 }
Q = { t }

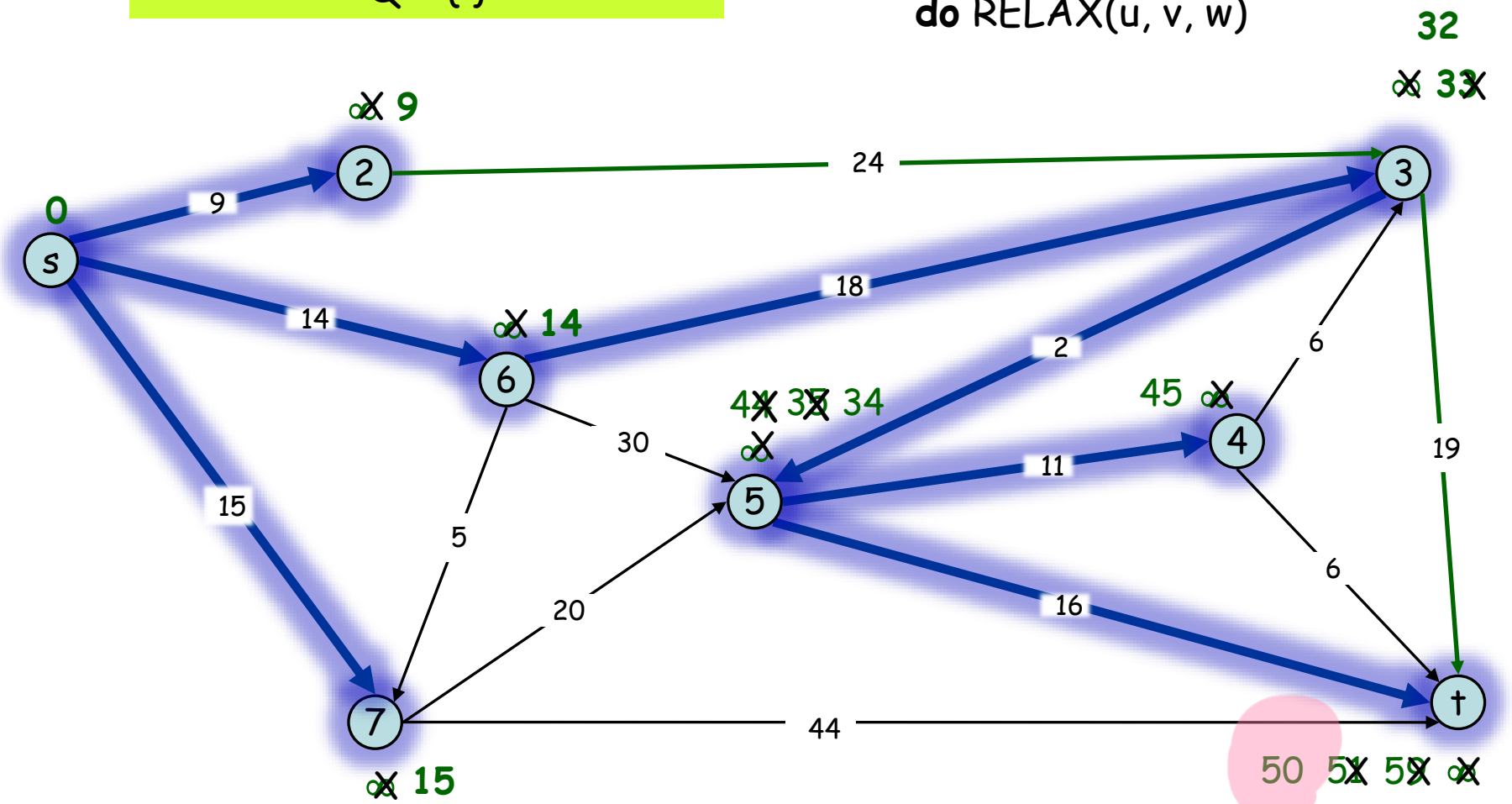
do u ← EXTRACT-MIN(Q)



Example

S = { s, 2, 3, 4, 5, 6, 7, t }
Q = { }

S ← S U {u}
for each vertex v ∈ Adj[u]
do RELAX(u, v, w)



Correctness

Theorem 24.6

loop invariant : At the start of each iteration of the while loop of lines 4-8, $d[v] = \delta(s,v)$ for each vertex $v \in S$.

Proof at page 660 ~ 661

Time Analysis

- Like Prim's algorithm, performance depends on implementation of priority queue.
  - Binary heap :
    - Each operation takes $O(\lg V)$ time

      $\rightarrow O(E \lg V)$
  - Fibonacci heap :
    - $O(V \lg V + E)$ time.

All Pairs Shortest Path

Floyd-Warshall Algorithm

- The easiest way!
  - Iterate Dijkstra's and Bellman-Ford $|V|$ times!

- Dijkstra:
  - $O(V \lg V + E)$ -> $O(V^2 \lg V + VE)$
- Bellman-Ford:
  - $O(VE)$ -> $O(V^2 E)$

On dense graph →

$O(V^3)$

$O(V^4)$

- Faster-All-Pairs-Shortest-Paths (Ch 25.1):
  - $O(V^3 \lg V)$ -> better than Dijkstra and Bellman-Ford ?
- Any other faster algorithms?
  - Floyd-Warshall Algorithm

Floyd-Warshall Algorithm

- Negative edges are allowed
- Assume that no negative-weight cycle
- Dynamic Programming Solution
  - Optimal substructure

The structure of a shortest path

- Intermediate vertex
  - In simple path $p = \langle v_1,\ldots,v_L \rangle$, any vertex of p other than v_1 and v_L, i.e., any vertex in the set $\{v_2,\ldots,v_{L-1}\}$.
- Key Observation
  - For any pair of vertices i, j in V.
  - Let p be a minimum-weight path of all paths from i to j whose intermediate vertices are all from $\{1,2,\ldots,k\}$.
  - Assume that we have all shortest paths from i to j whose intermediate vertices are from $\{1,2,\ldots,k\text{-}1\}$.
  - Observe relationship between path p and above shortest paths.

Key Observation

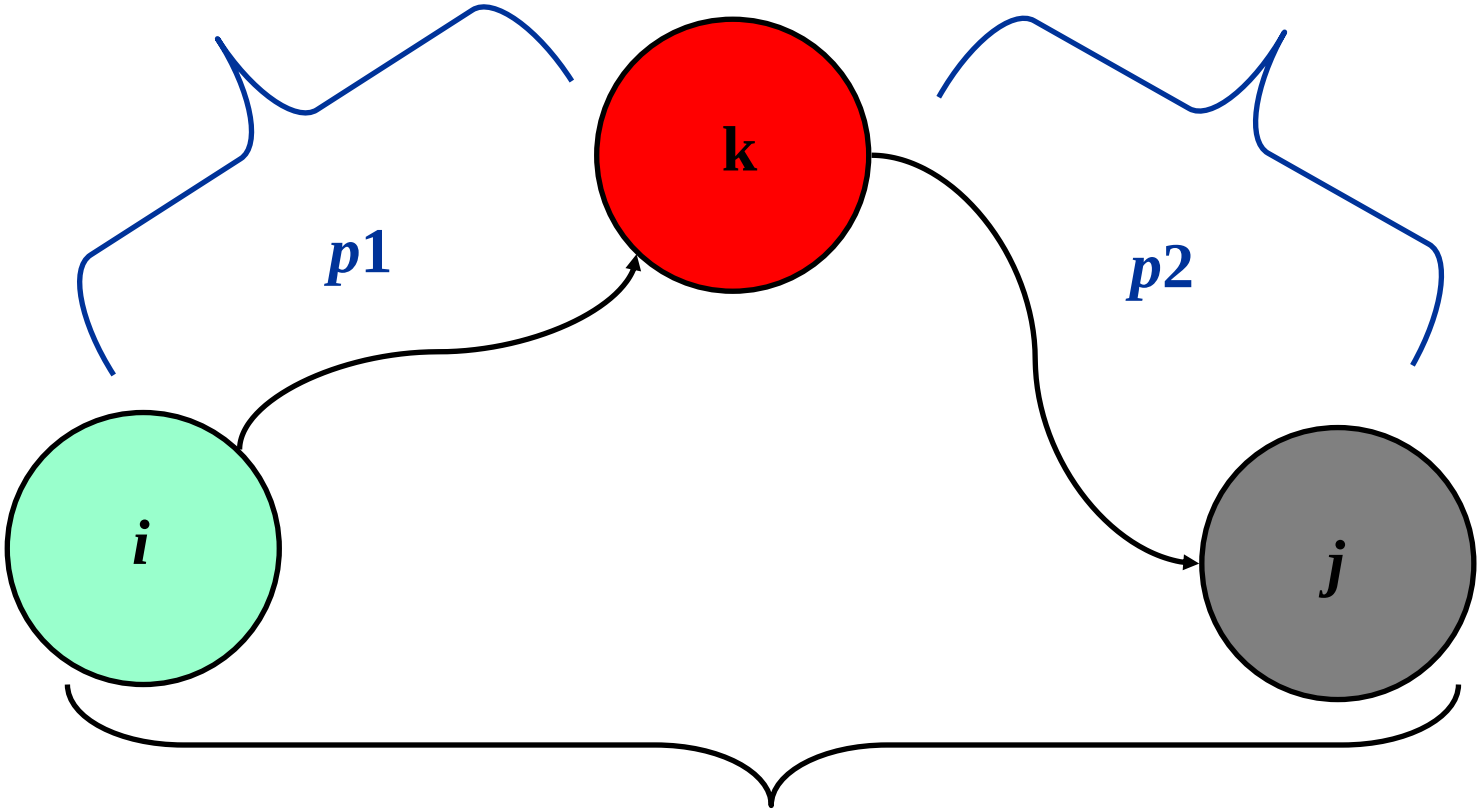
- A shortest path does not contain the same vertex twice.
  - Proof: A path containing the same vertex twice contains a cycle. Removing cycle give a shorter path.

Key Observation

- *p* is determined by the shortest paths whose intermediate vertices from {1,…,*k*-1}.
- Case1: If *k* is not an intermediate vertex of *p*.
  - Path *p* is the shortest path from *i* to *j* with intermediates from {1,…*k*-1}.
- Case2: If *k* is an intermediate vertex of path *p*.
  - Path *p* can be broken down into *i* -- $p1$→ *k* - $p2$ → *j*.
  - *p*1 is the shortest path from *i* to *k* with all intermediate vertices in the set {1,2,…,*k*-1}.
  - *p*2 is the shortest path from *k* to *j* with {1,2,…,*k*-1}.

Key Observation

***p*1**:All intermediate vertices in {1,2,..,*k*-1} ***p*2**:All intermediate vertices in {1,2,..,*k*-1}



***p*: All intermediate vertices in {1,2,..,*k*}**

A recursive solution

- Let $d_{ij}^{(k)}$ be the length of the shortest path from i to j such that all intermediate vertices on the path are in set $\{1,2,\ldots,k\}$.
- Let $D^{(k)}$ be the n X n matrix $[d_{ij}^{(k)}]$.
- $d_{ij}^{(0)}$ is set to be w_{ij} (no intermediate vertex).
- $d_{ij}^{(k)} = \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)})$ $(k \geq 1)$
- $D^{(n)} = (d_{ij}^{(n)})$ gives the final answer, for all intermediate vertices are in the set $\{1,2,\ldots,n\}$.

A recursive solution

- $d_{ij}^{(k)} = \begin{cases} w_{ij} & \text{(if } k=0\text{)} \\ \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}) & \text{(if } k \geq 1\text{)} \end{cases}$

- The Matrix $D^{(n)} = (d_{ij}^{(n)})$ gives the final answer:

  $d_{ij}^{(n)} = \delta(i,j)$ for all $i,j \in V$.

Extracting the Shortest Paths

- The predecessor pointers *pred*[*i*,*j*] can be used.
- Initially all *pred*[*i*,*j*] = nil
- Whenever the shortest path from *i* to *j* passing through an intermediate vertex *k* is discovered, we set *pred*[*i*,*j*] = *k*

Extracting the Shortest Paths

- Observation:
  - If *pred*[*i*,*j*] = nil, shortest path does not exist.
  - If there exists shortest path and the shortest path does not pass through any intermediate vertex, then *pred*[*i*,*j*] = *i*.
  - If *pred*[*i*,*j*] = *k*, vertex k is an intermediate vertex on shortest path form *i* to *j*

Extracting the Shortest Paths

- How to find?
  - If *pred*[*i*,*j*] = *i*, the shortest path is edge (*i*,*j*)
  - Otherwise, recursively compute

    (*i* , *pred*[*i*,*j*]) and (*pred*[*i*,*j*] , *j*)

Computing the weights bottom up

- The Floyd-Warshall Algorithm: Version 1

Floyd-Warshall(*w,n*)

```
{   for i =1 to n do                          initialize
        for j=1 to n do
        { D^0[i,j] = w[i,j] ;
```

$$\pi_{ij}^{(0)} = \begin{cases} \text{NIL} & \text{if } i = j \text{ or } w_{ij} = \infty , \\ i & \text{if } i \neq j \text{ and } w_{ij} < \infty . \end{cases}$$

```
    for k = 1 to n do                         dynamic programming
        for i = 1 to n do
            for j = 1 to n do
                if(d^(k-1)[i,k] + d^(k-1)[k,j] < d^(k-1)[i,j])
                    { d^(k)[i,j] = d^(k-1)[i,k] + d^(k-1)[k,j];      case2
                      pred[i,j] = k;
                      else d^(k)[i,j] = d^(k-1)[i,j];                case1
    return d(n)[1..n, 1..n];
}
```

See figure 25.4.

Can you find a shortest path from vertex 1 to vertex 2 from the π matrix?

Analysis

- Running time is clearly Θ(?)
- $\Theta(n^3)$ -> $\Theta(|V|^3)$
- Faster than previous algorithms. $O(|V|^4)$,$O(|V|^3 \lg |V|)$
- Problem: Space Complexity $\Theta(|V|^3)$.
- It is possible to reduce this down to $\Theta(|V|^2)$by keeping only one matrix instead of n.

Transitive Closure

- Given directed graph $G = (V, E)$
- Compute $G^* = (V, E^*)$
- $E^* = \{(i,j) :$ there is path from i to j in $G\}$
- Could assign weight of 1 to each edge, then run FLOYD-WARSHALL
- If $d_{ij} < n$, then there is a path from i to j.
- Otherwise, $d_{ij} = \infty$ and there is no path.

Transitive Closure – Warshall

- Using logical operations $\vee$ (OR), $\wedge$ (AND)
- Assign weight of 1 to each edge, then run FLOYD-WARSHALL with this weights.
- Instead of $D^{(k)}$, we have $T^{(k)} = (t_{ij}^{(k)})$
  - $t_{ij}^{(0)} = \begin{cases} 0 & (\text{if } i \neq j \text{ and } (i, j) \notin E) \\ 1 & (\text{if } i = j \text{ or } (i, j) \in E) \end{cases}$
  - $t_{ij}^{(k)} = \begin{cases} 1 & (\text{ if there is a path from } i \text{ to } j \text{ with all intermediate vertices in } \{1, 2, \ldots, k\}) \\ & \quad (t_{ij}^{(k-1)} \text{ is 1}) \text{ or } (t_{ik}^{(k-1)} \text{ is 1 and } t_{kj}^{(k-1)} \text{ is 1}) \\ 0 & (\text{otherwise}) \end{cases}$

Transitive Closure

TRANSITIVE-CLOSURE(E, n)

for $i = 1$ to n

  do for $j = 1$ to n

    do if $i=j$ or $(i, j) \in E$

      then $t_{ij}^{(0)} = 1$

      else $t_{ij}^{(0)} = 0$

for $k = 1$ to n

  do for $i = 1$ to n

    do for $j = 1$ to n

      do $t_{ij}^{(k)} = t_{ij}^{(k-1)} \vee (t_{ik}^{(k-1)} \wedge t_{kj}^{(k-1)})$

return $T^{(n)}$

< Exercise 2 >

Strongly connected component 를 찾아라 (graph slide 에 있음)
(SCC)