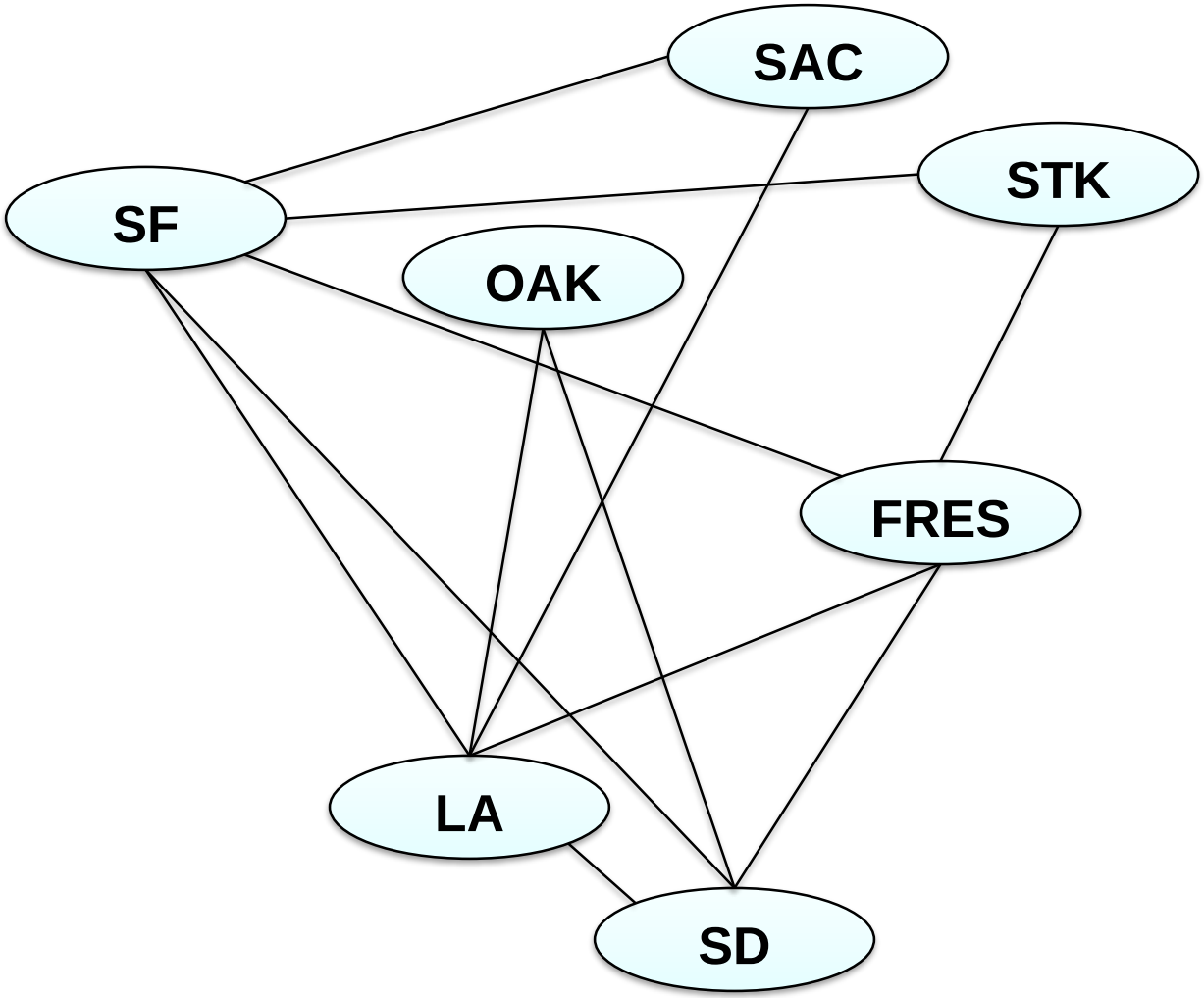
HANDONG GLOBAL UNIVERSITY

Chapter 22
Elementary Graph Algorithms

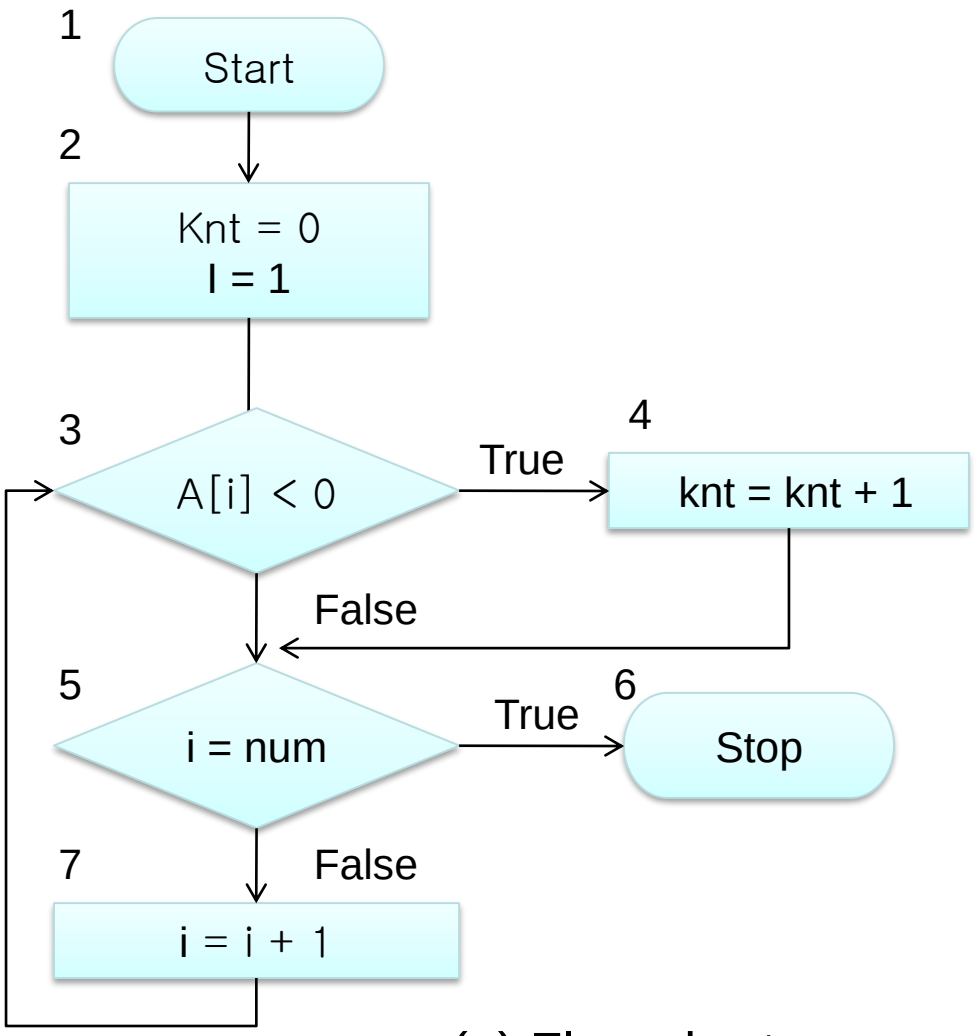
Algorithm Analysis

School of CSEE

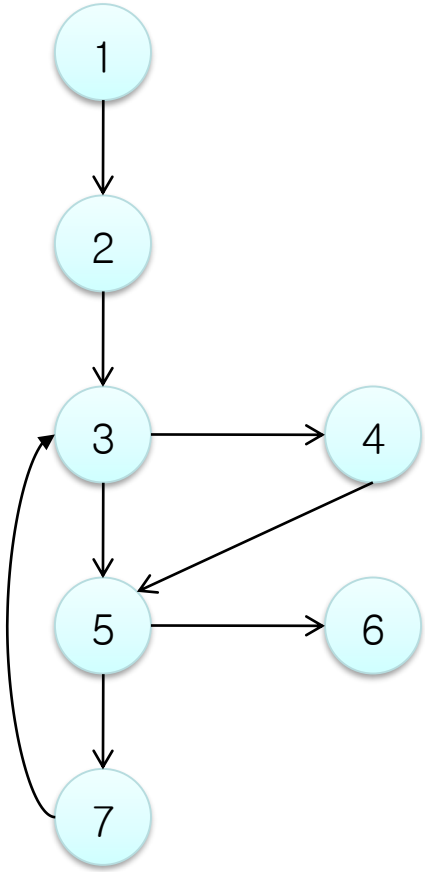
Problems: Airline Routes



Problems: Flowcharts



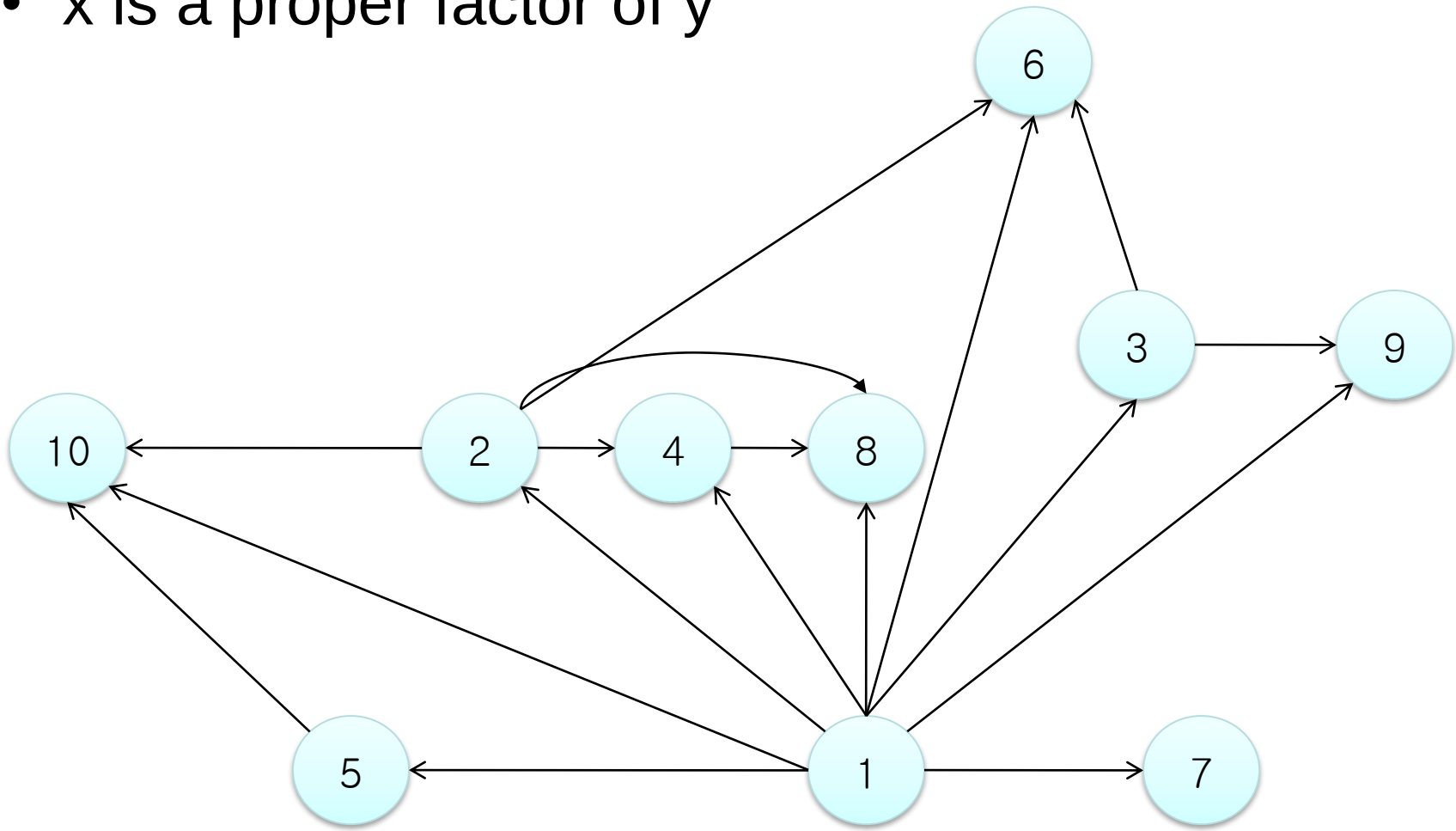
(a) Flow chart



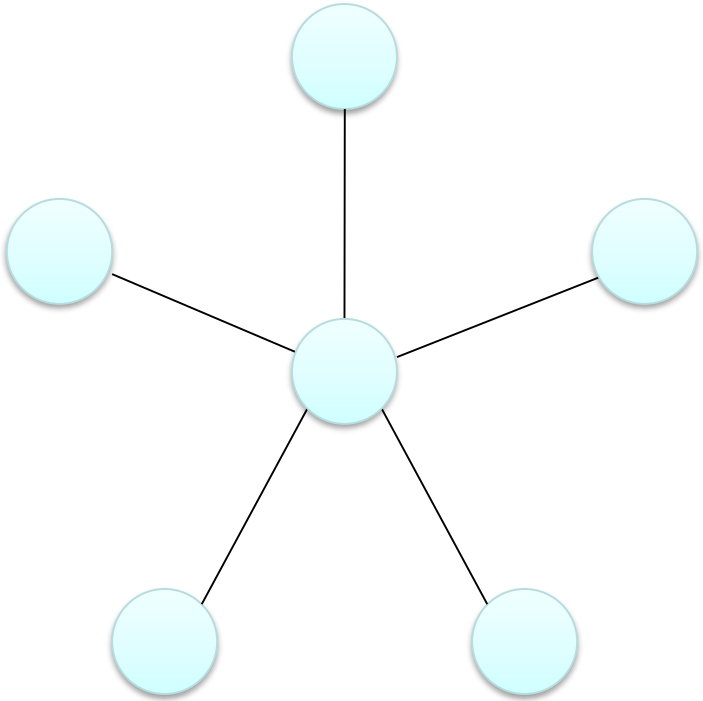
(b) Directed graph

Problems: Binary relation

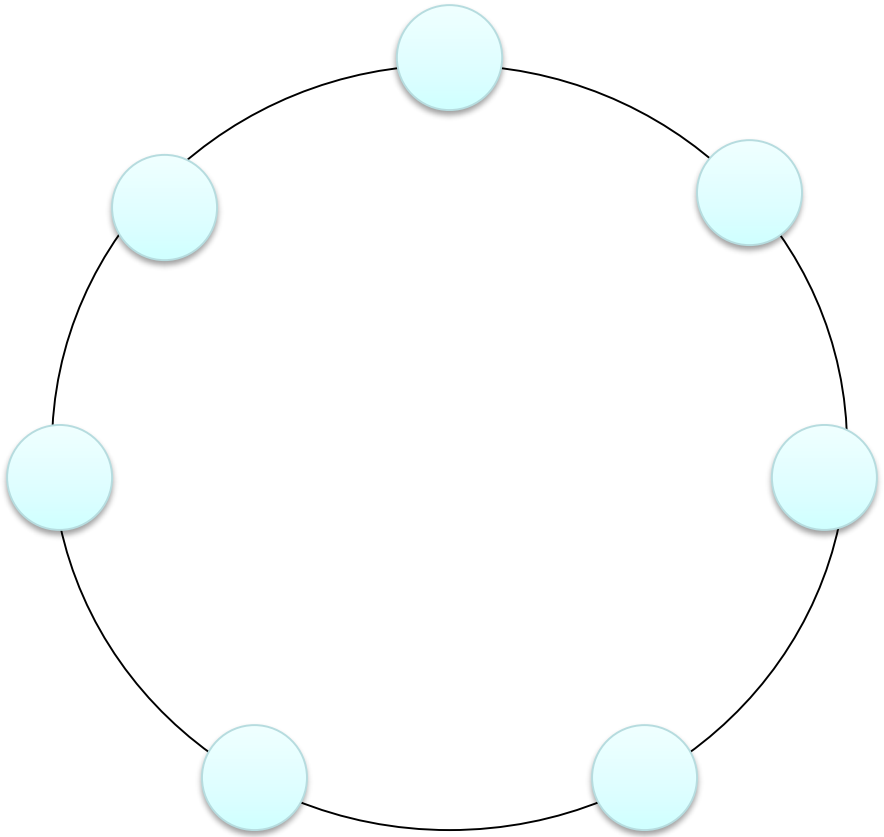
- x is a proper factor of y



Problems: Computer Networks



(a) A star network



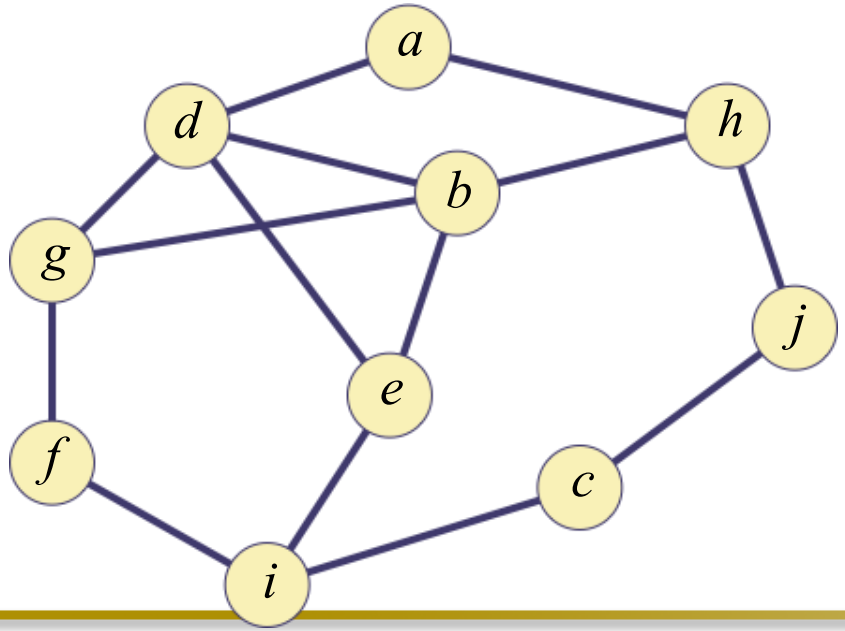
(b) A ring network

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?

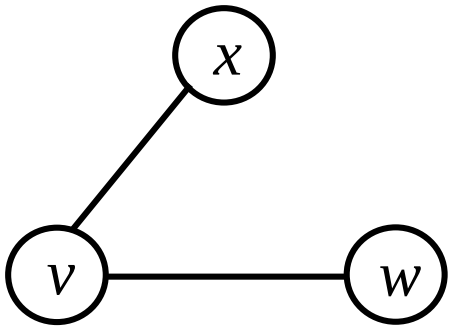
Terms You Should Know or Learn Now

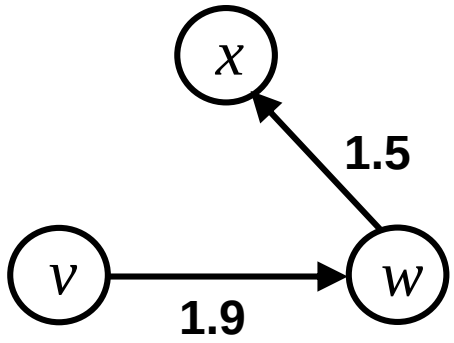
- Vertex (plural *vertices*) or Node
- Edge (sometimes referred to as an *arc*)
  - Note the meaning of *incident*
- Degree of a vertex: how many adjacent vertices
  - Digraph: in-degree (num. of incoming edges) vs. out-degree

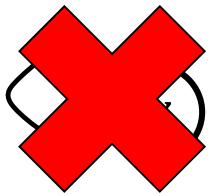


Terms You Should Know or Learn Now

- 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

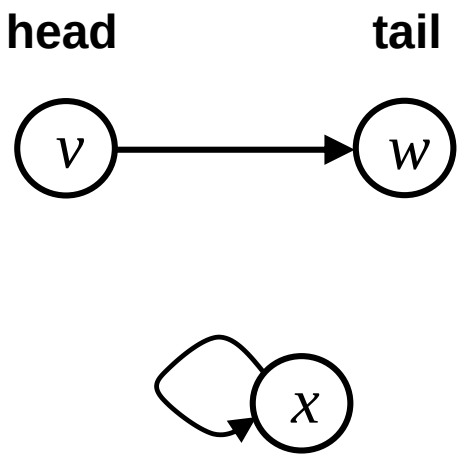


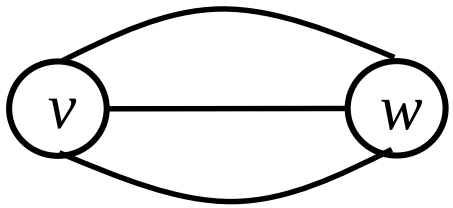




Terms You Should Know or Learn Now

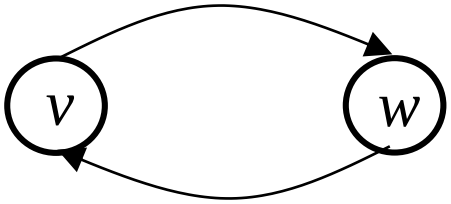
- A directed graph (also known as a *digraph*)
  - "Originating" node is the *head*, the target the *tail*
  - An edge may (or may not) connect a vertex to itself
- A graph may not have multiple occurrences of the same edge.



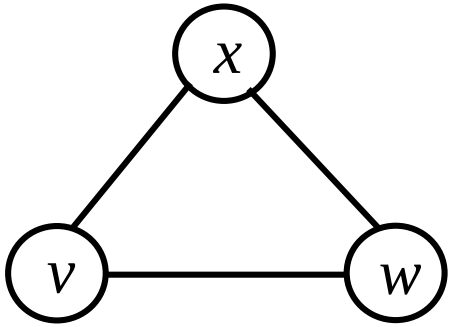


Terms You Should Know or Learn Now

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



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



Terms You Should Know or Learn Now

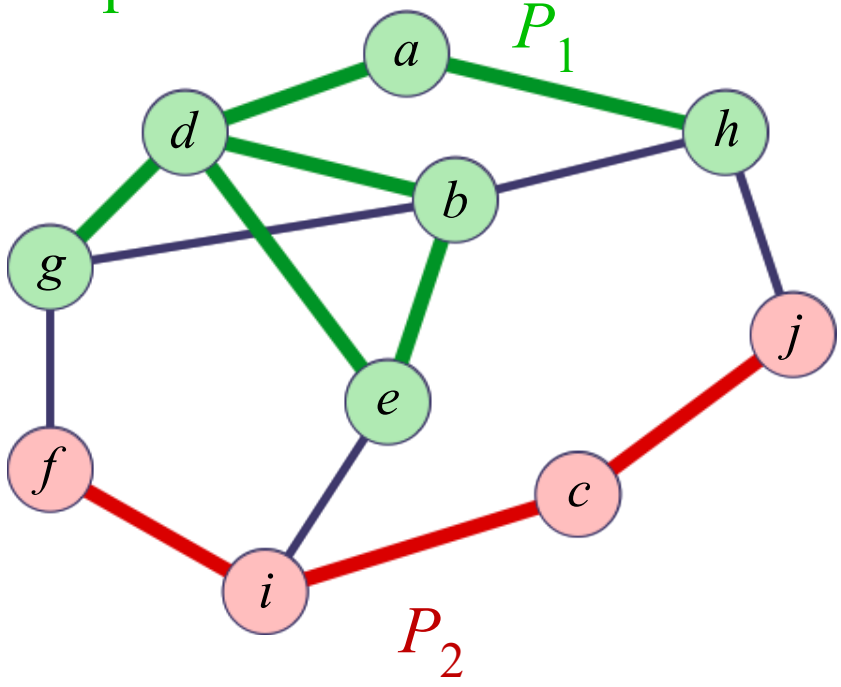
- Size of graph? Two measures:
  - Number of nodes. Usually n
  - Number of edges: Usually m
- Dense graph: many edges
  - Undirected:

    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.

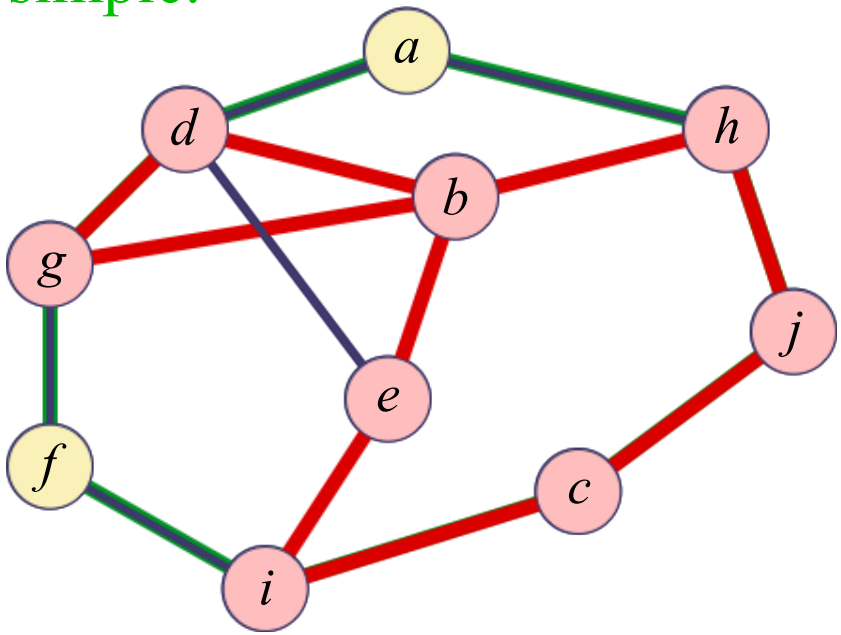


$P_2 = (f, i, c, j)$ is simple.

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.

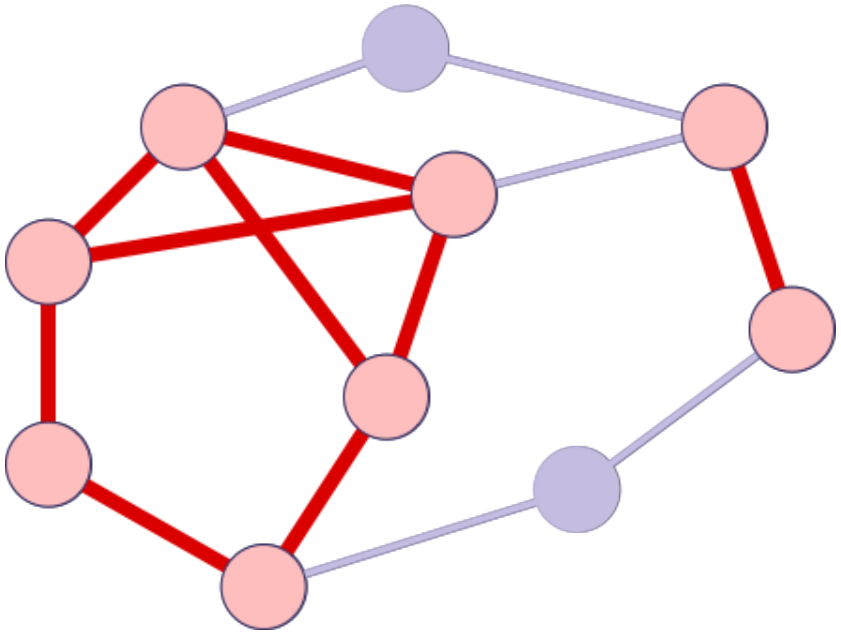
$C_1 = (a, h, j, c, i, f, g, d)$ is simple.



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

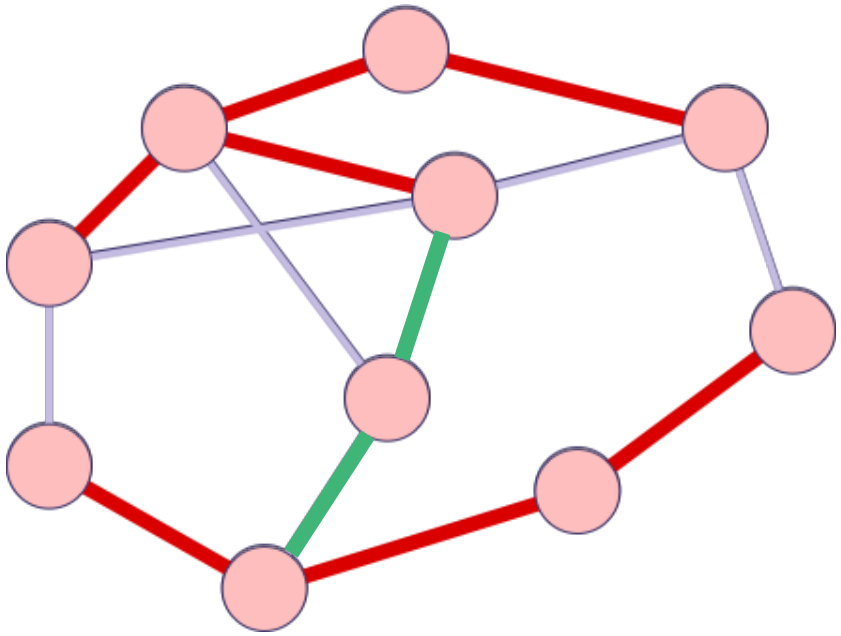
Subgraphs

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



Spanning Graphs

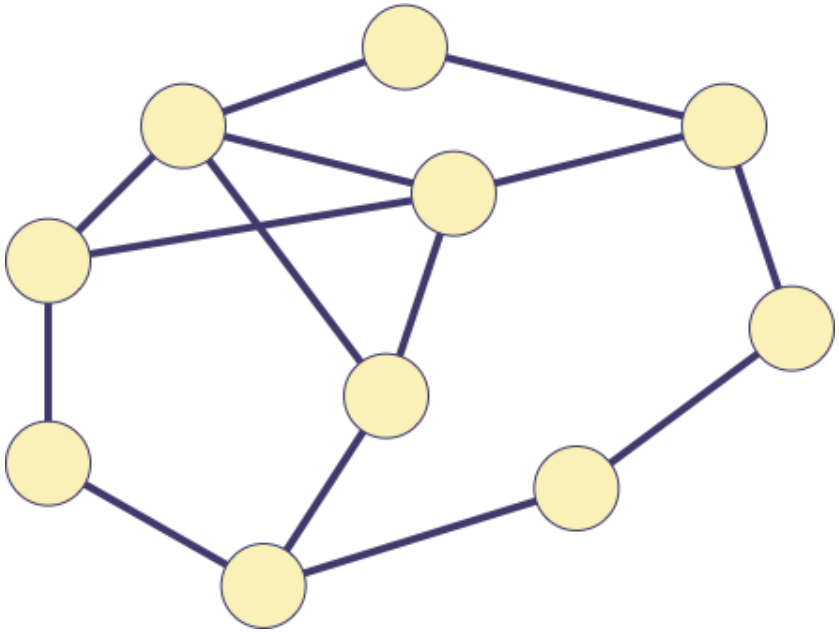
- A *spanning graph* of *G* is a subgraph of *G* that contains all vertices of *G*.

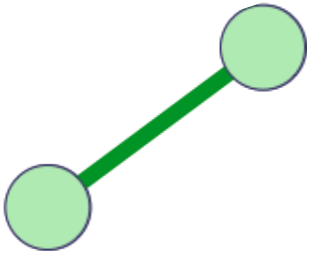


Connectivity : Undirected graph

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.





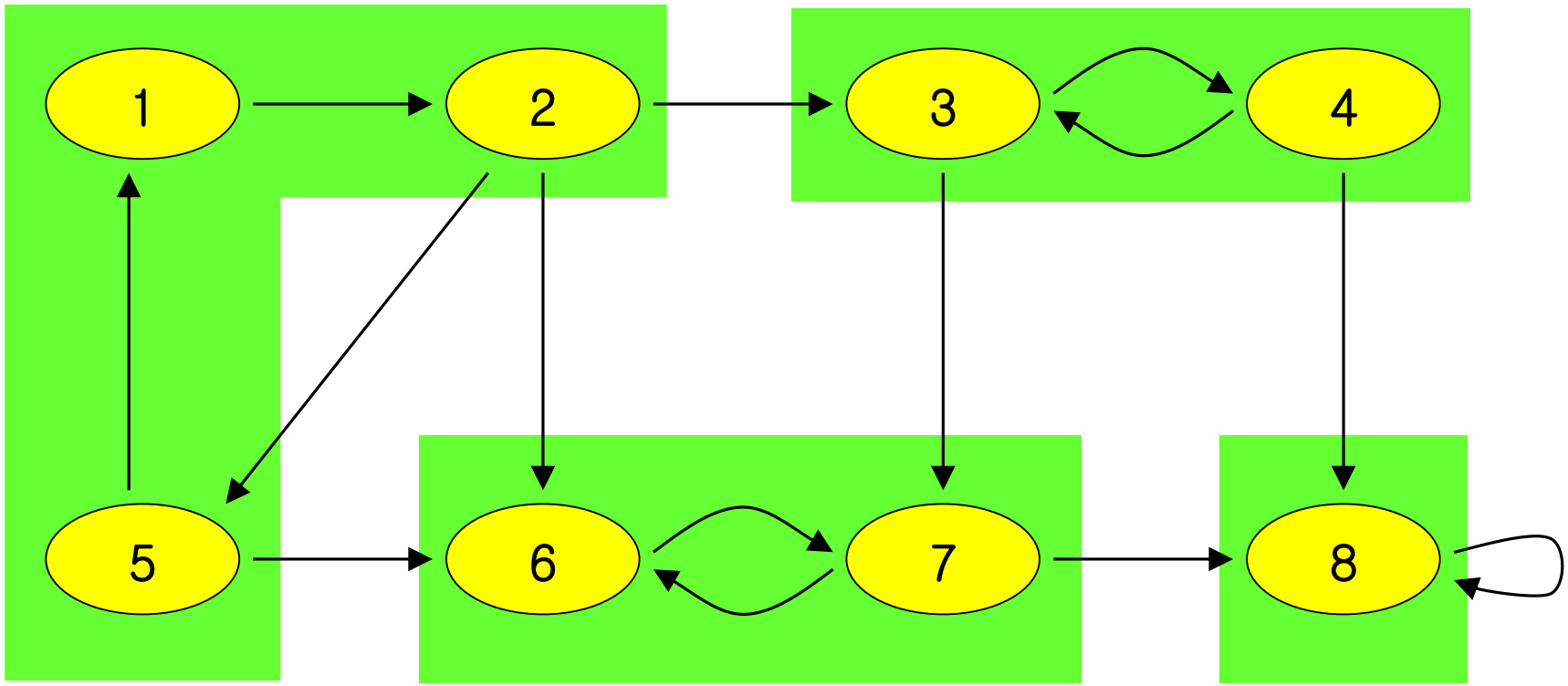
Not a connected component

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
- Directed acyclic graph: a DAG

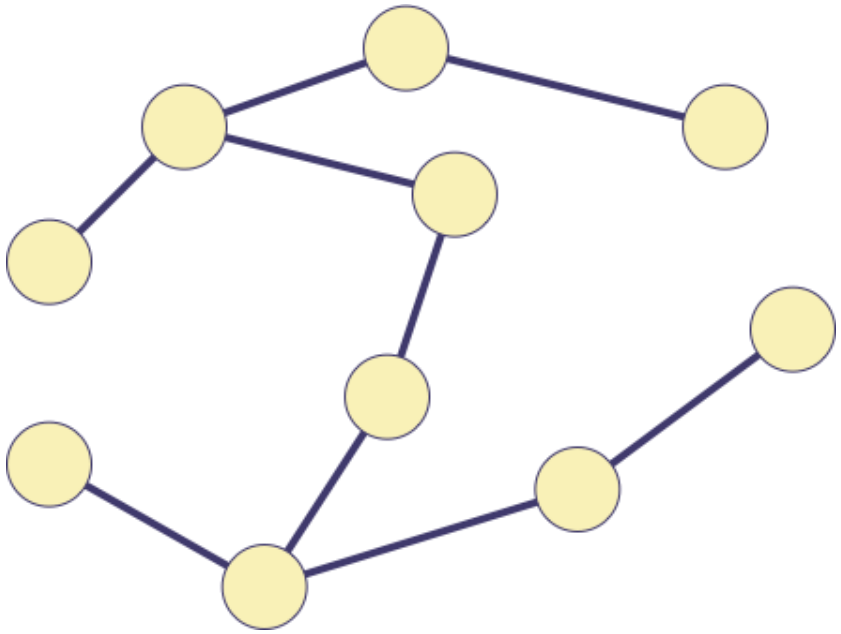
Example



Green areas are the Strongly Connected Components.

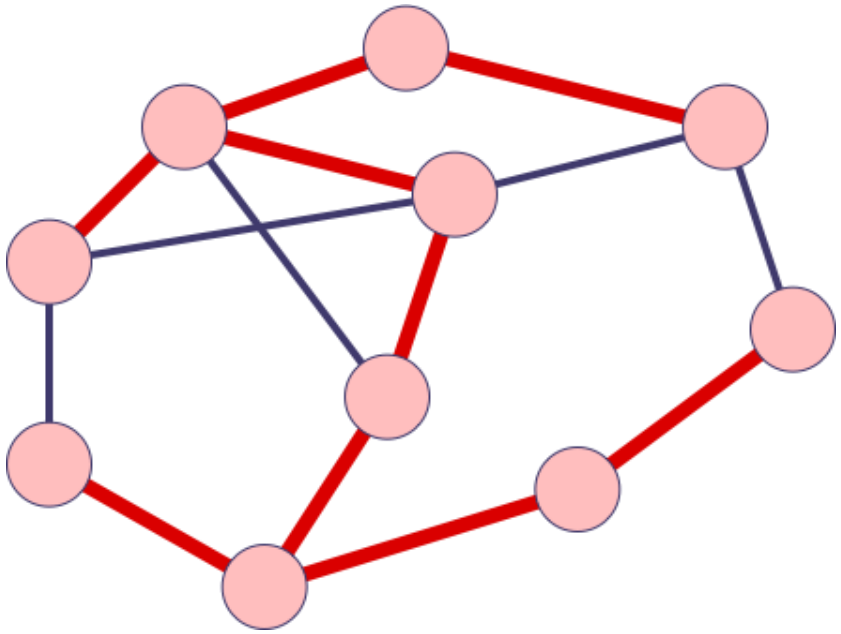
Trees

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



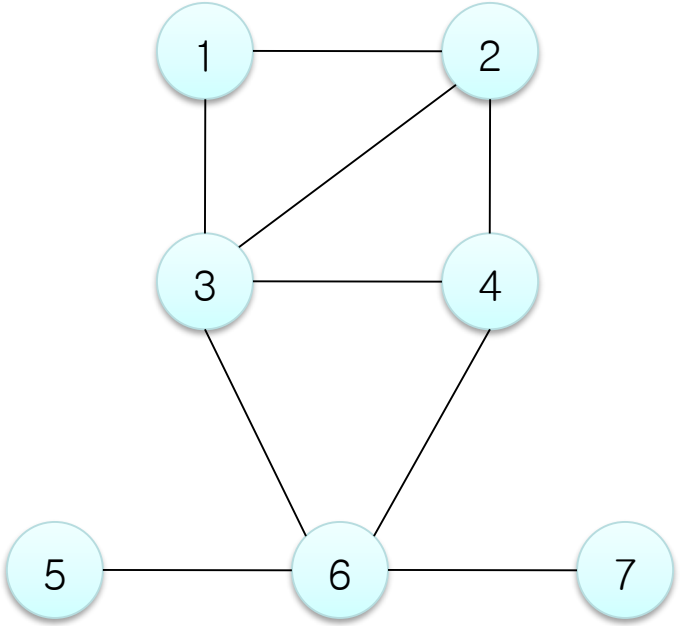
Spanning Trees

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



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

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

(b) Its adjacency matrix

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?*

  A: 6 bits

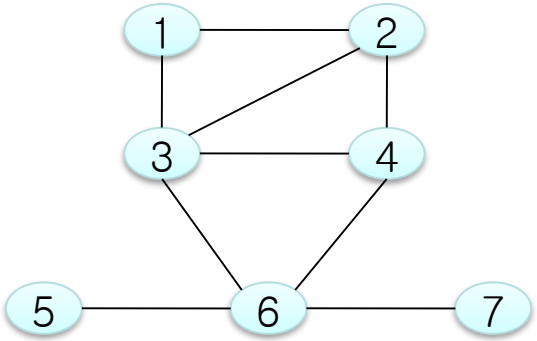
  – Undirected graph $\rightarrow$ matrix is symmetric

  – No self-loops $\rightarrow$ 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

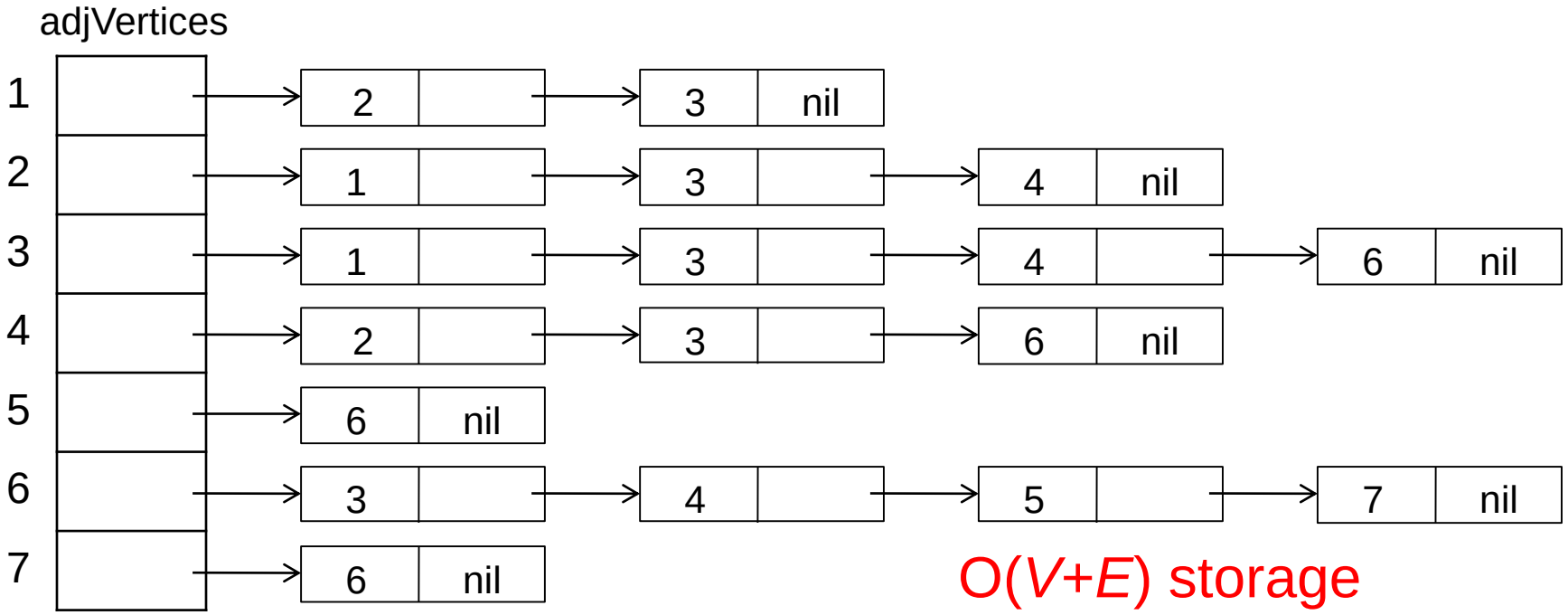
Array of Adjacency Lists Representation



(a) An undirected graph

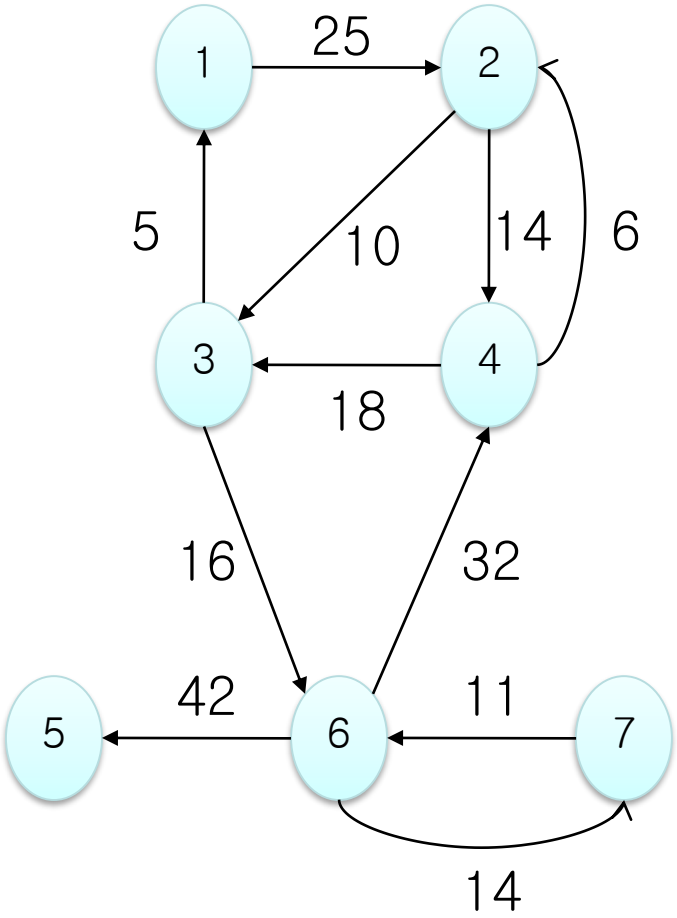
$$\begin{pmatrix} 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix}$$

(b) Its adjacency matrix



O(*V*+*E*) storage

Adjacency Matrix for weight digraph

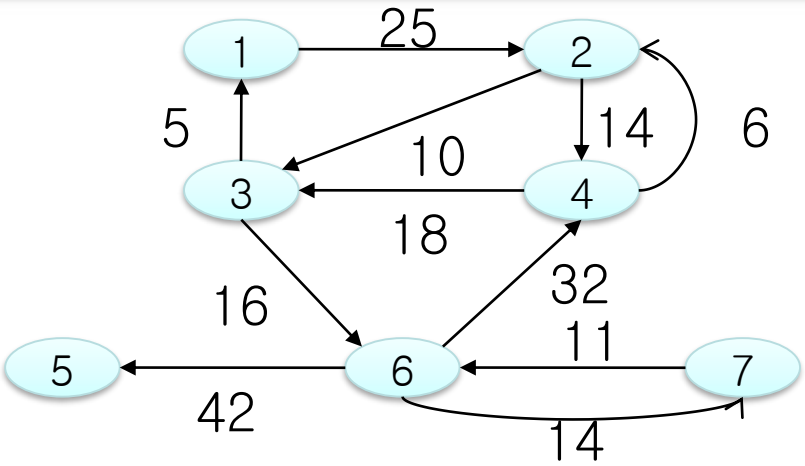


(a) A weighted digraph

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

(b) Its adjacency matrix

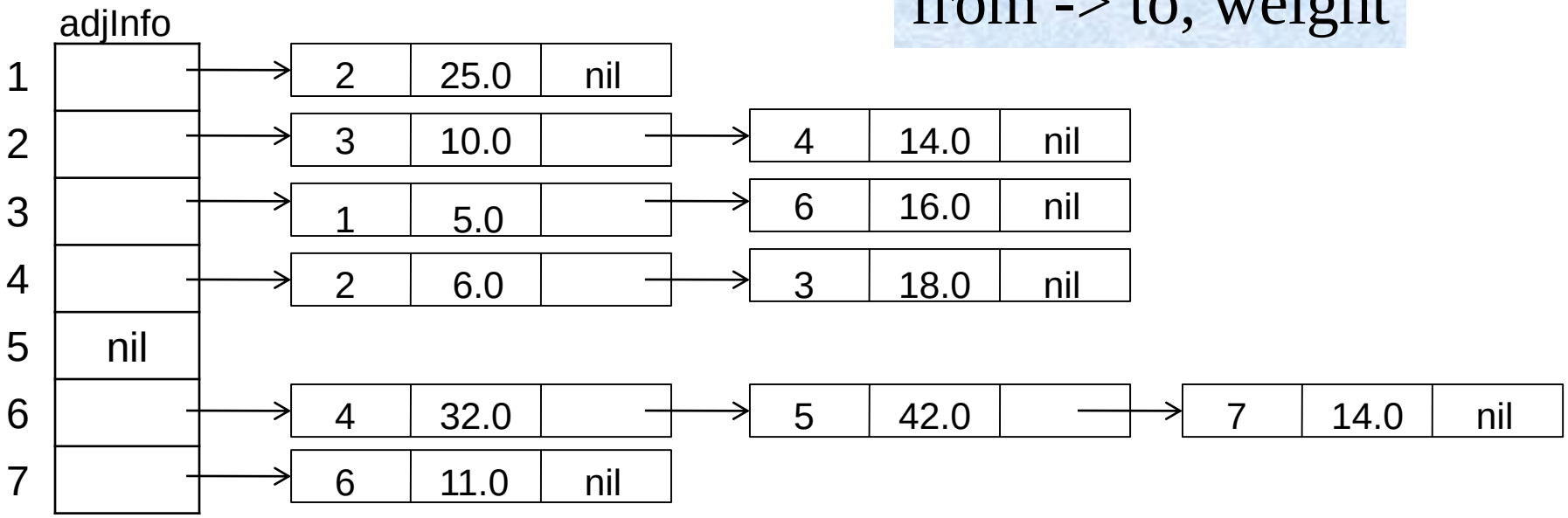
Array of Adjacency Lists Representation



(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’
  - Tree with discovered vertices and explored edges.
  - One vertex at a time
  - Pick a *source vertex, s* to be the root.
  - Find (“discover”) its children, then their children, etc.
  - 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.

* ‘Explore’ the edge : The edge is first examined.

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.
    - 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;
  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:

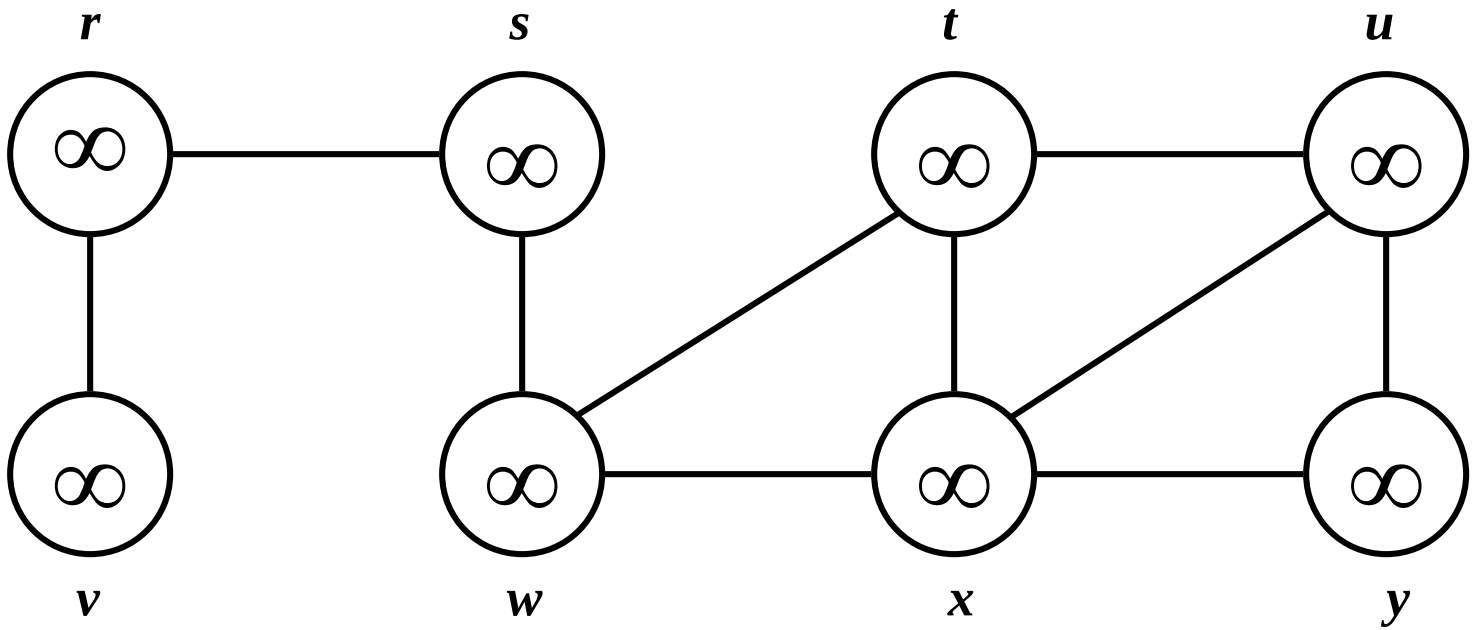
A. Each node is enqueued once. (white → gray) : $\Theta(V)$

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

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

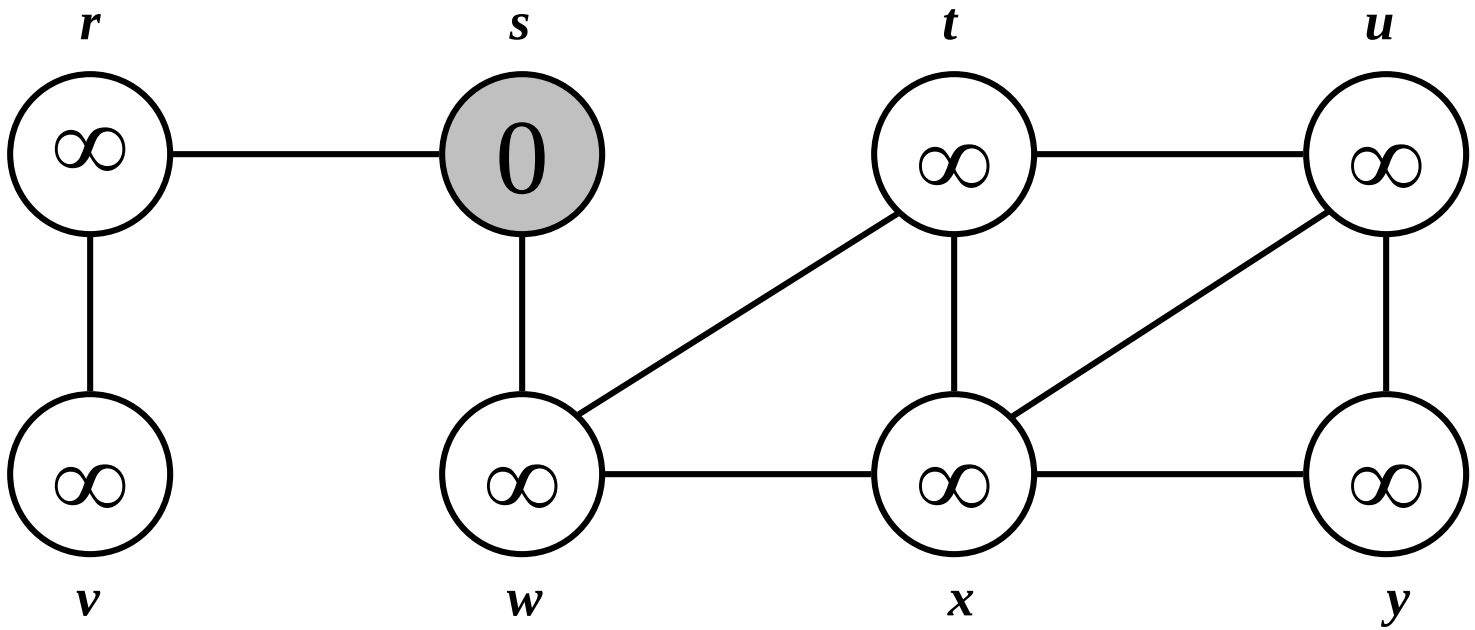
Overall running time is $\Theta(V+E)$.

Breadth-First Search: Example



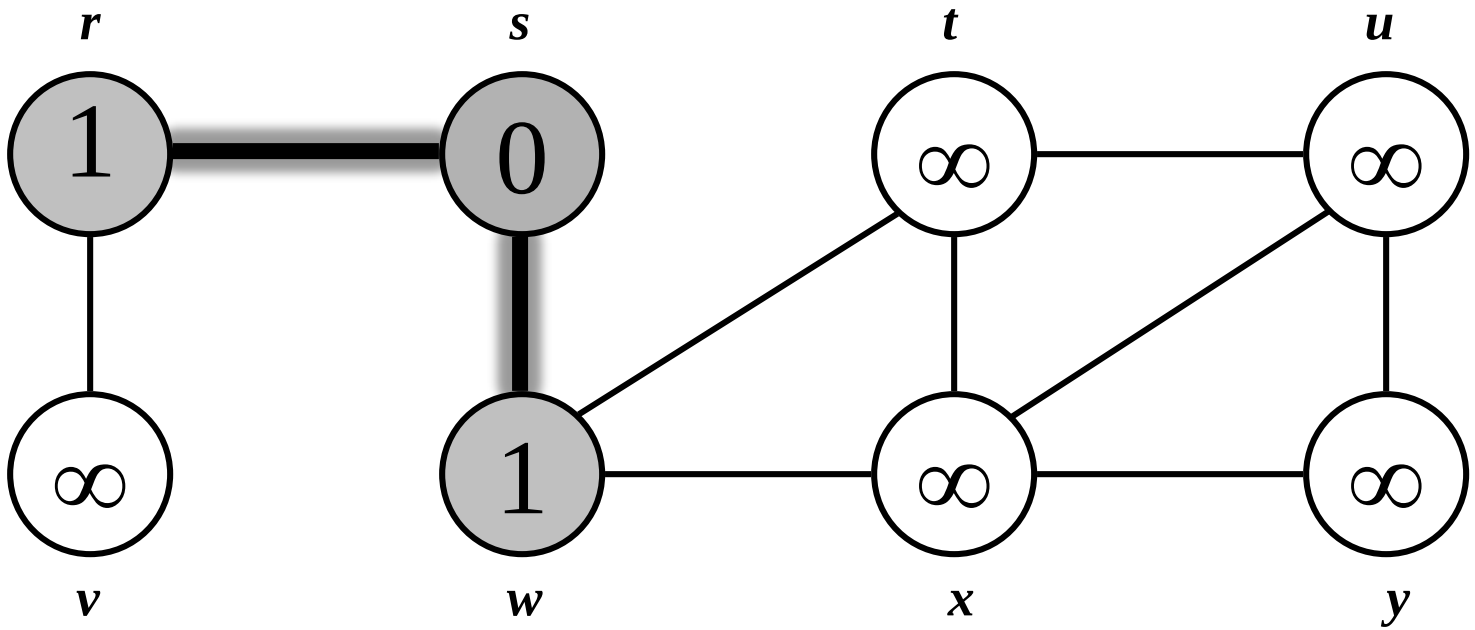
Vertex is an island and edge is an bridge. BFS is visualized as many simultaneous (or nearly simultaneous) explorations from the starting point and spreading out independently.

Breadth-First Search: Example



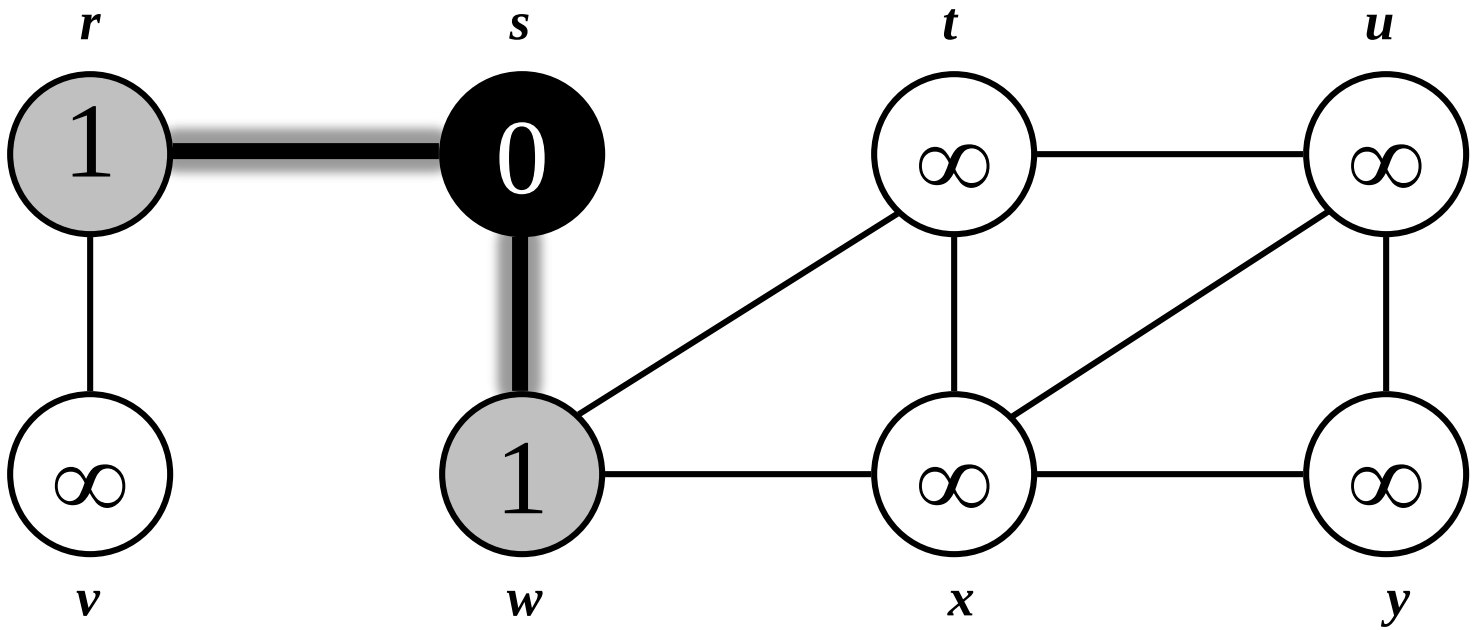
Q: | *s* |

Breadth-First Search: Example



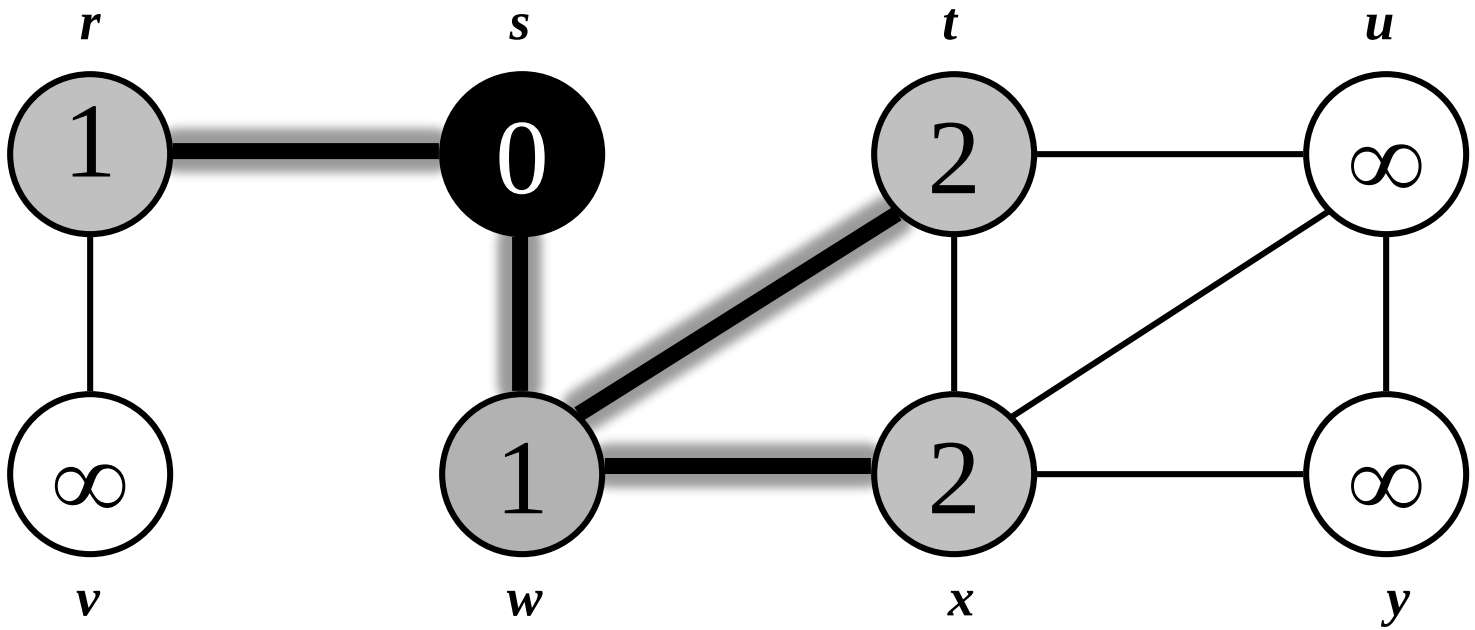
Q: | *w* | *r* |

Breadth-First Search: Example



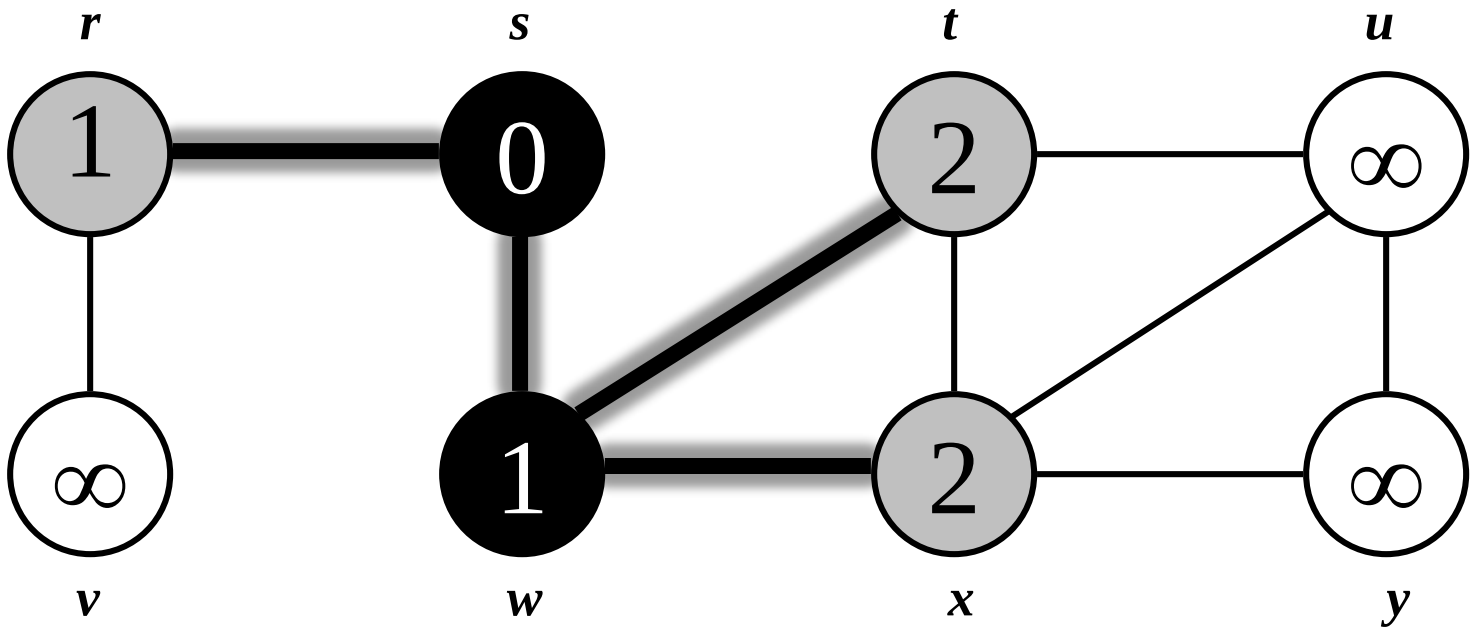
Q: | *w* | *r* |

Breadth-First Search: Example



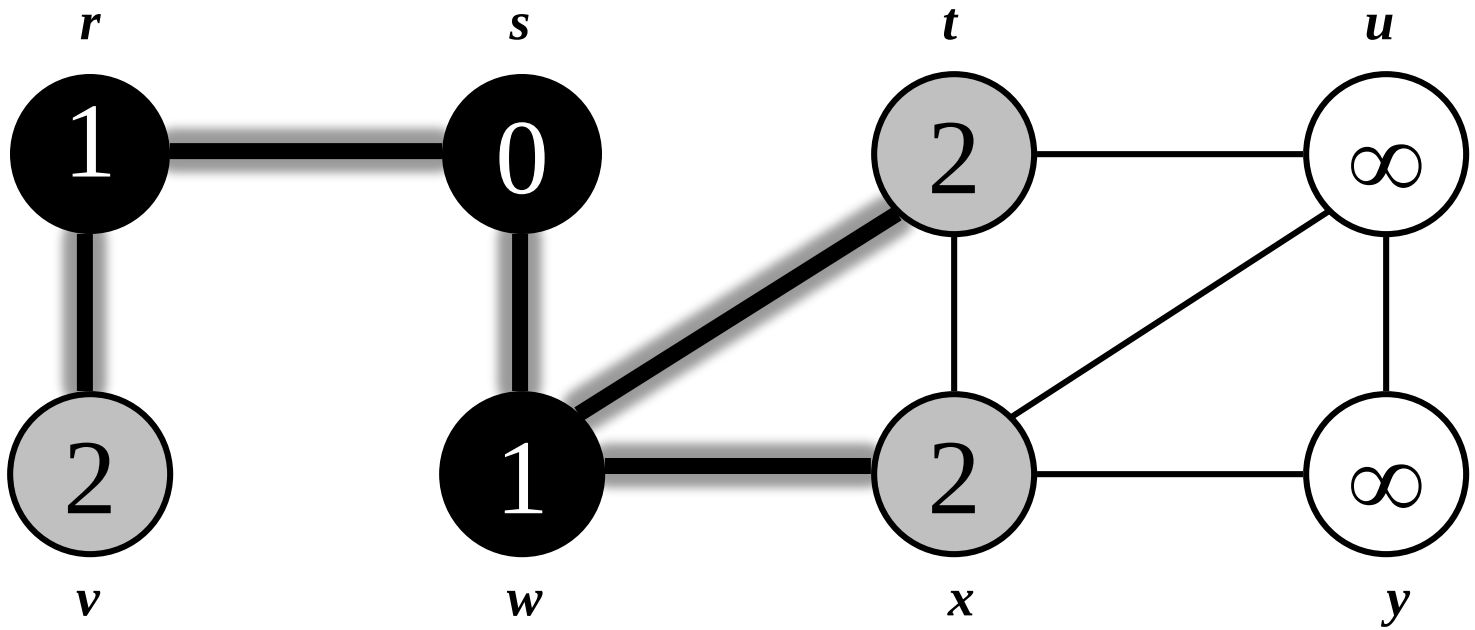
Q: | *r* | *t* | *x* |

Breadth-First Search: Example



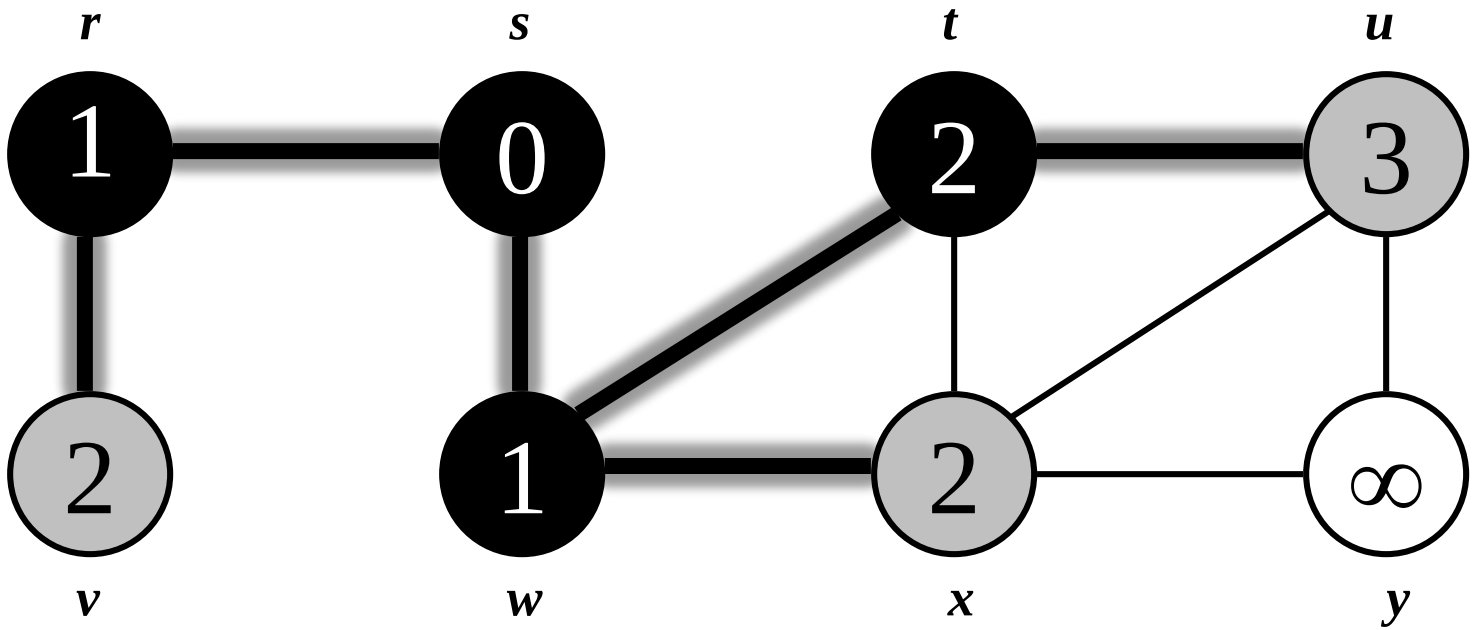
Q: | *r* | *t* | *x* |

Breadth-First Search: Example



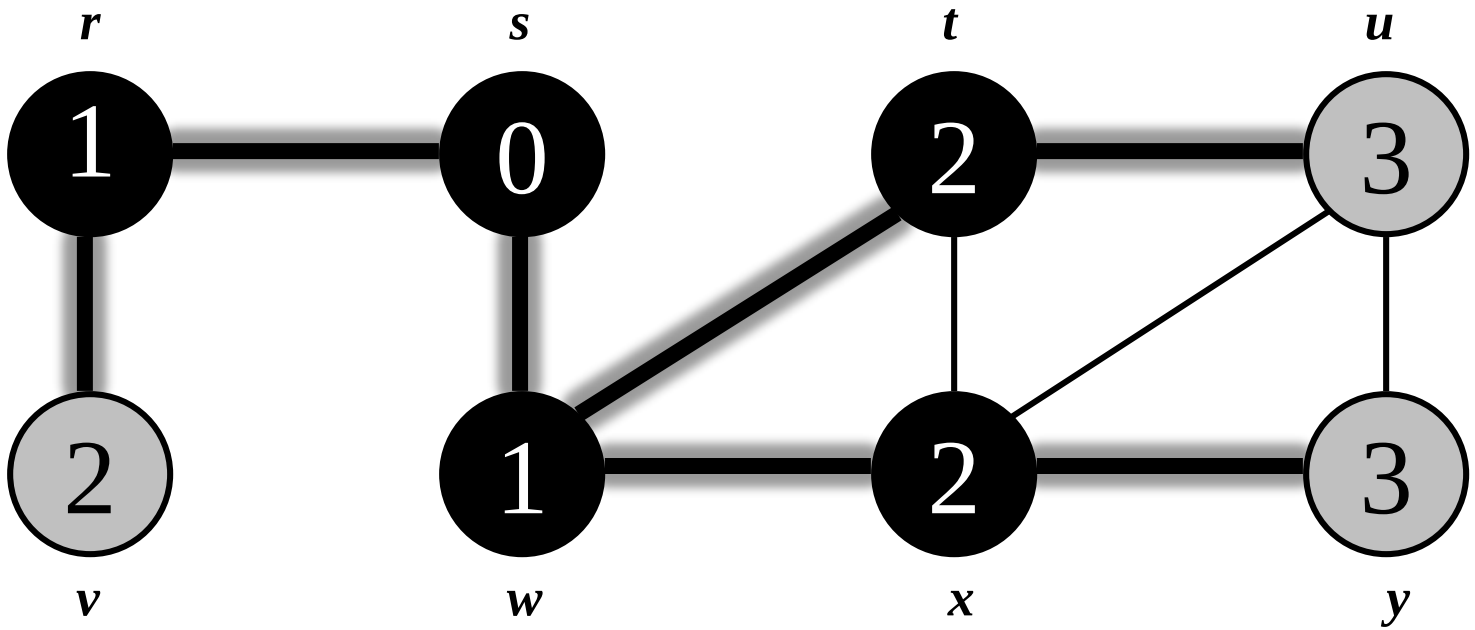
Q: | *t* | *x* | *v* |

Breadth-First Search: Example



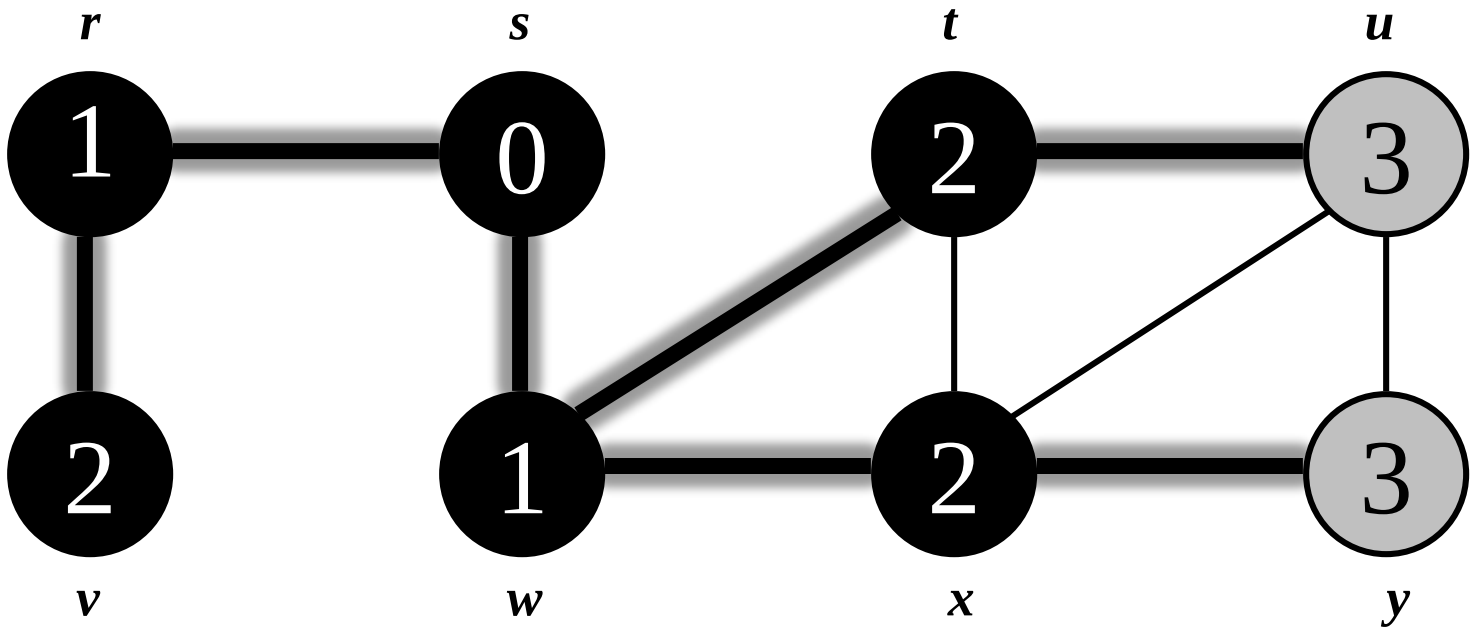
Q: | *x* | *v* | *u* |

Breadth-First Search: Example



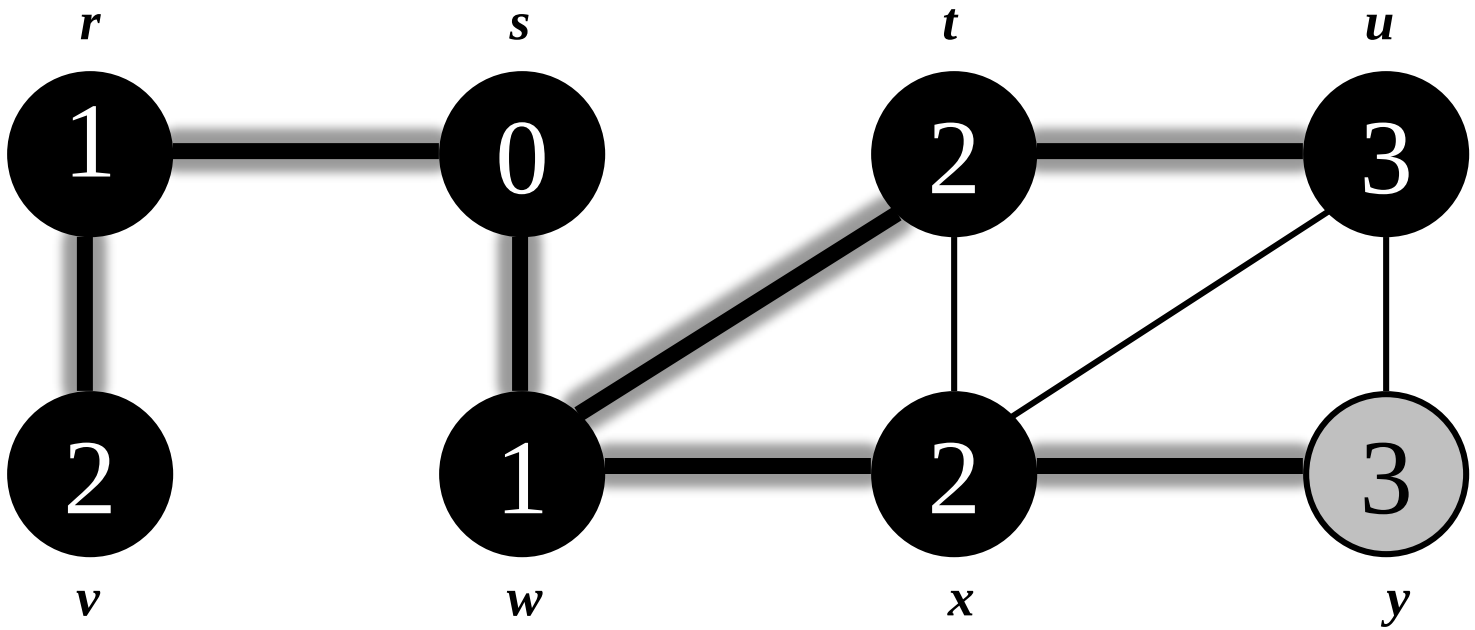
Q: | *v* | *u* | *y* |

Breadth-First Search: Example



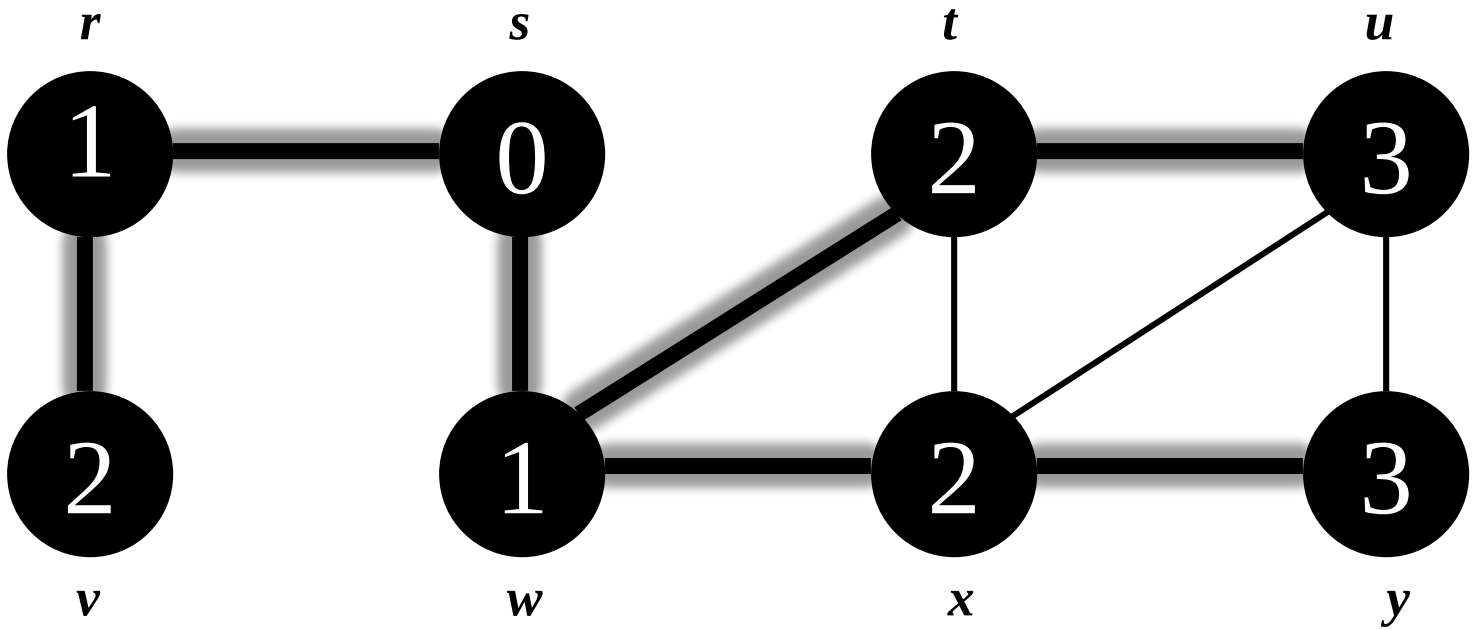
Q: | *u* | *y* |

Breadth-First Search: Example



Q: y

Breadth-First Search: Example



breadth-first tree :

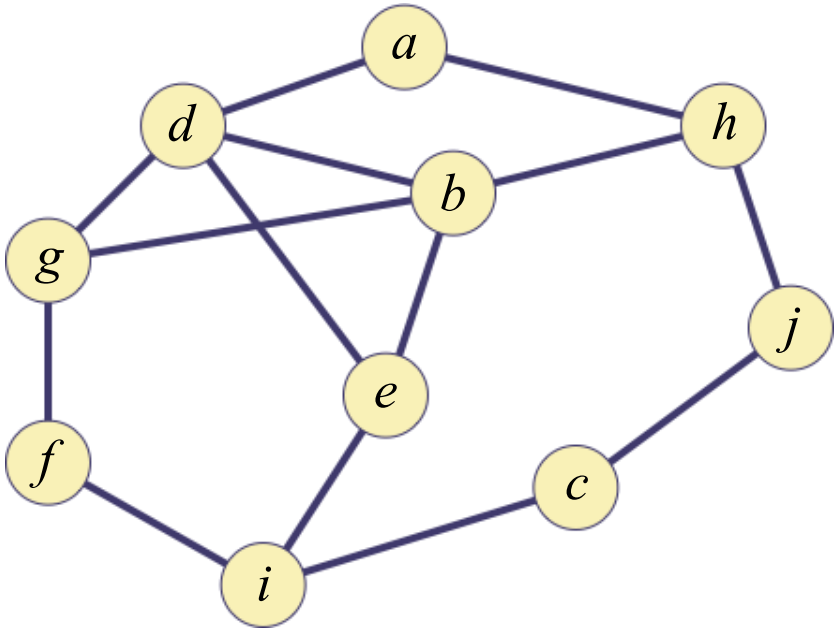
$Q:$ $\emptyset$ consists of vertices and explored edges

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.

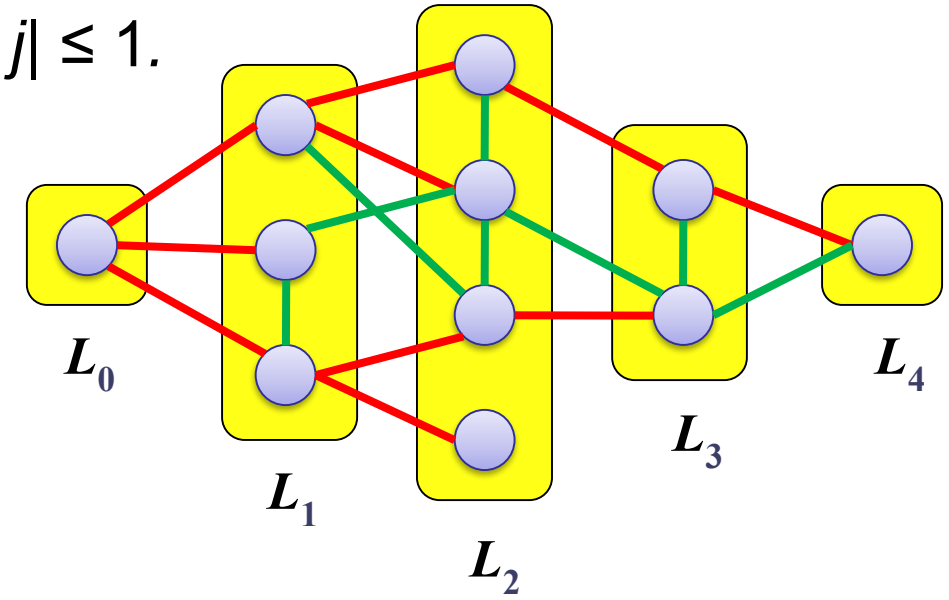
2) What is the minimum distance of vertex *c* from the root?



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* $w \in L_j$, $|i - j| \leq 1$.



DFS

Depth-first search

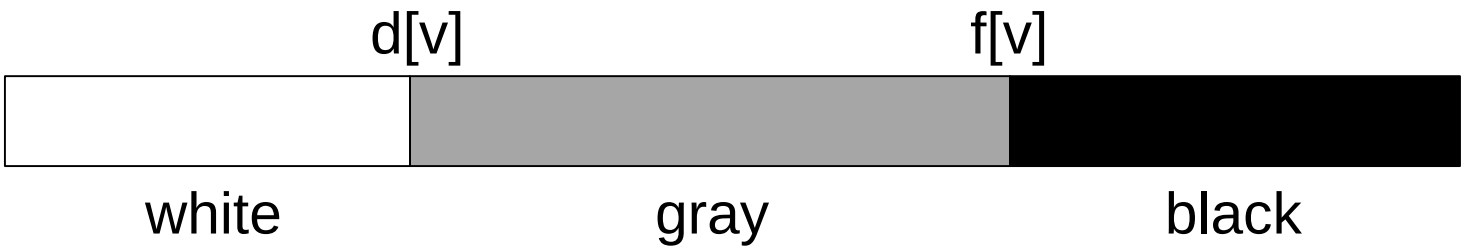
- *Depth-first search* is another strategy for exploring a graph.
  - Explore "deeper" in the graph whenever possible.
  - Pick a *source vertex, s* to be the root.
  - Edges are explored out of the most recently discovered vertex *v* that still has unexplored edges.
  - When all of *v*'s edges have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

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, $d[v] < f[v]$.
  - $1 \le d[v] < f[v] \le 2|V|$



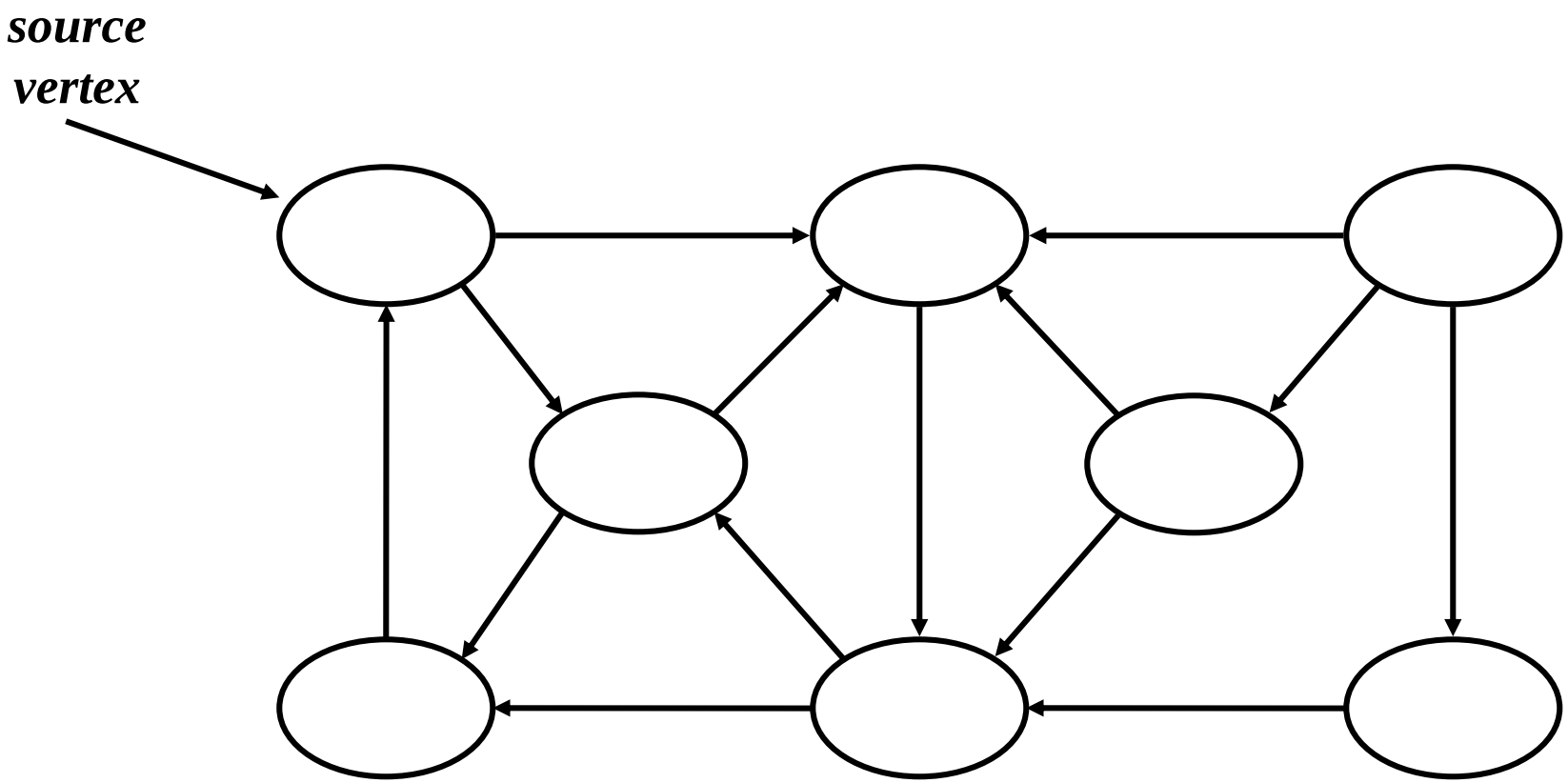
Depth-first search

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

- uses a global timestamp *time*

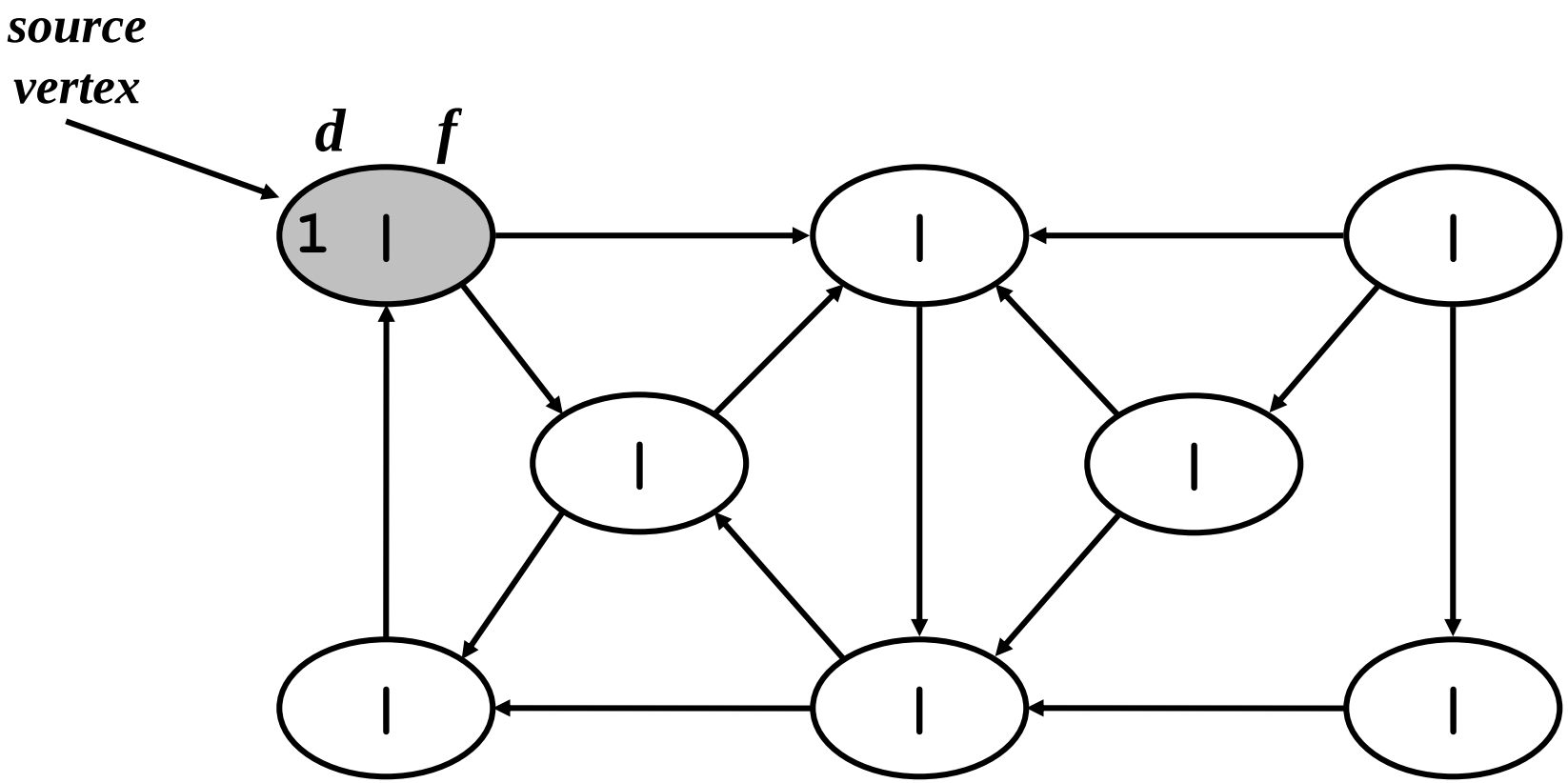
```
DFS-VISIT(u)
color[u] = GRAY
time = time + 1
d[u] = time
for each v ∈ Adj[u]
           do if color[v] = WHITE
                 then DFS-VISIT(v)
color[u] = BLACK
time = time + 1
f[u] = time
```

DFS Example

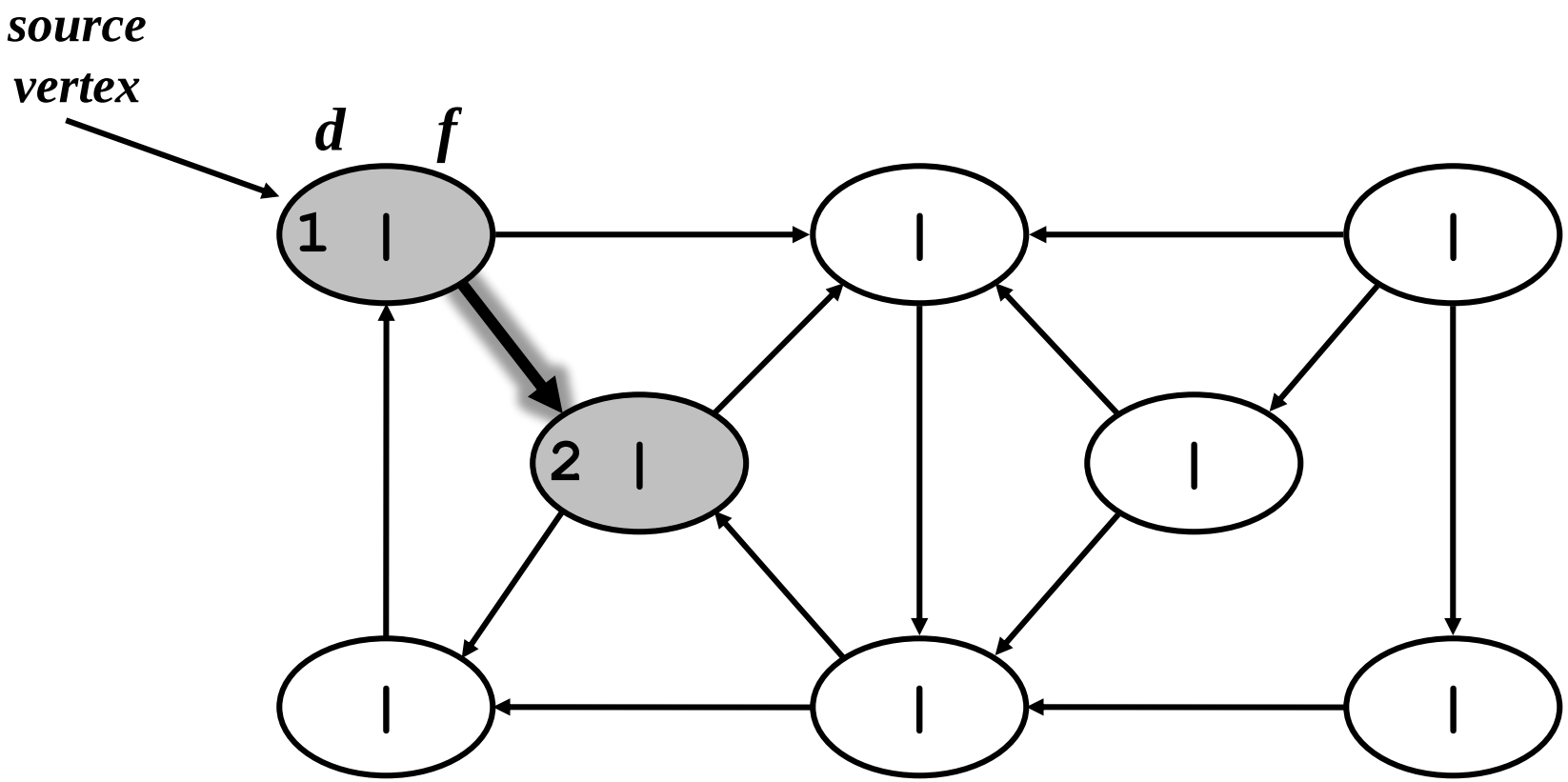


One person explores the islands. If an island is not discovered yet, walk in the direction as arrows. If one walk across the bridge in the reverse direction, one should walk backward – backtracking.

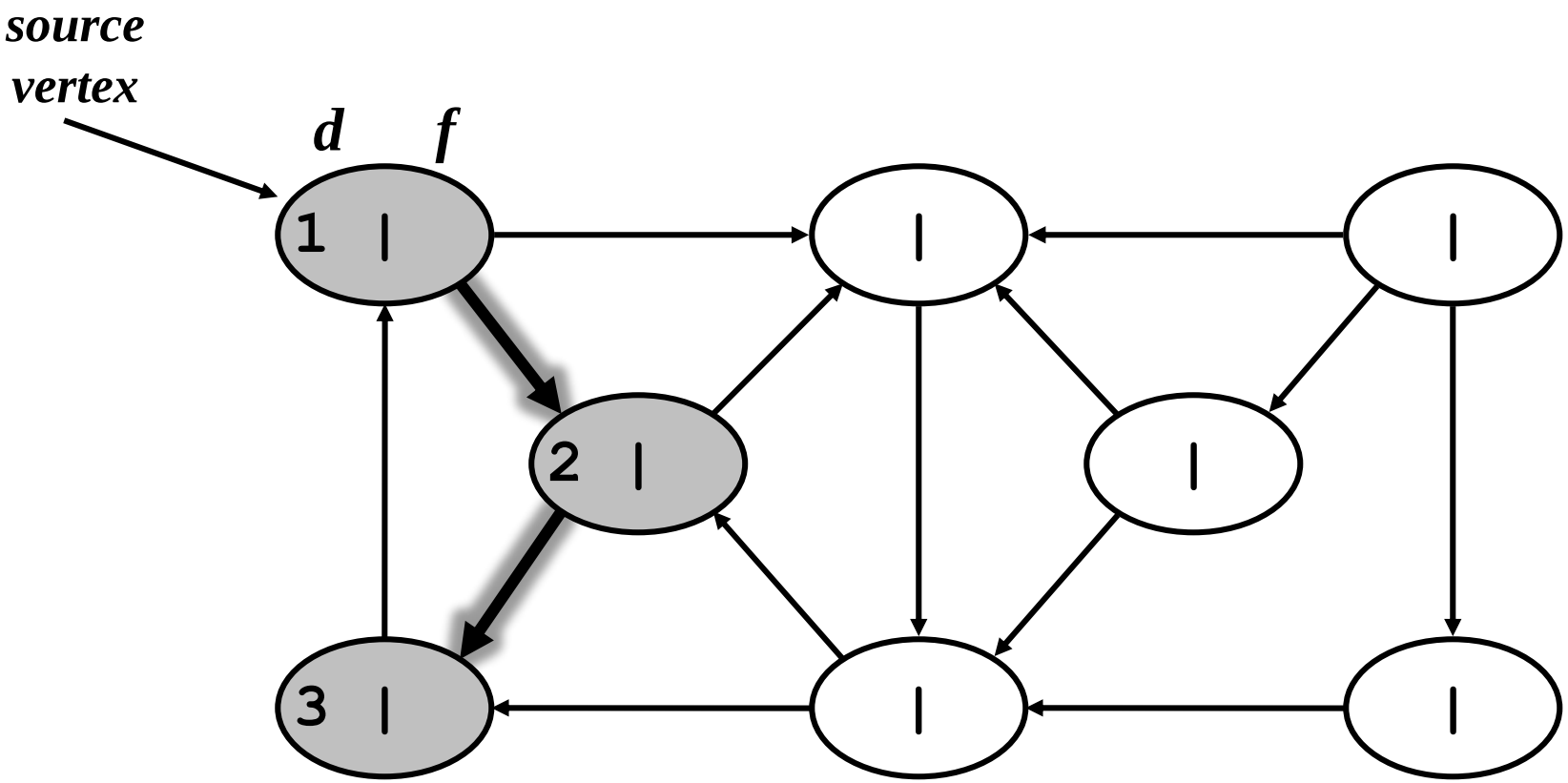
DFS Example



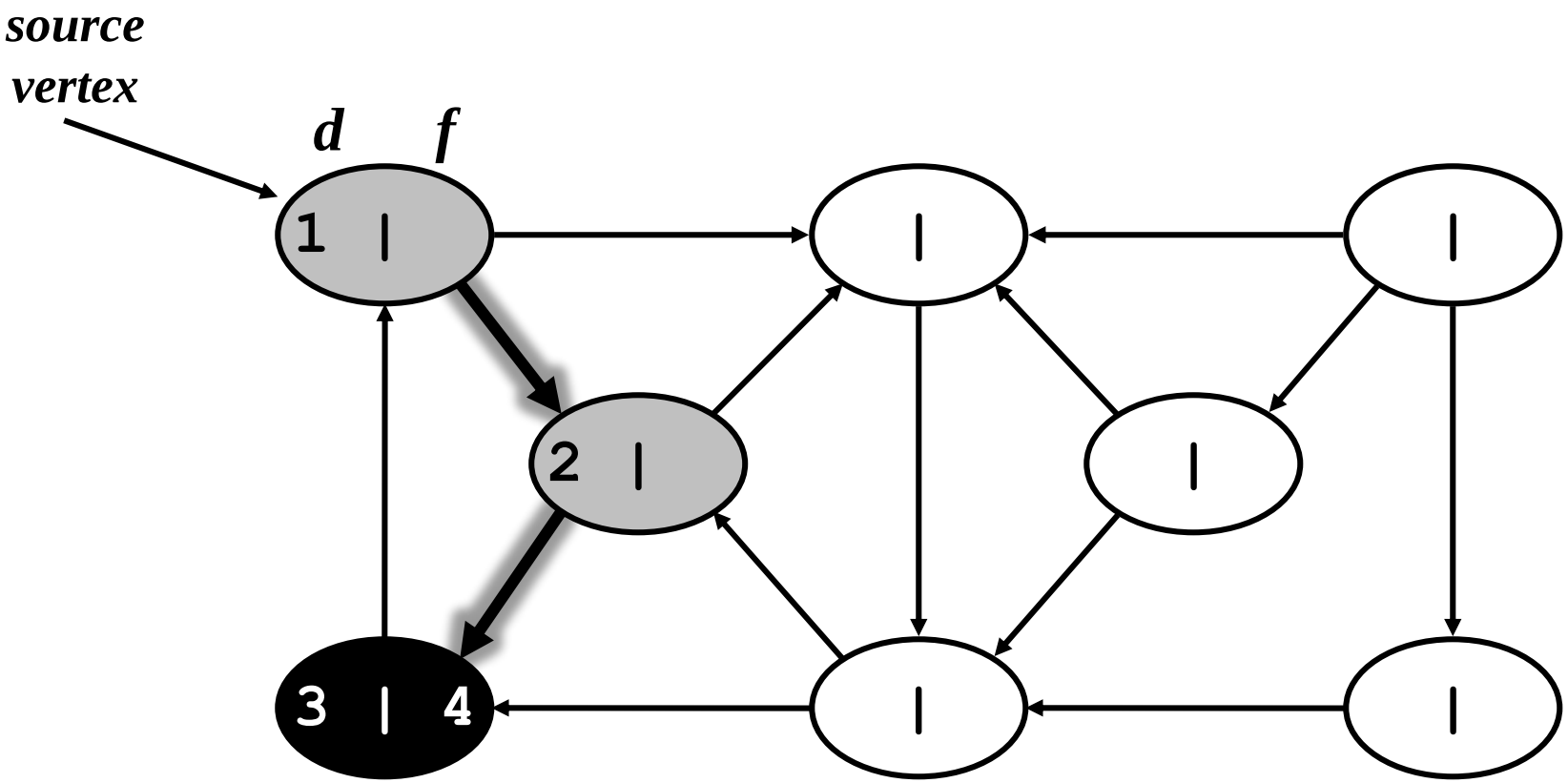
DFS Example



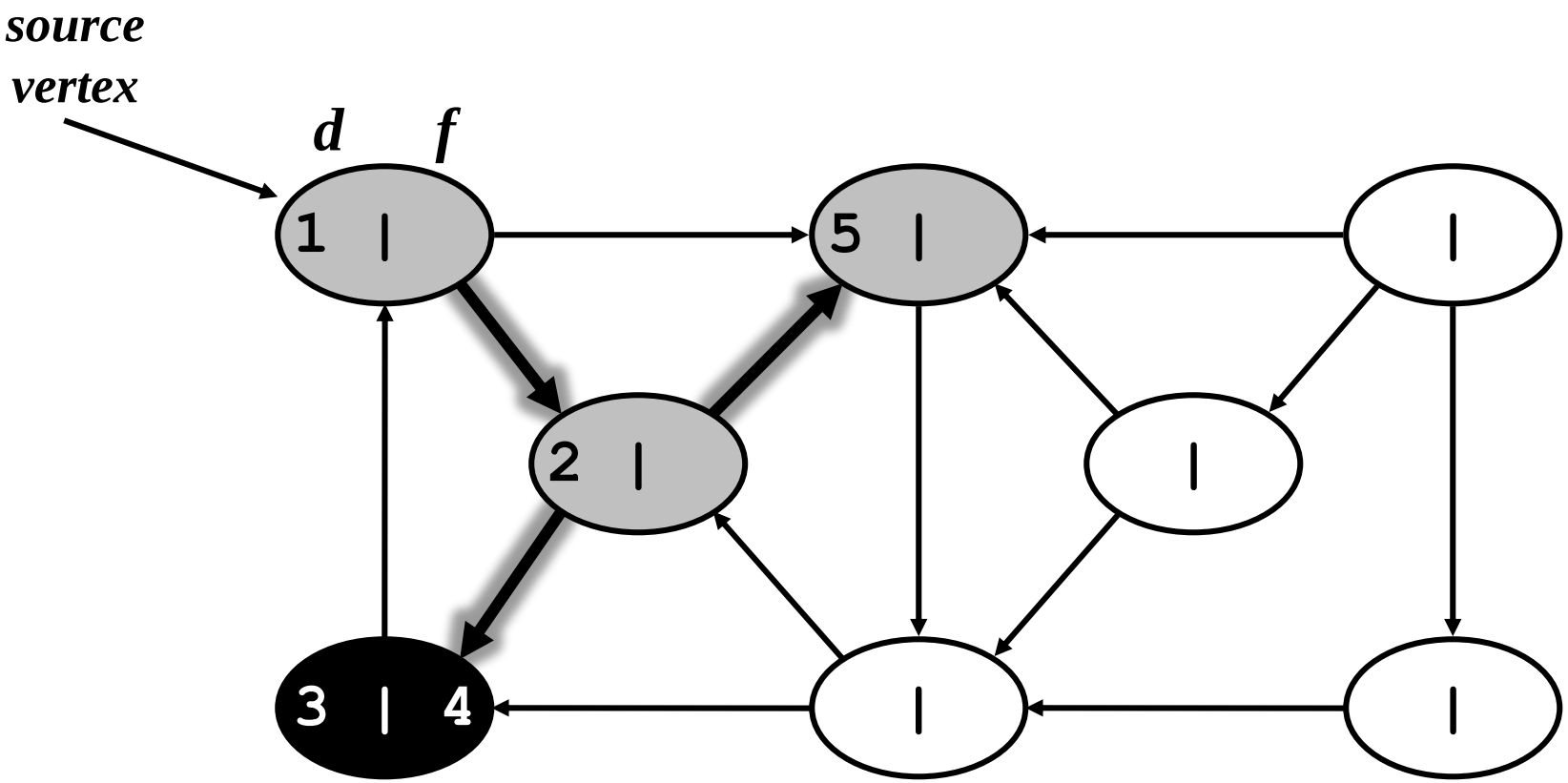
DFS Example



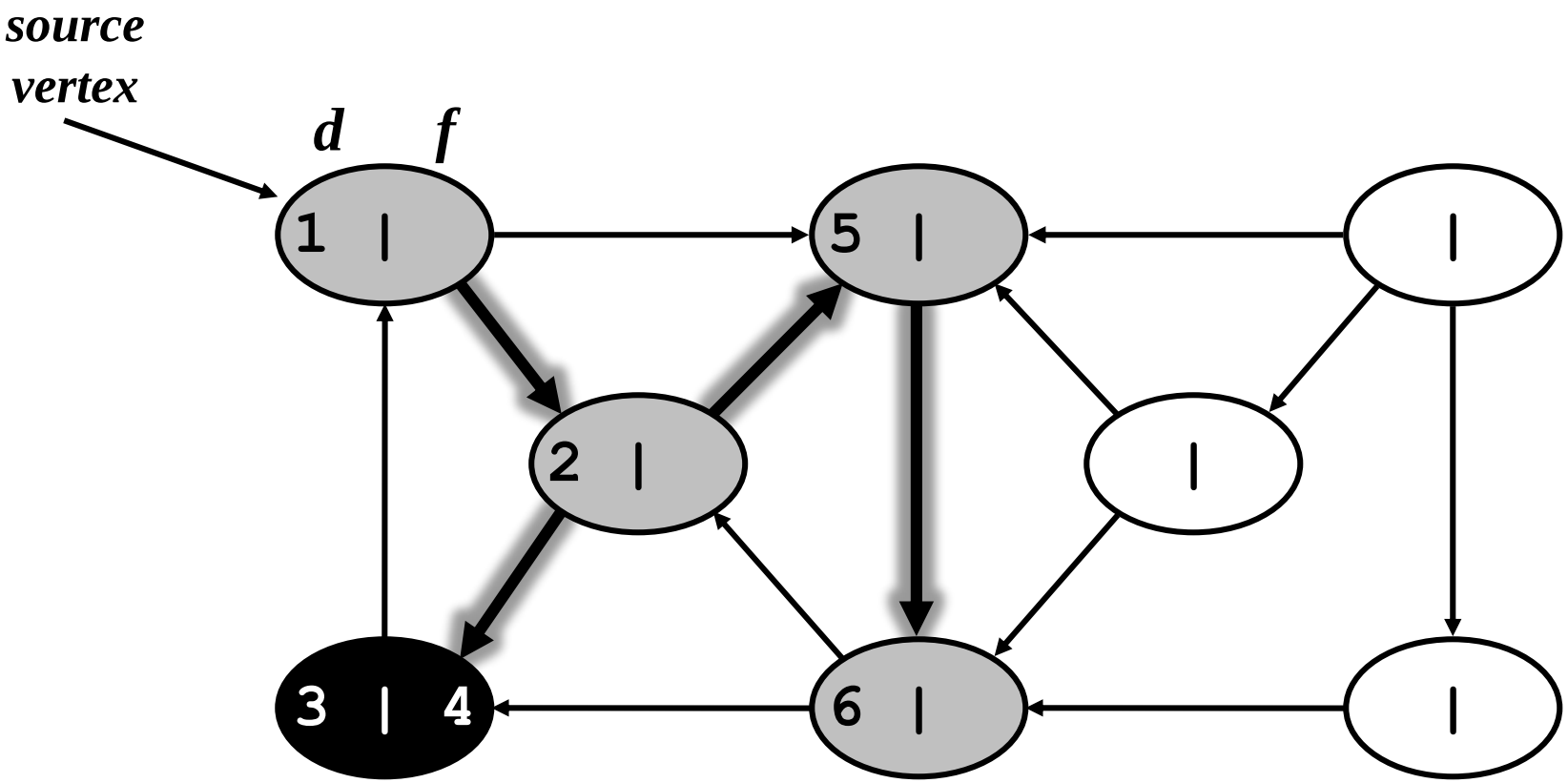
DFS Example



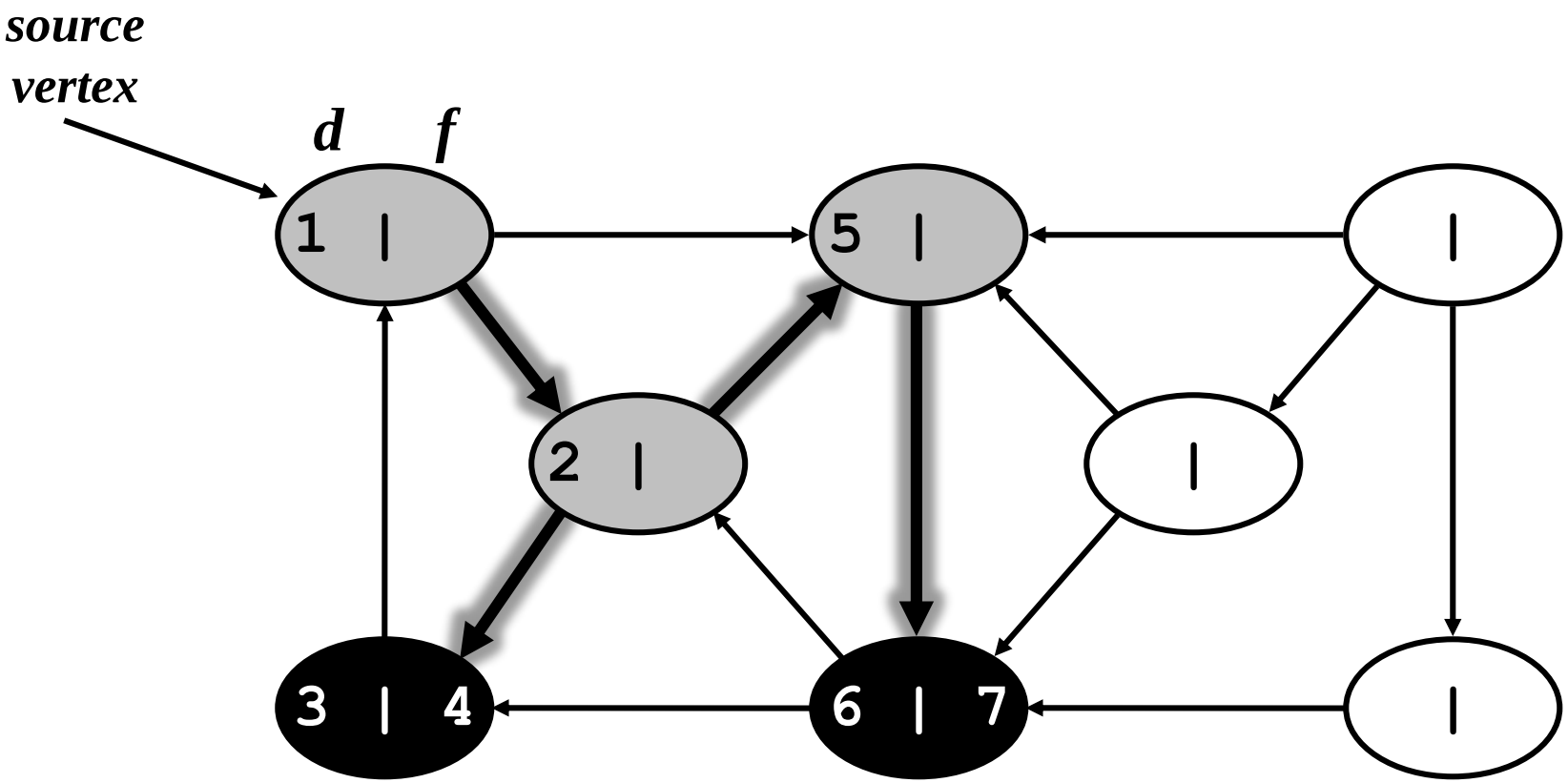
DFS Example



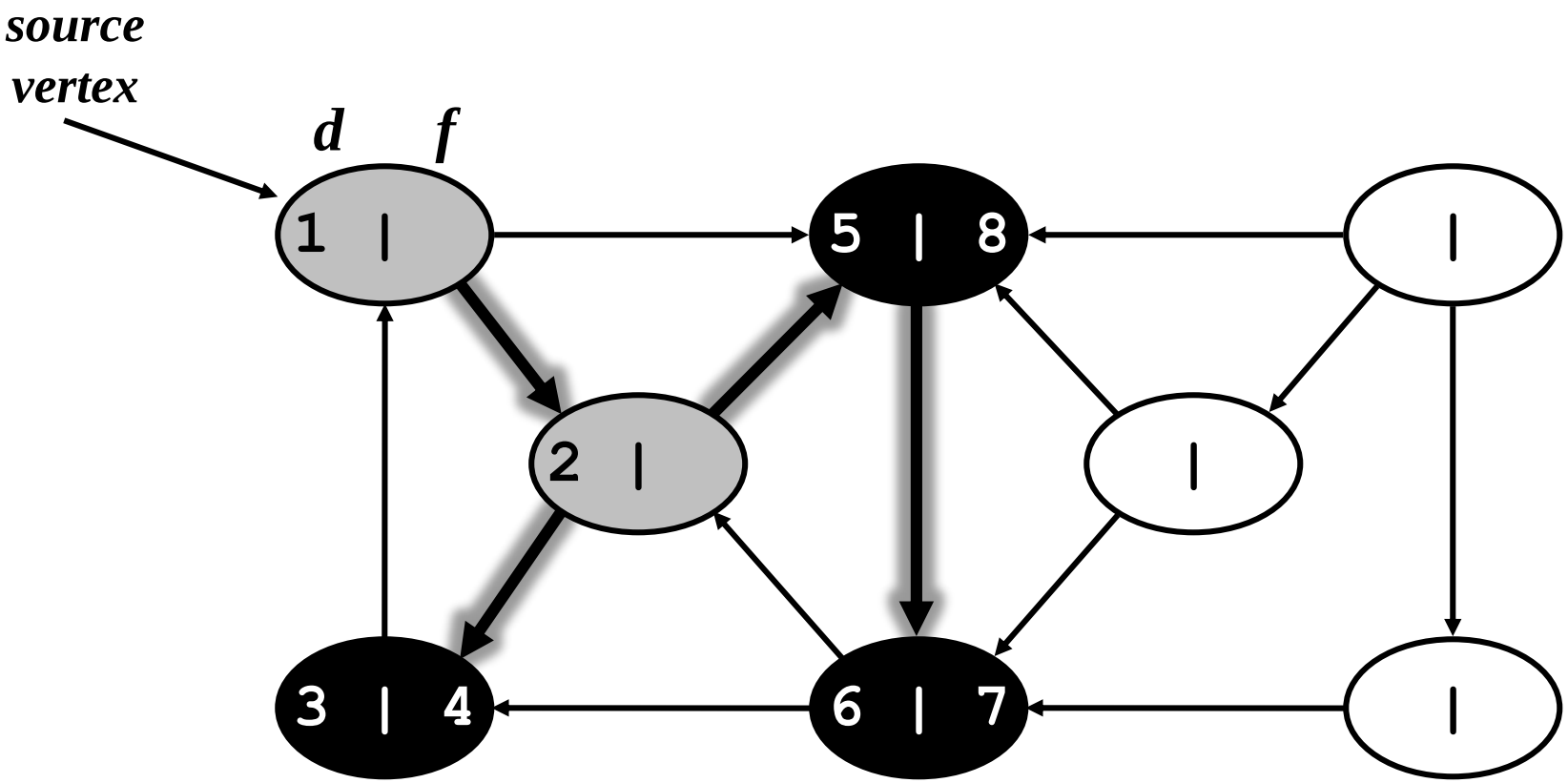
DFS Example



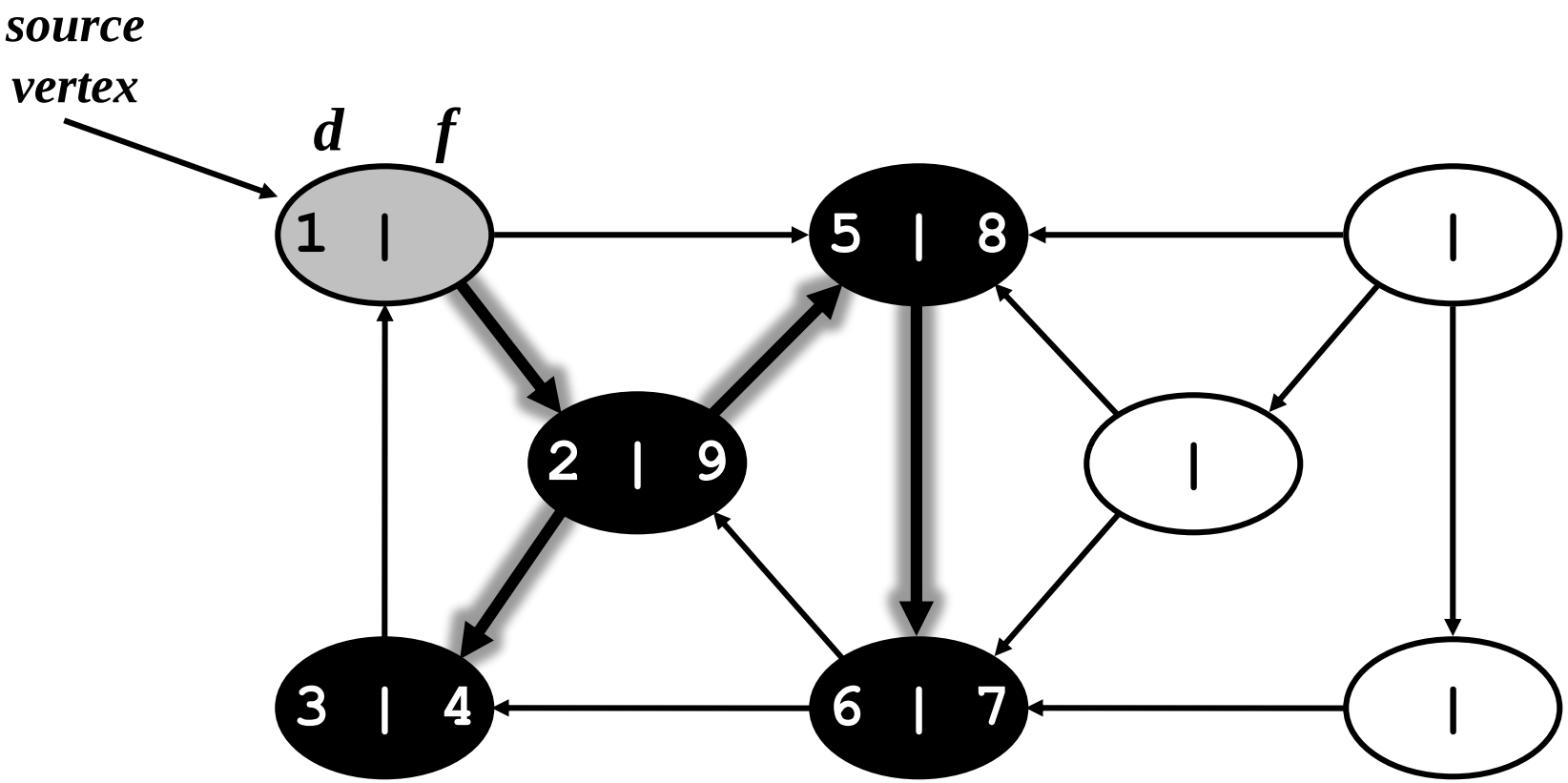
DFS Example



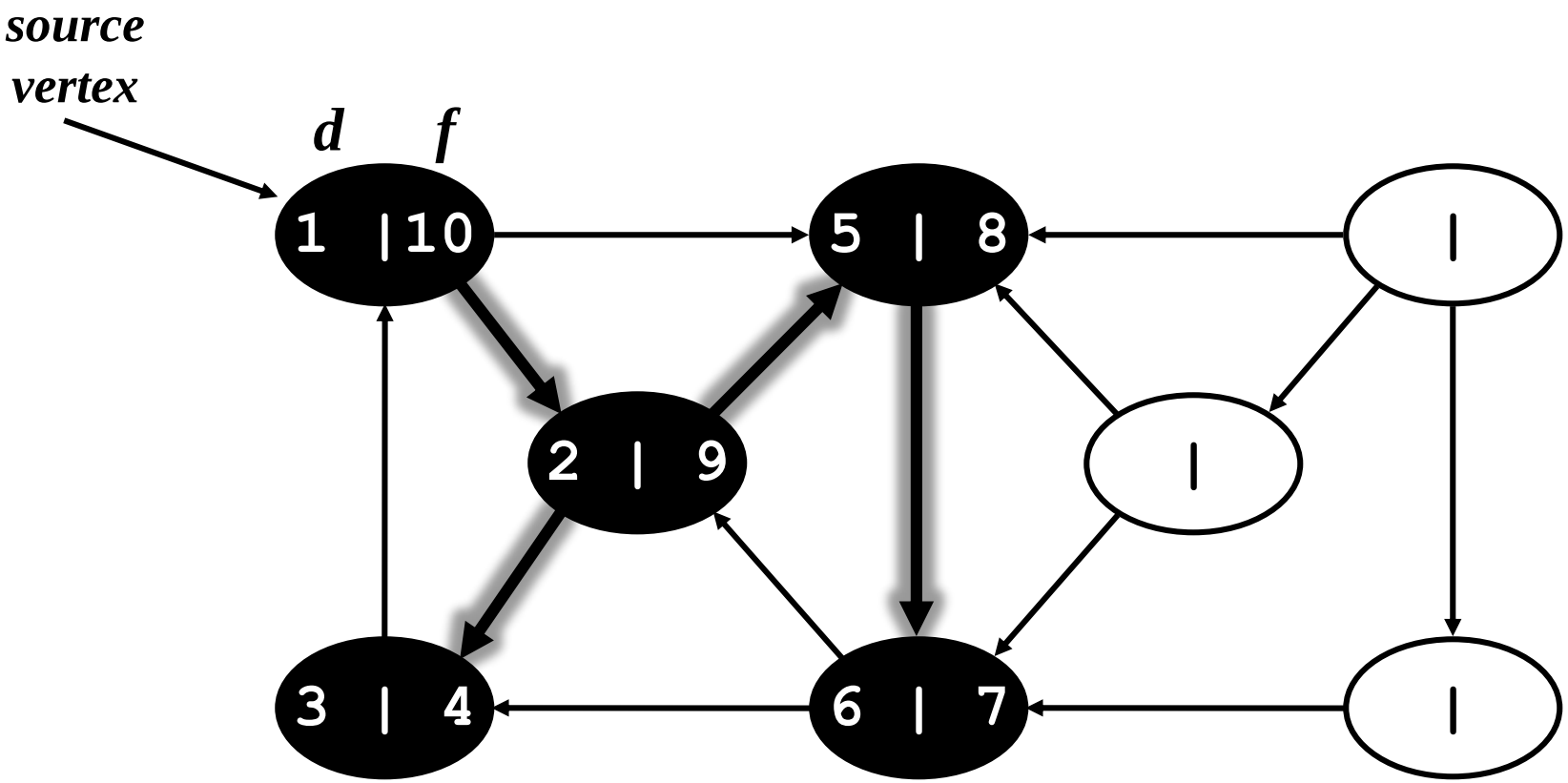
DFS Example



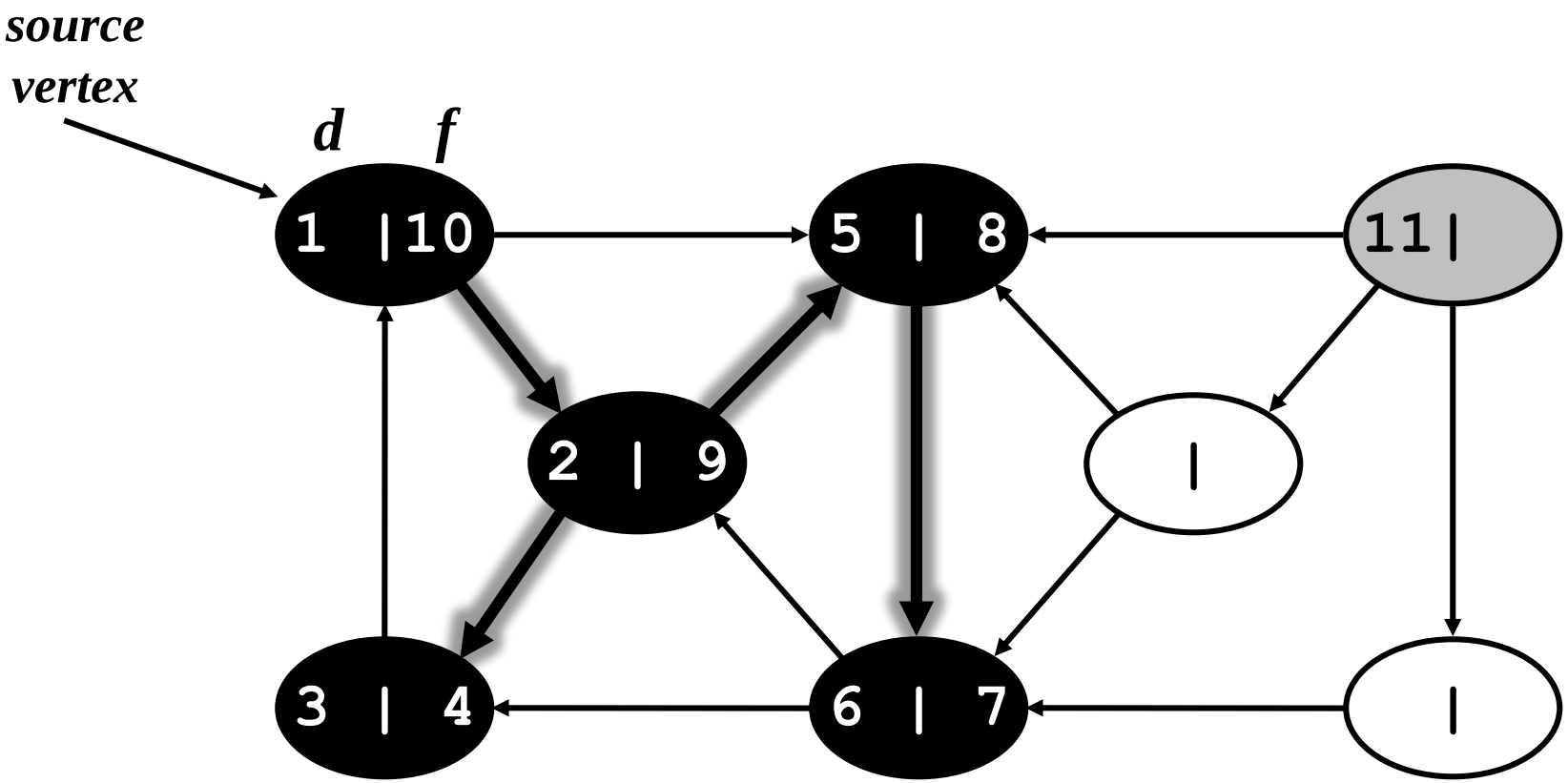
DFS Example



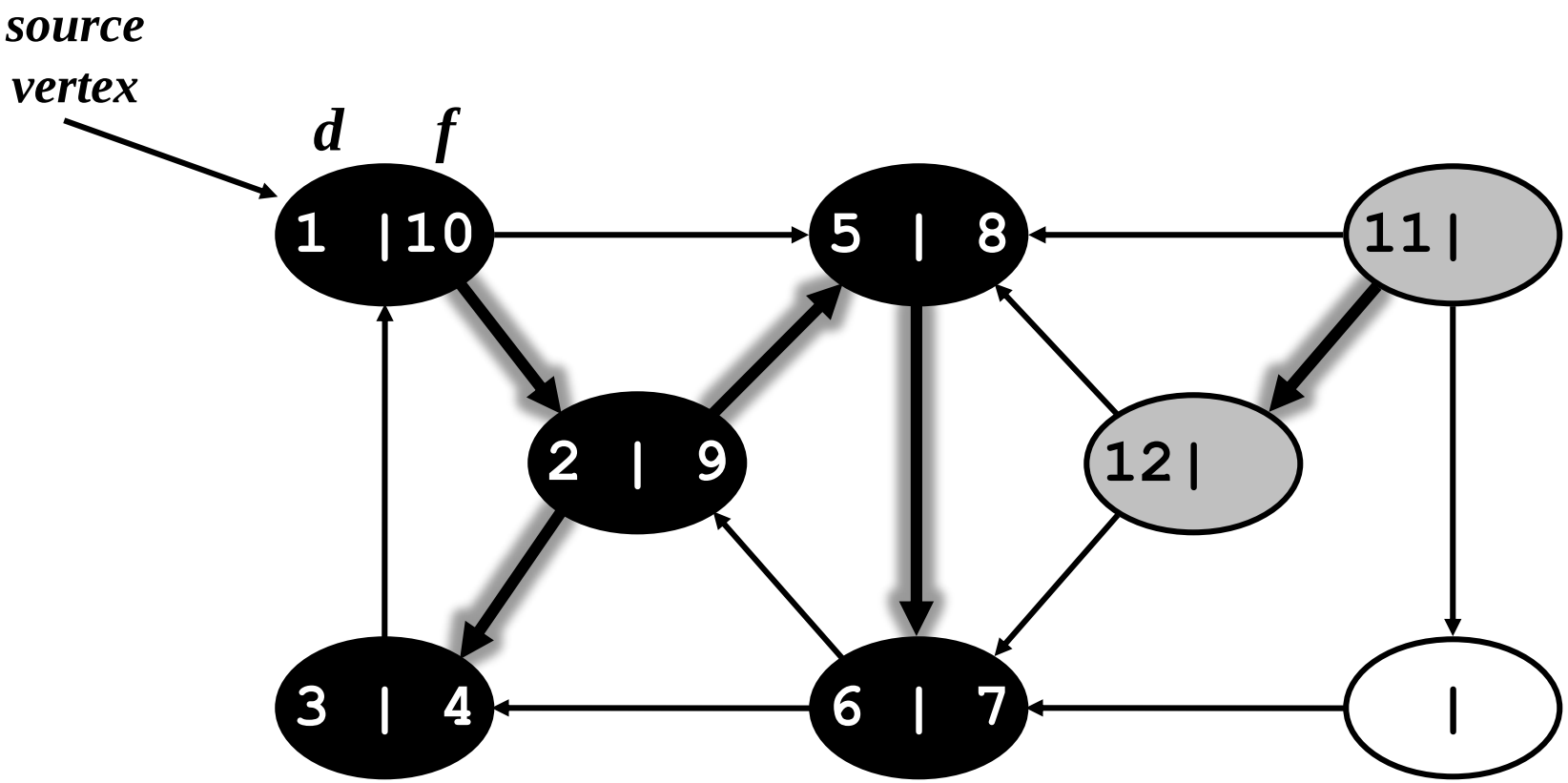
DFS Example



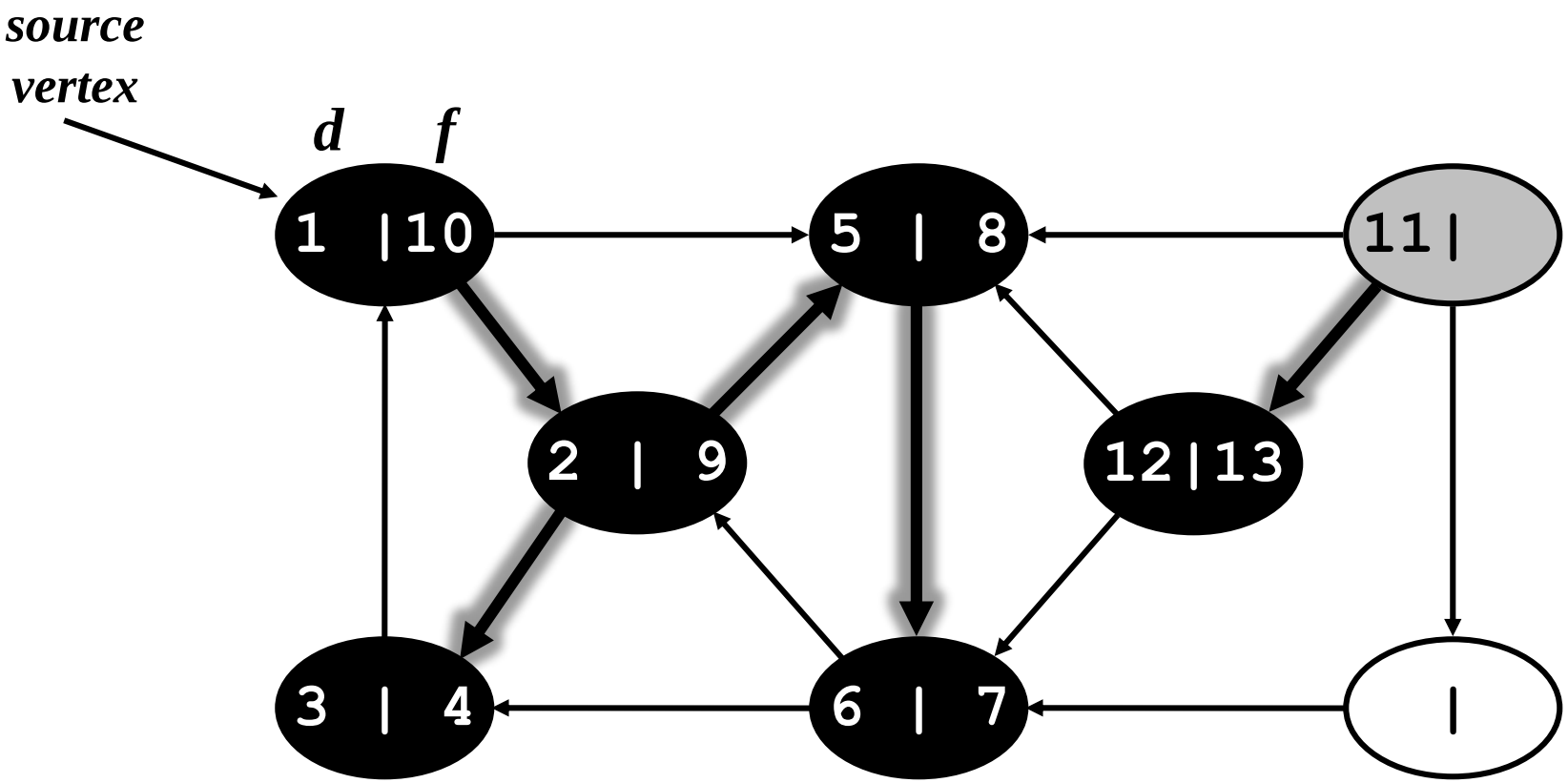
DFS Example



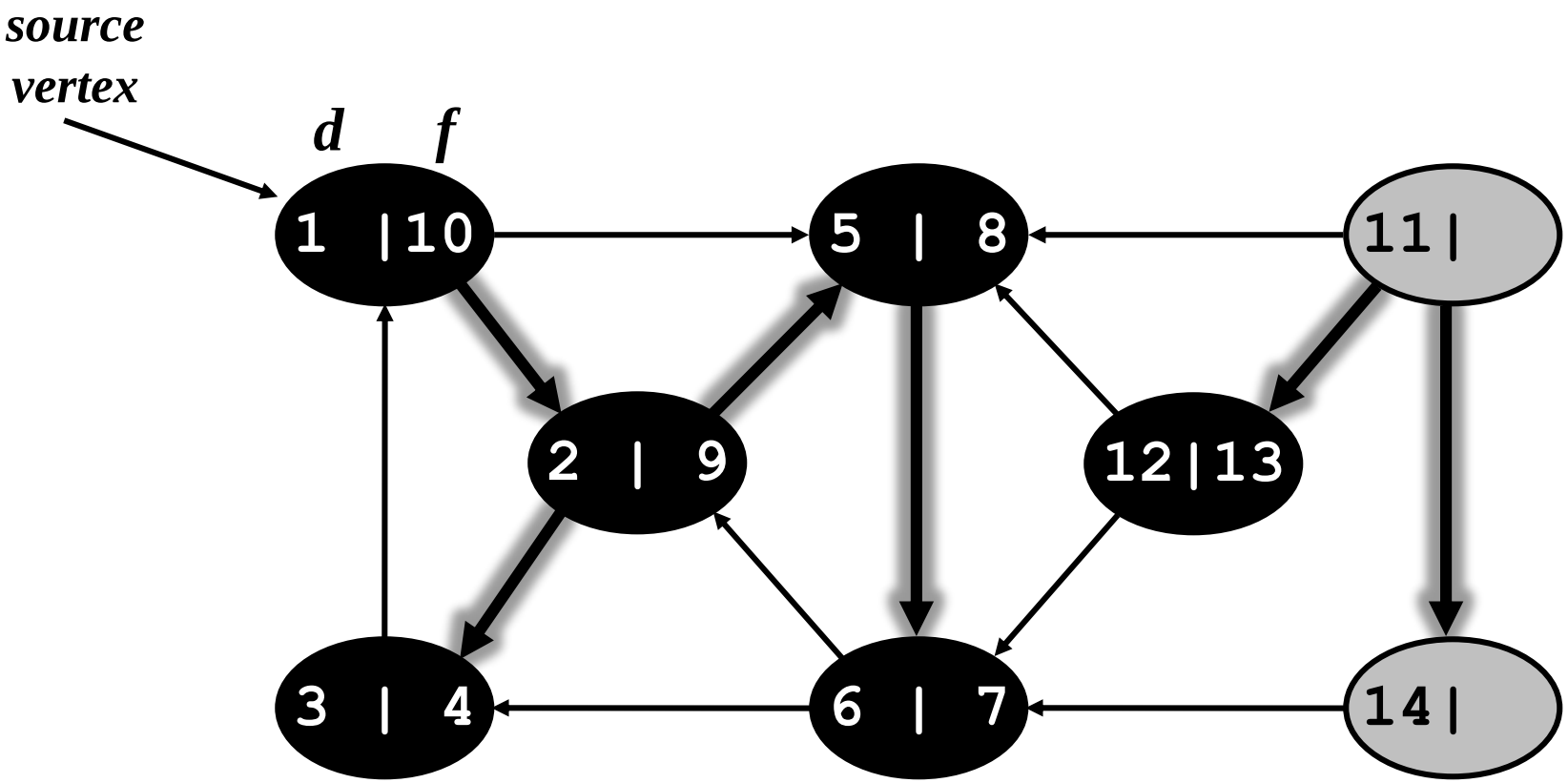
DFS Example



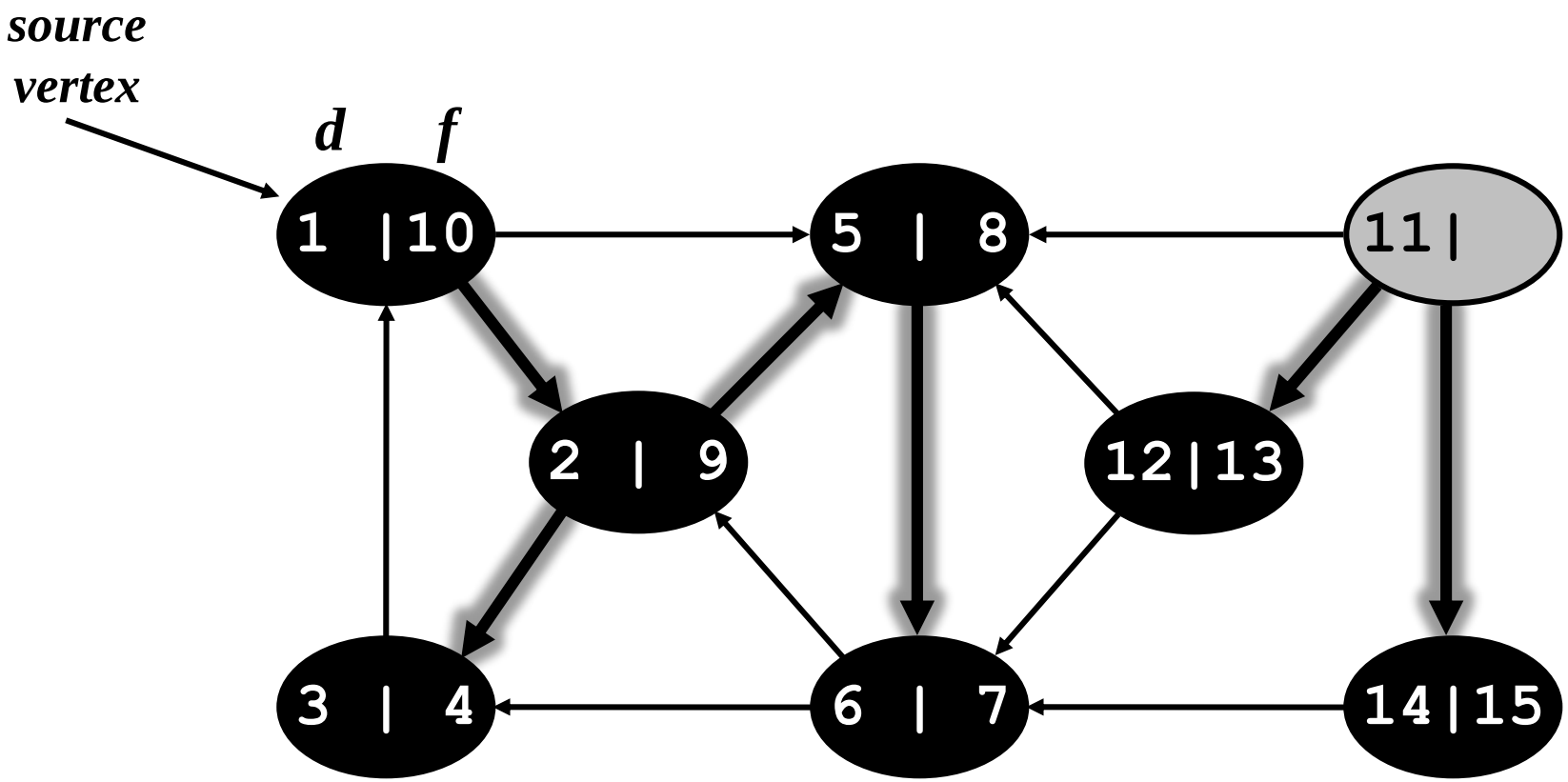
DFS Example



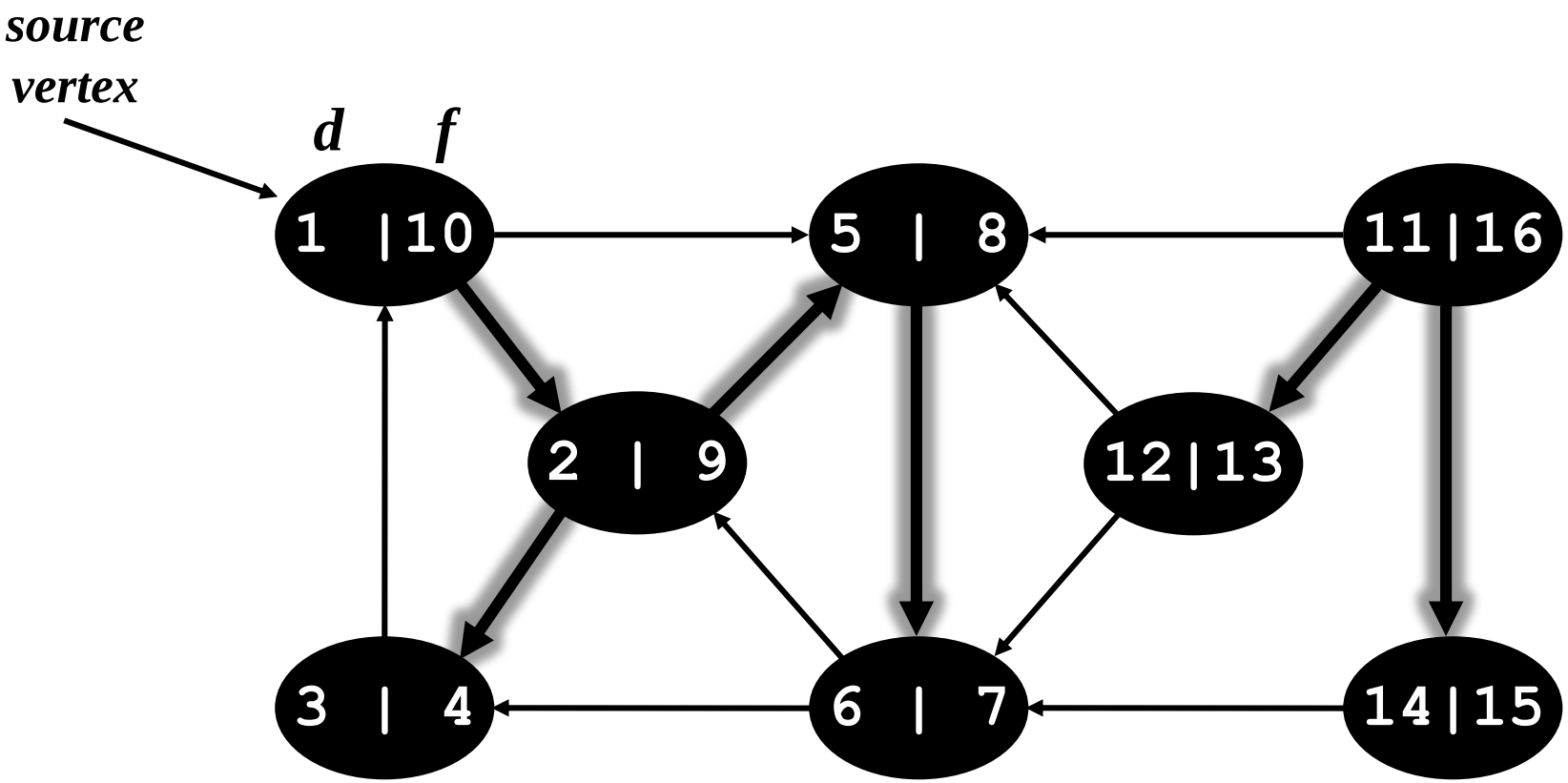
DFS Example



DFS Example

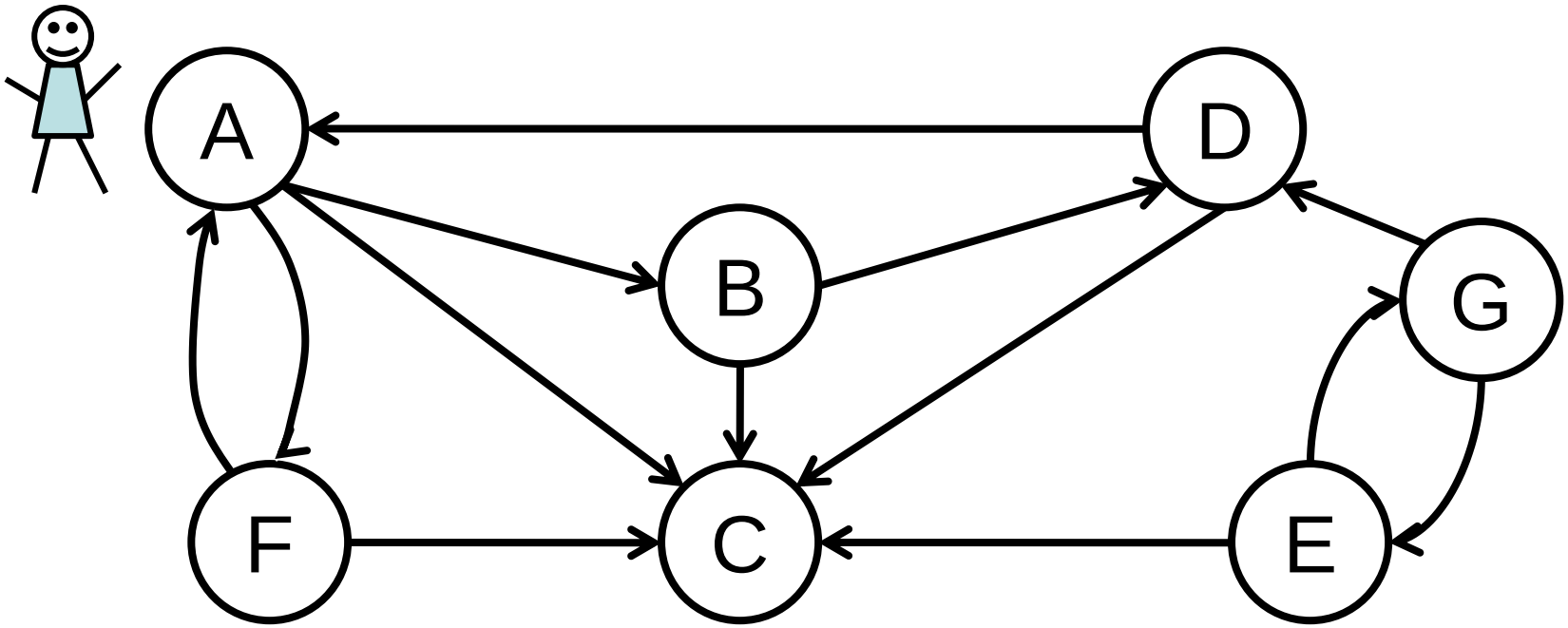


DFS Example



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

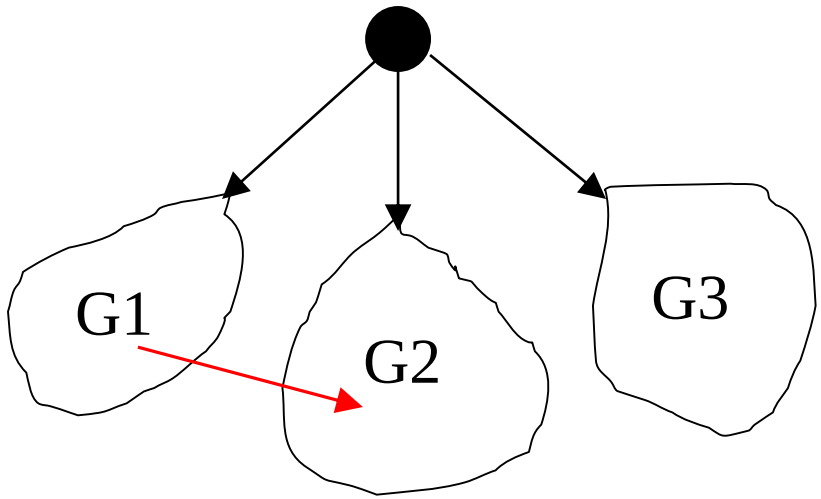
Each vertex is visited once and each edge is explored or checked once

$\therefore \Theta(V+E)$

```
DFS-VISIT(u)
 1 color[u] ←GRAY
 2 d[u] ← time ← time + 1
 3 for each v ∈ Adj[u]               ⟶ Σ_{v∈V} |Adj[v]| = Θ(E)
 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
```

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.

Deapth-first search

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

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

```
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

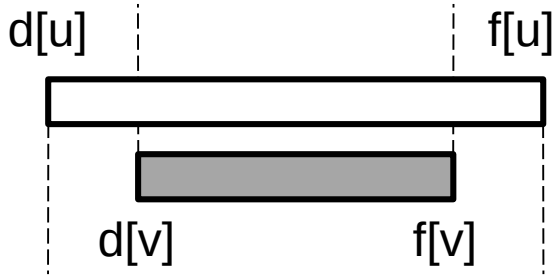
Assume that $d[u] < d[v]$ (because of symmetry)

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

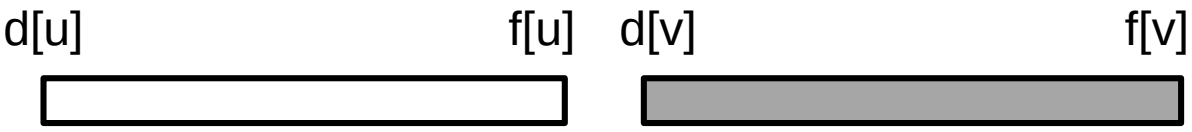
So, $f[v] < f[u]$.

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

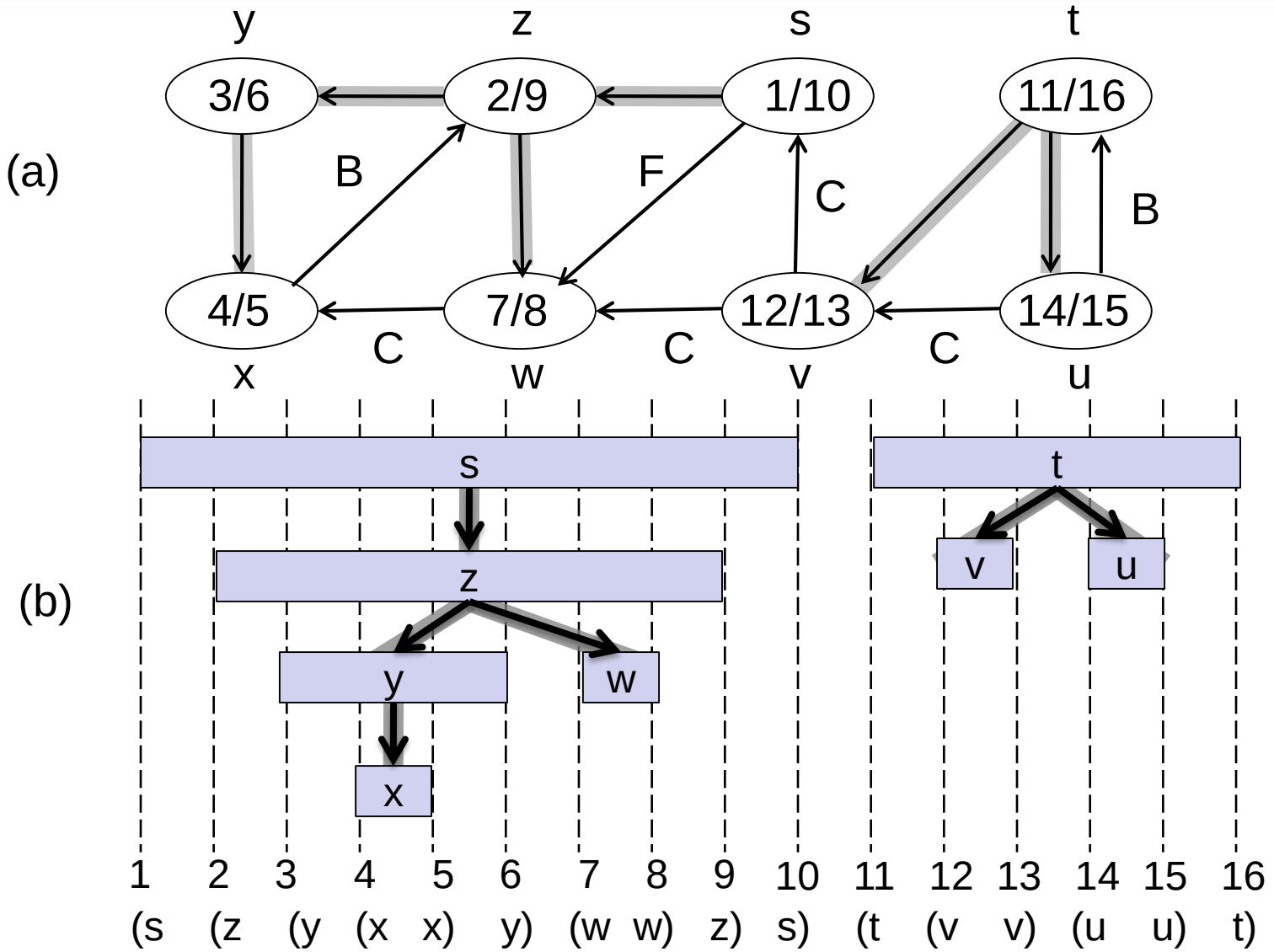
Thus, the interval $[d[v], f[v]]$ is entirely contained within the interval $[d[u], f[u]]$.

② $d[v] > f[u]$ (disjoint intervals)

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



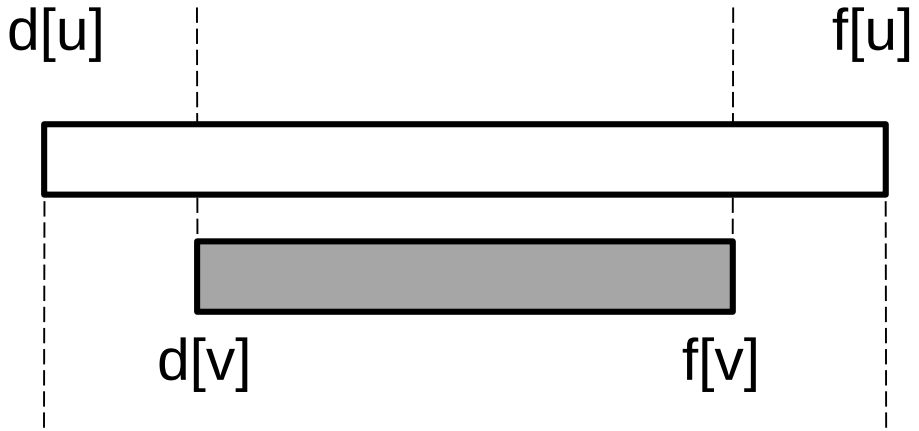
Parenthesis theorem



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 :

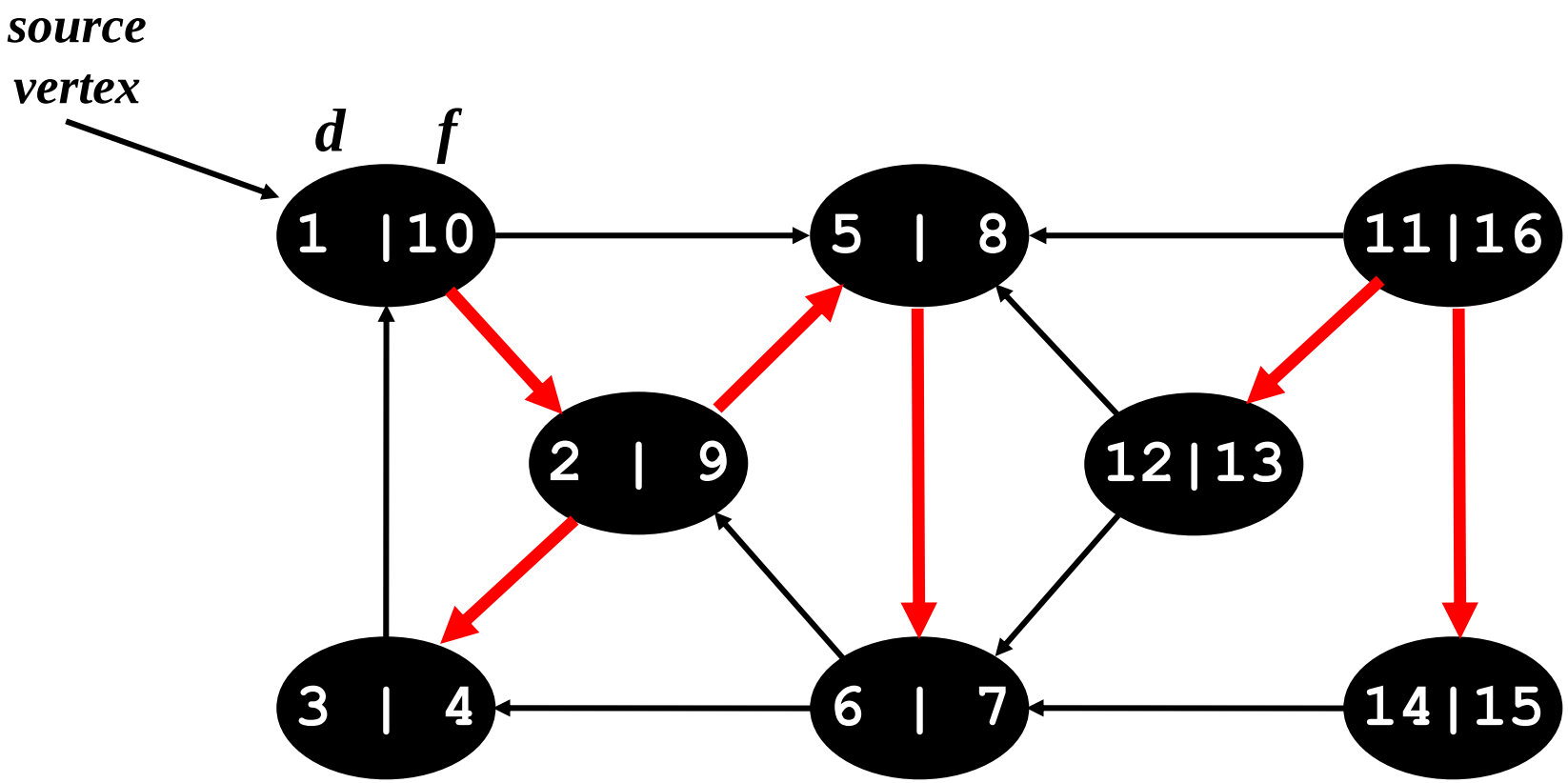
$$d[u] < d[v] < f[v] < f[u]$$



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

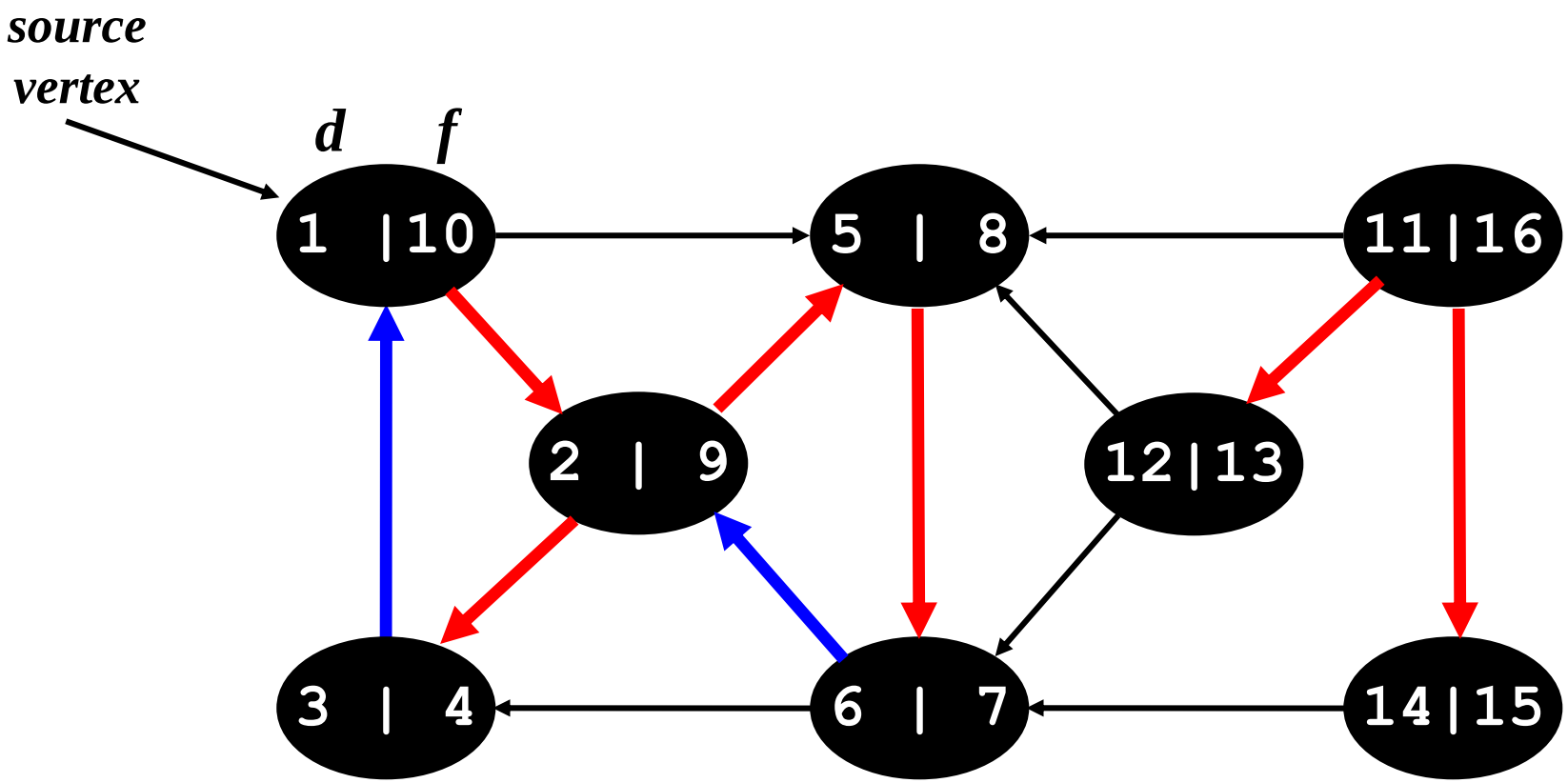


Tree edges

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

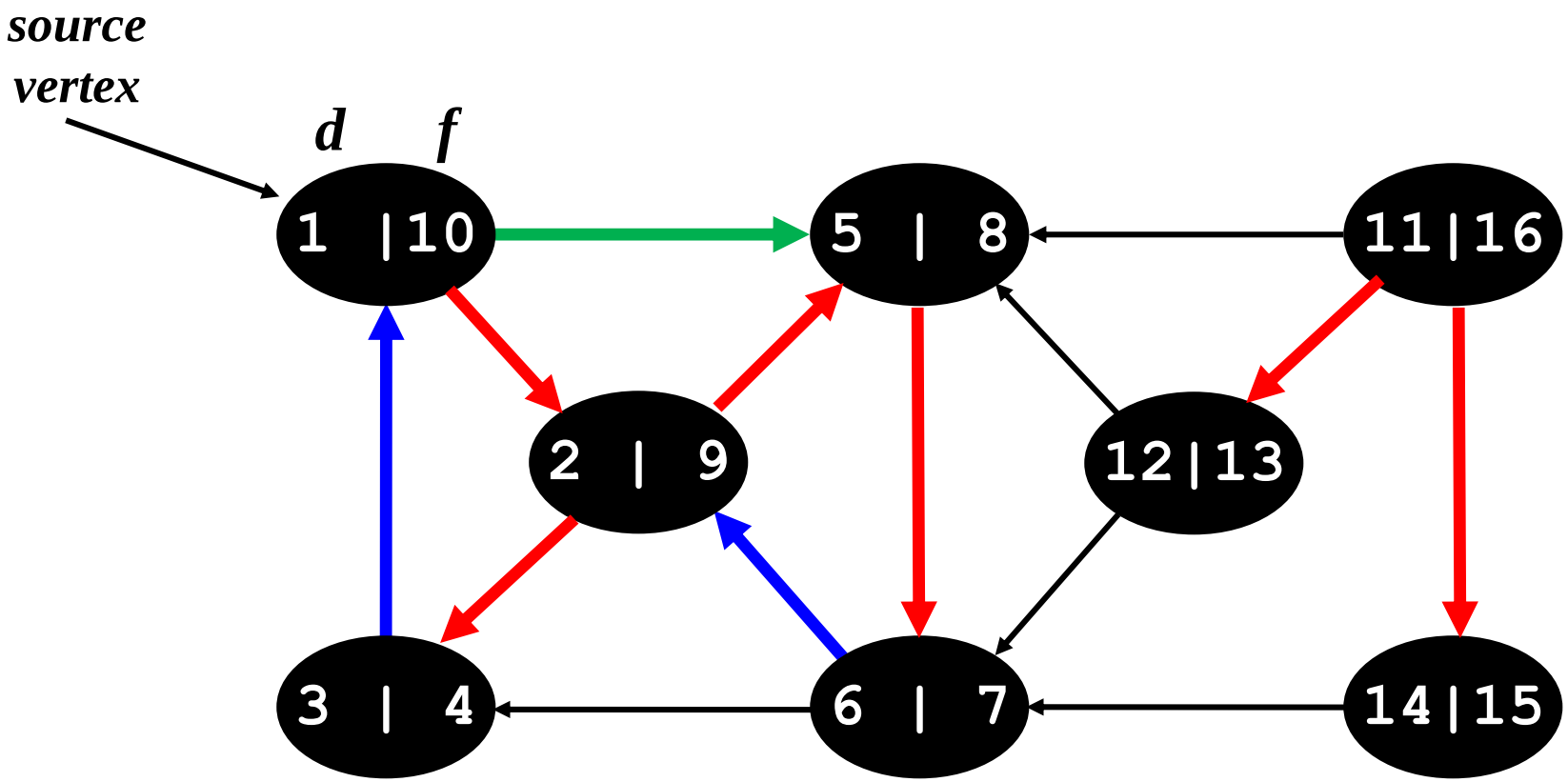


Tree edges *Back edges*

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
    - From gray node to black node

Forward edge

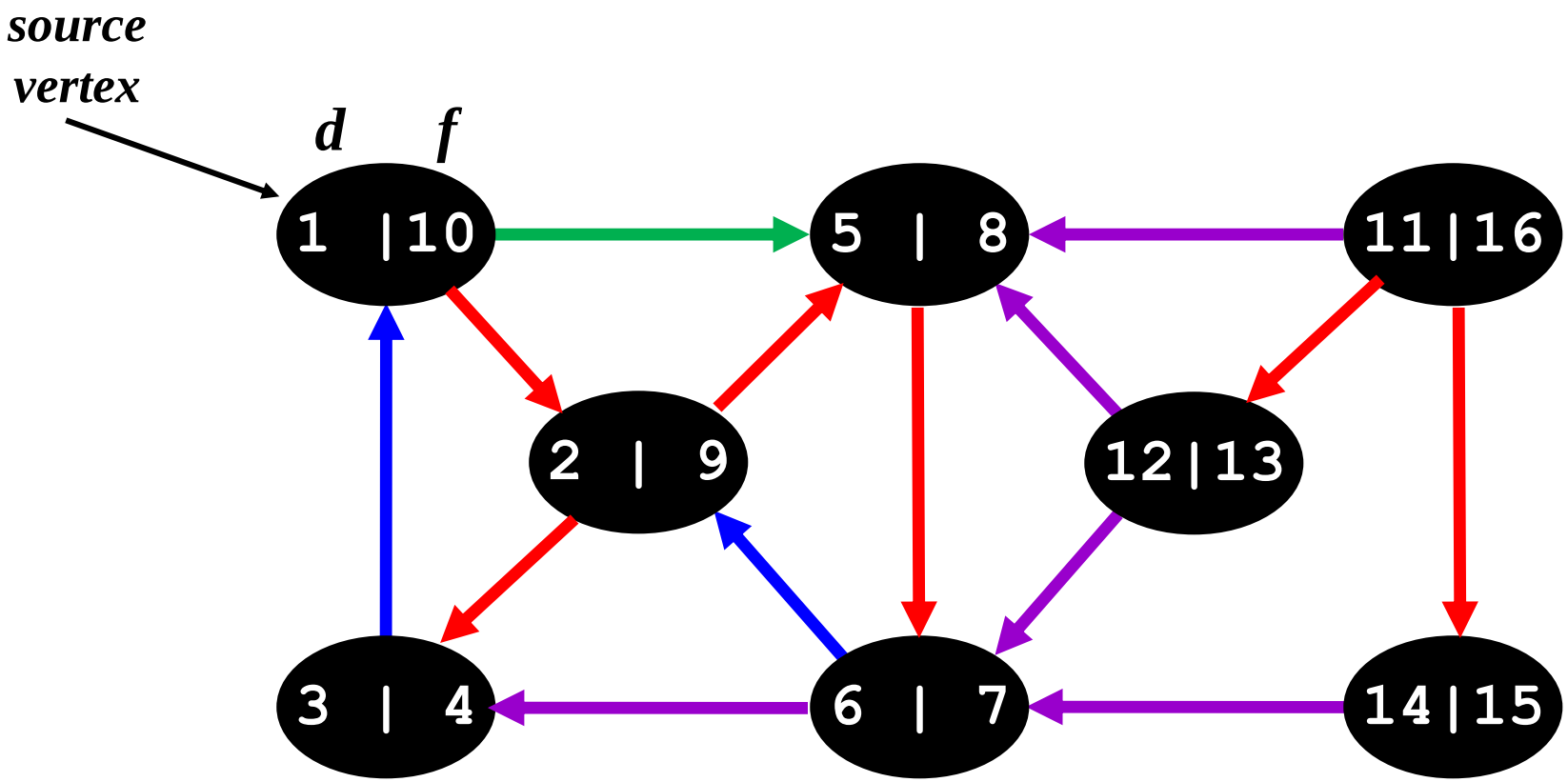


Tree edges *Back edges* *Forward edges*

DFS: Kinds of edges

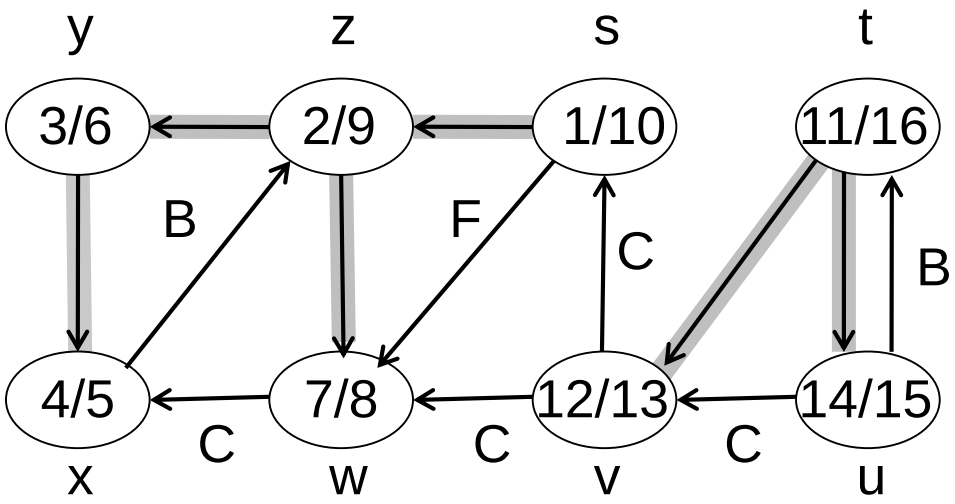
- DFS introduces an important distinction among edges in the original graph:
  - *Tree edge*: encounter new (white) vertex
  - *Back edge*: from descendent to ancestor
  - *Forward edge*: from ancestor to descendent
  - *Cross edge*: between a tree or subtrees (checked)
    - From a gray node to a black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

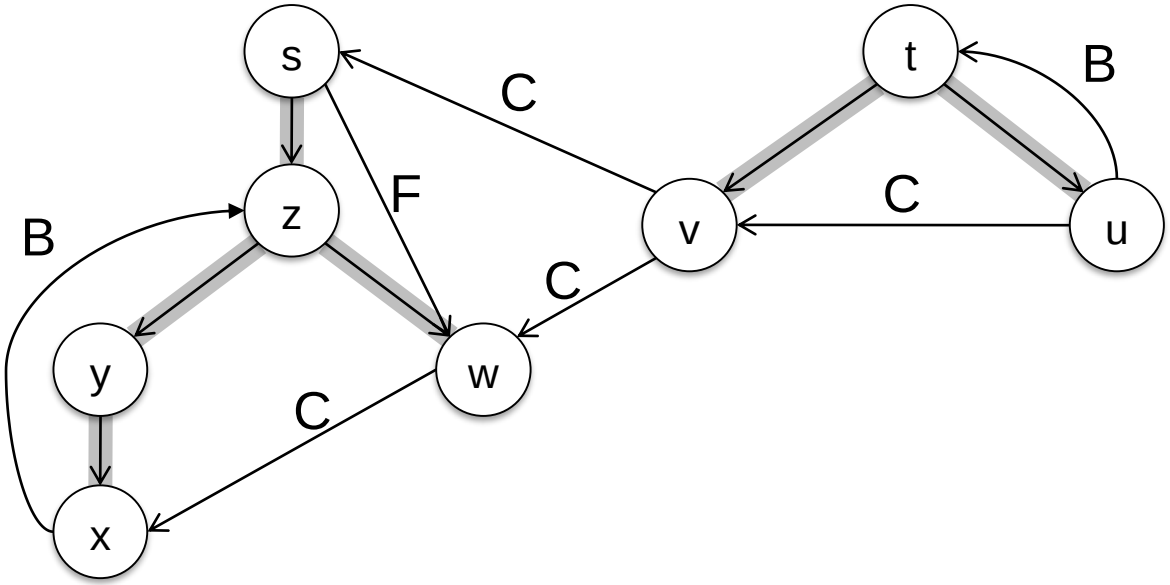
Cross edge



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

DFS Example





B : Back Edge
F : Forward Edge
C : Cross Edge
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

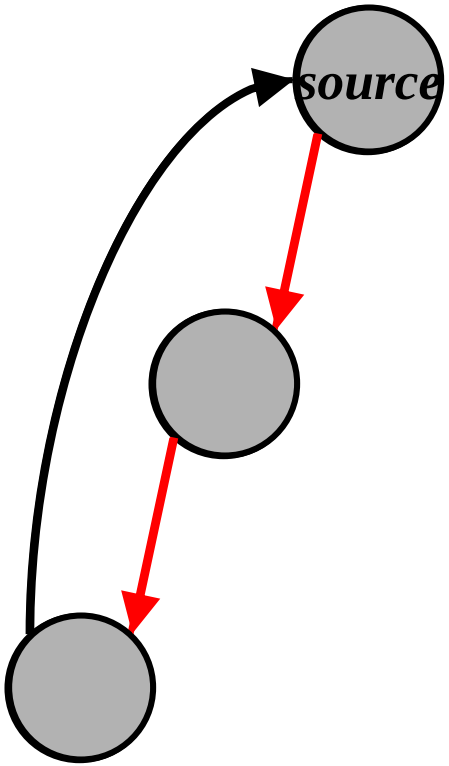
transform it into symmetric digraph, then process edge twice.

Else

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

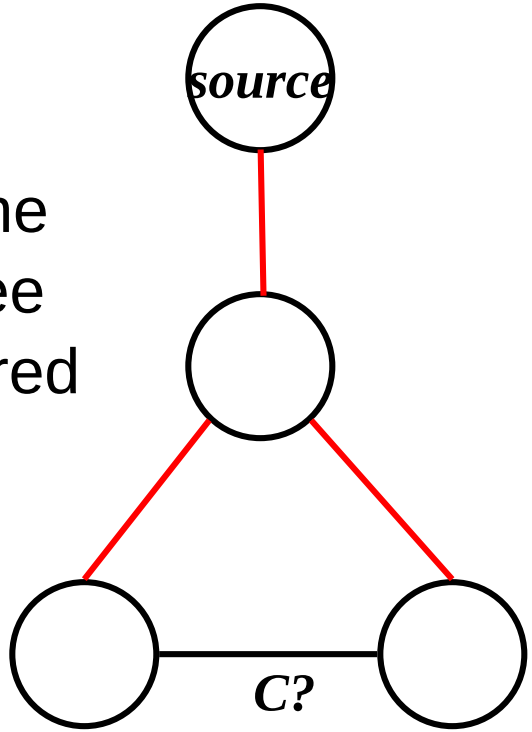
DFS: Kinds Of Edges

- Theorem 22.10: If *G* is undirected, a DFS produces only tree and back edges
- Proof by contradiction:
  - Assume there's a forward edge
    - But F? edge must actually be a back edge (*why?*)



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

  BFS : Queue

- DFS : Two processing opportunity

  - Once when it is discovered : Preorder
  - Once when it is marked finished : Postorder
  - Postorder : We have information about adjacent vertices.

  ex) Counting the number of leaves in binary tree

  BFS : One processing opportunity

Applications of DFS

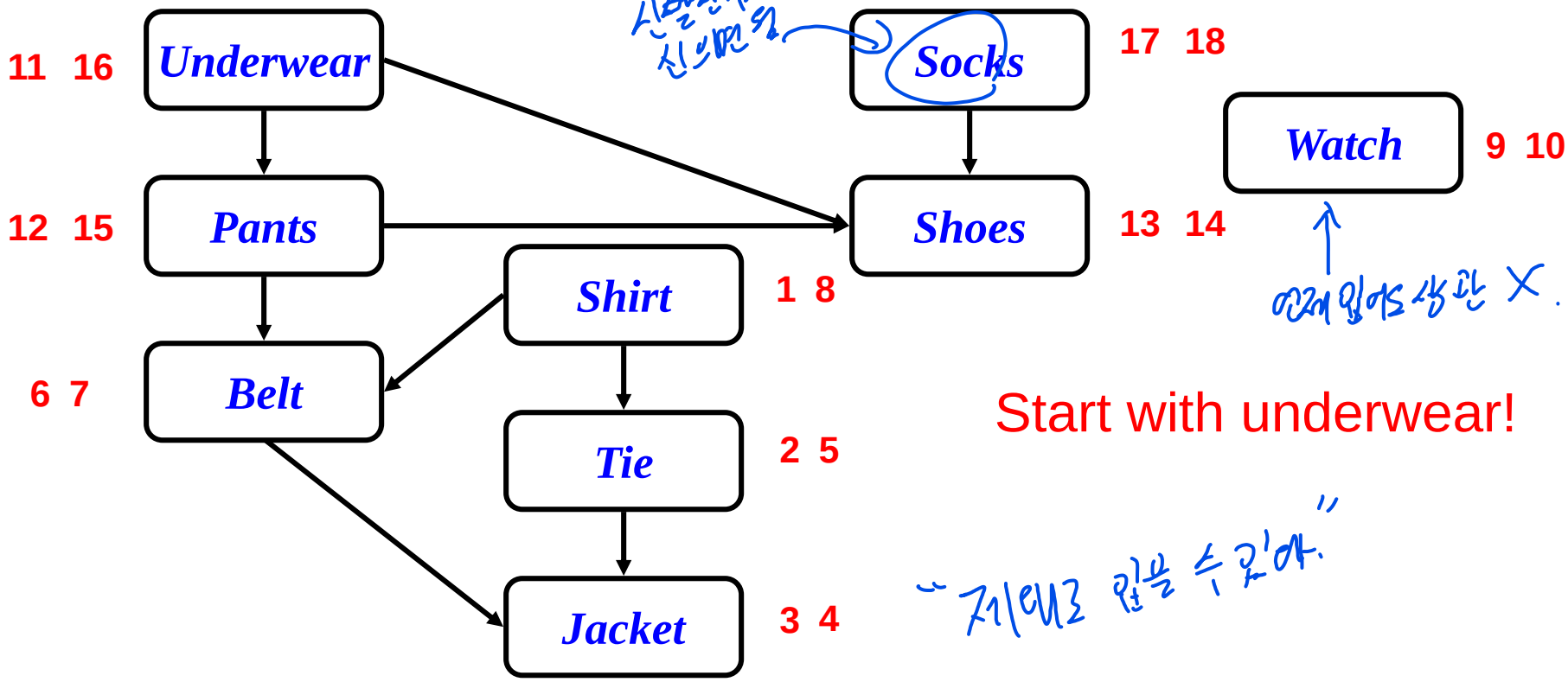
1. Topological Sort
2. Finding Connected Component (Undirected)
3. Finding Strongly Connected Component (Directed)
4. Critical Path Analysis
5. Biconnected Component (Articulation point) Problem
6. Etc…

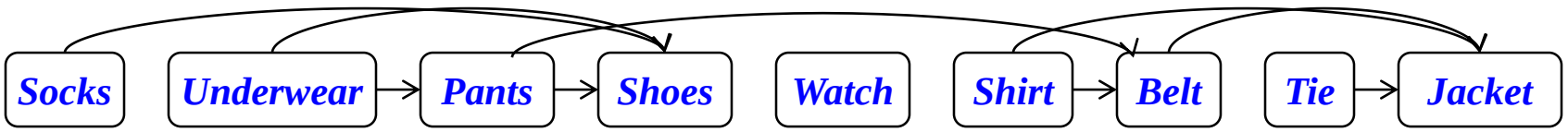
1) Topological Sort

- *Topological sort* of a DAG: → Direct Acyclic Graph 시키는 짓.
  - 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

dfs 를 하고 각 요소를 끝난 시간을 기준으로 sort 하면 끝

Getting Dressed Example





Topological Sort Algorithm

Topological-Sort()

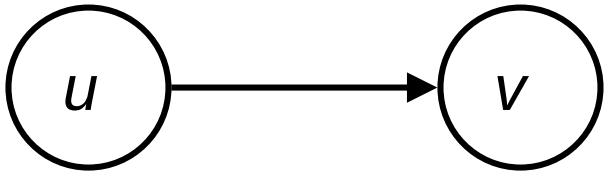
{

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.)

}

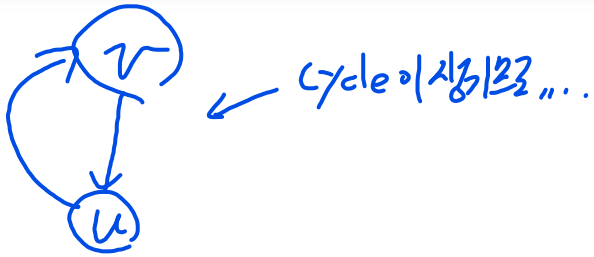
- Time: $\Theta(V+E)$
- Correctness: Want to prove that

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



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?*)

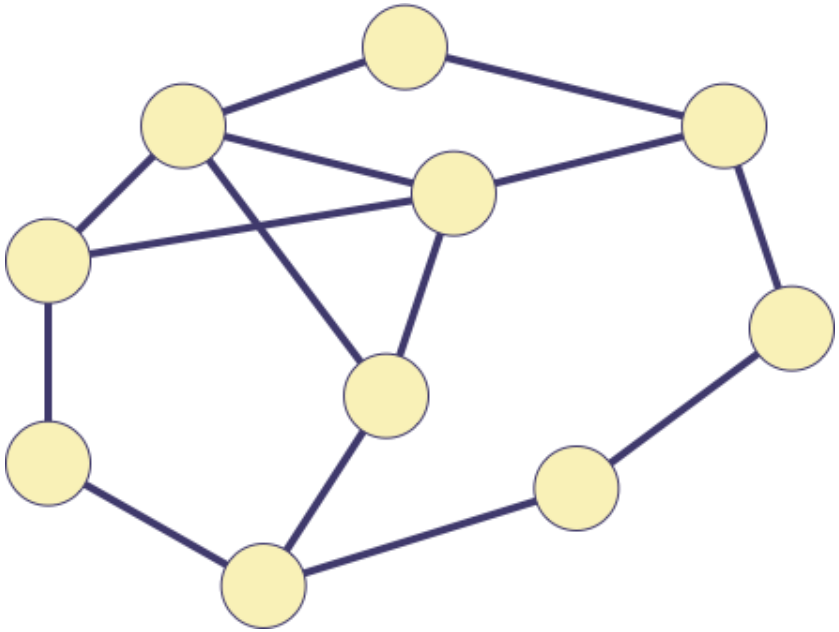
      We are dealing with DAG (no cycle) here.
    - v = white $\Rightarrow$ v becomes descendent of u $\Rightarrow$ $f[v] < f[u]$

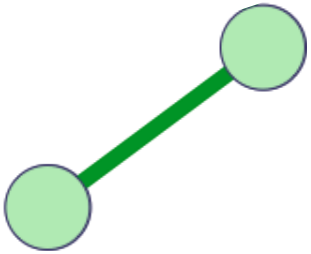
      (since must finish v before backtracking and finishing u.)
    - v = black $\Rightarrow$ v already finished $\Rightarrow$ $f[v] < f[u]$

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

- 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.

dfs 전에 아이디를 생성해 dfs 에 시작 노드와 더불어 아이디도 같이 넘겨준다. dfs 가 돌면서 방문하는 노드에 같은 아이디 값을 저장해 준다.

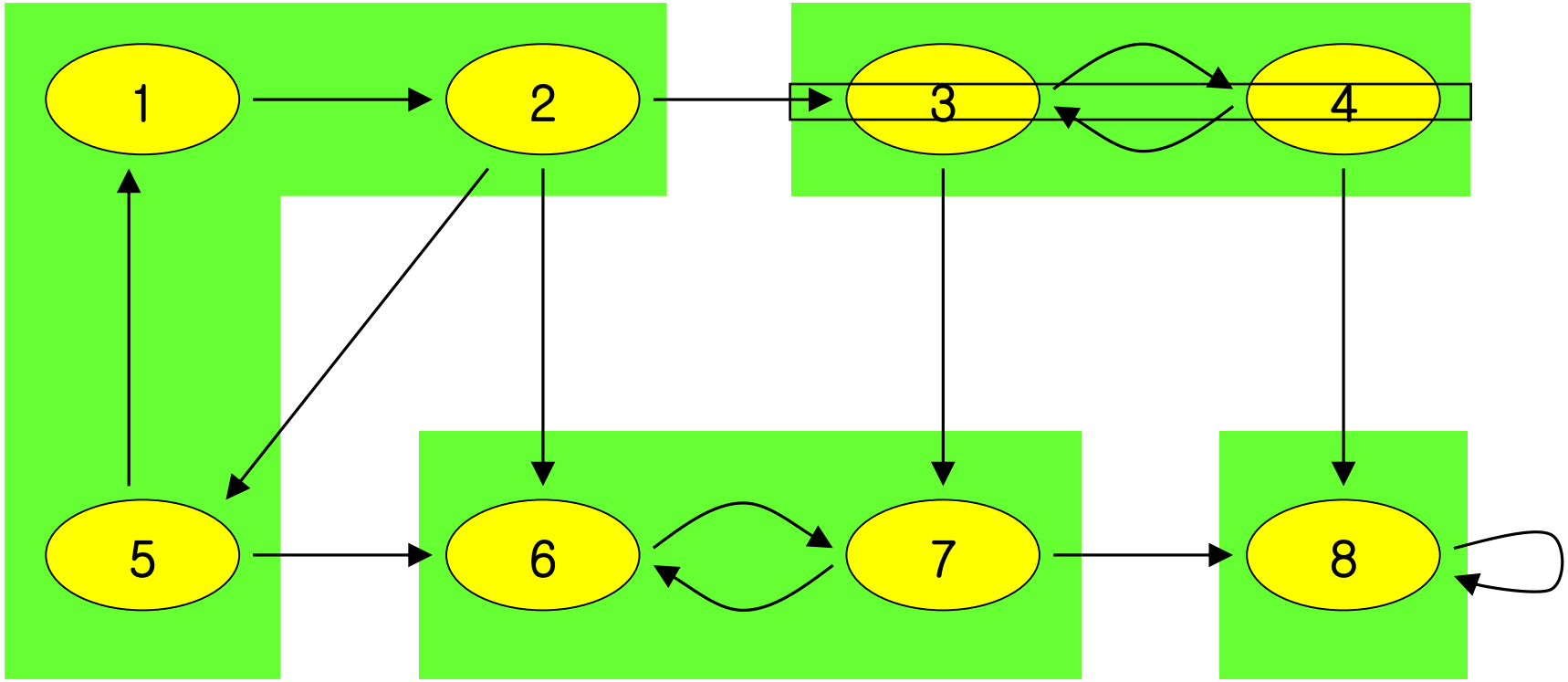
3) Strongly Connected Component

- 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.

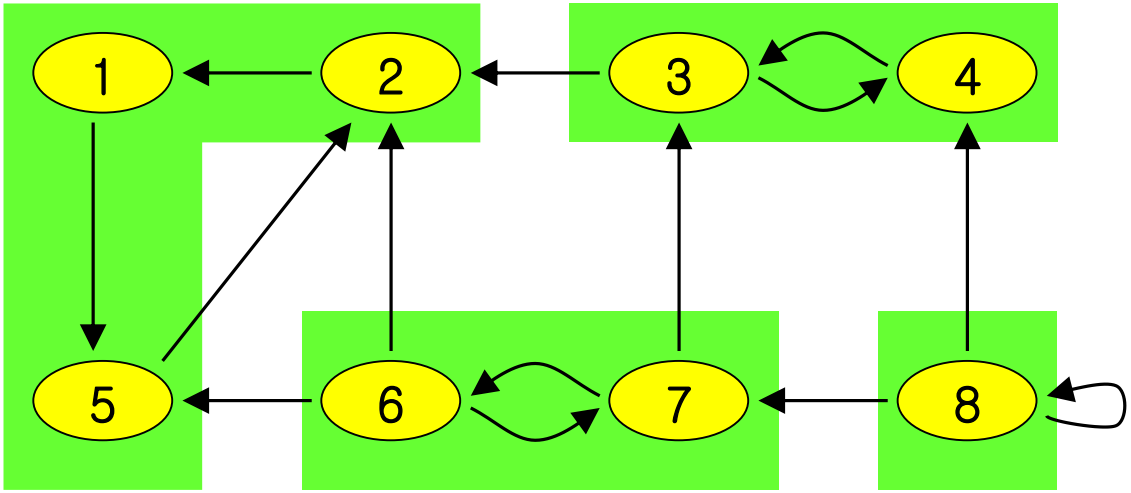
Example



Green areas are the Strongly Connected Components.

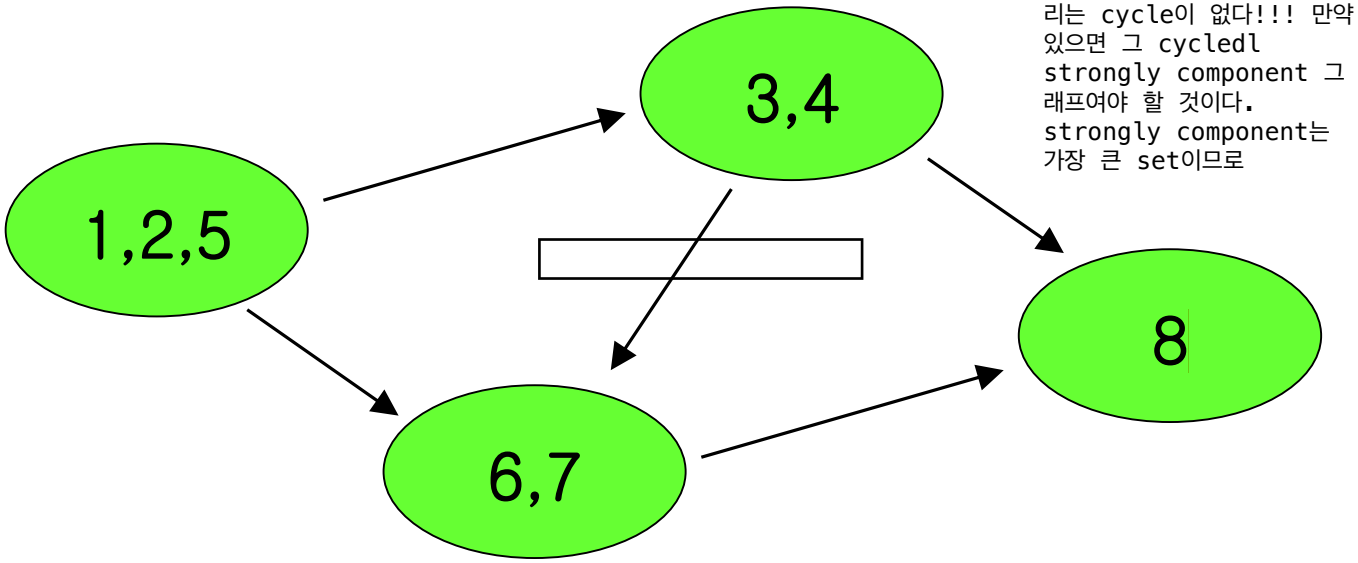
Property

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



Property

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



만약 strongly component 끼리 묶어서 그래프를 만들면 그 그래프들 끼리는 cycle이 없다!!! 만약 있으면 그 cycledl strongly component 그래프여야 할 것이다. strongly component는 가장 큰 set이므로

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

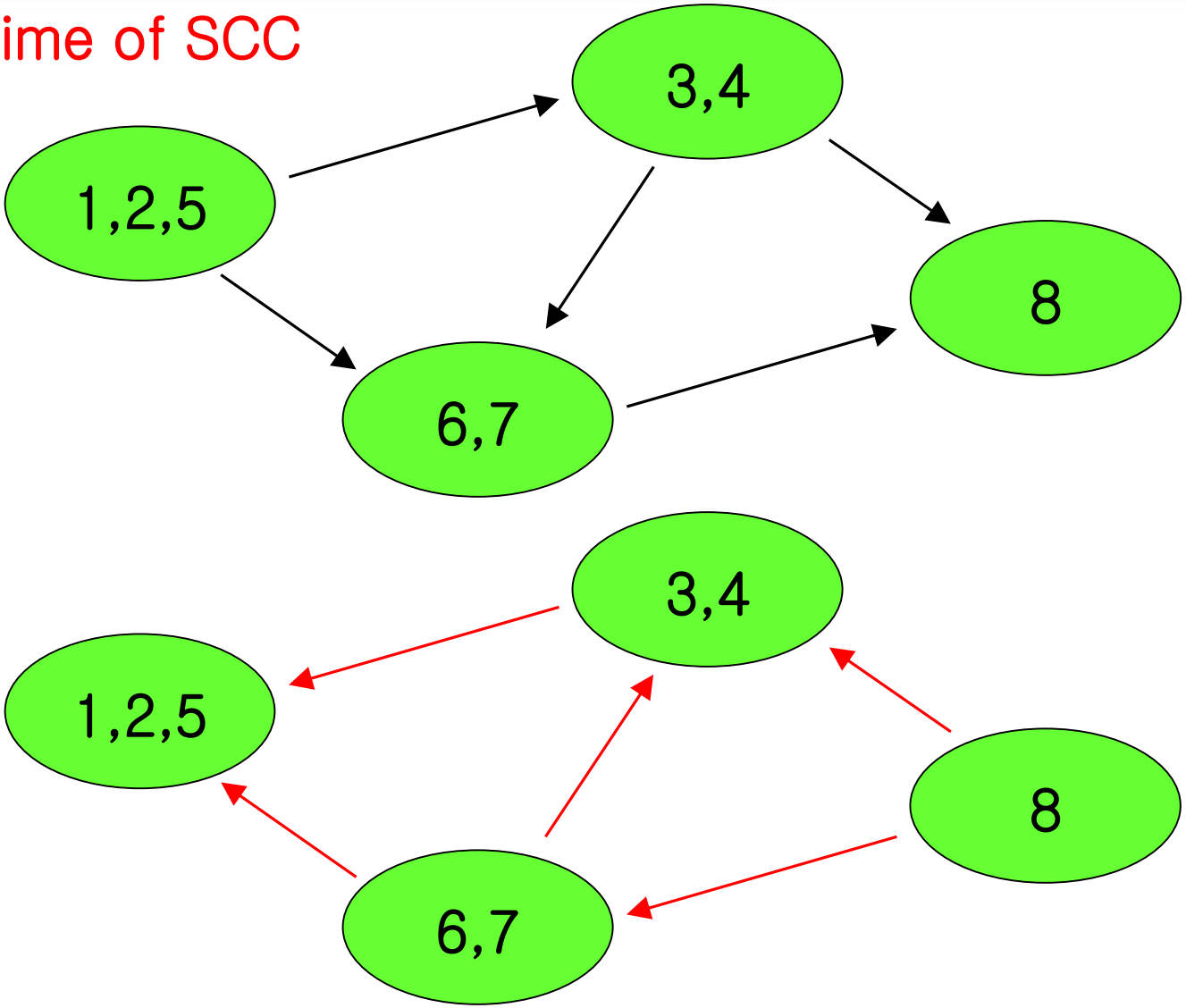
STRONGY-CONNECTED-COMPONENTS(G)

1. call DFS(G) to compute finishing times $f[u]$ for each vertex u
2. compute G^T
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)
4. Output the vertices of each tree in the depth-first forest formed in line 3 as a separate strongly connected component

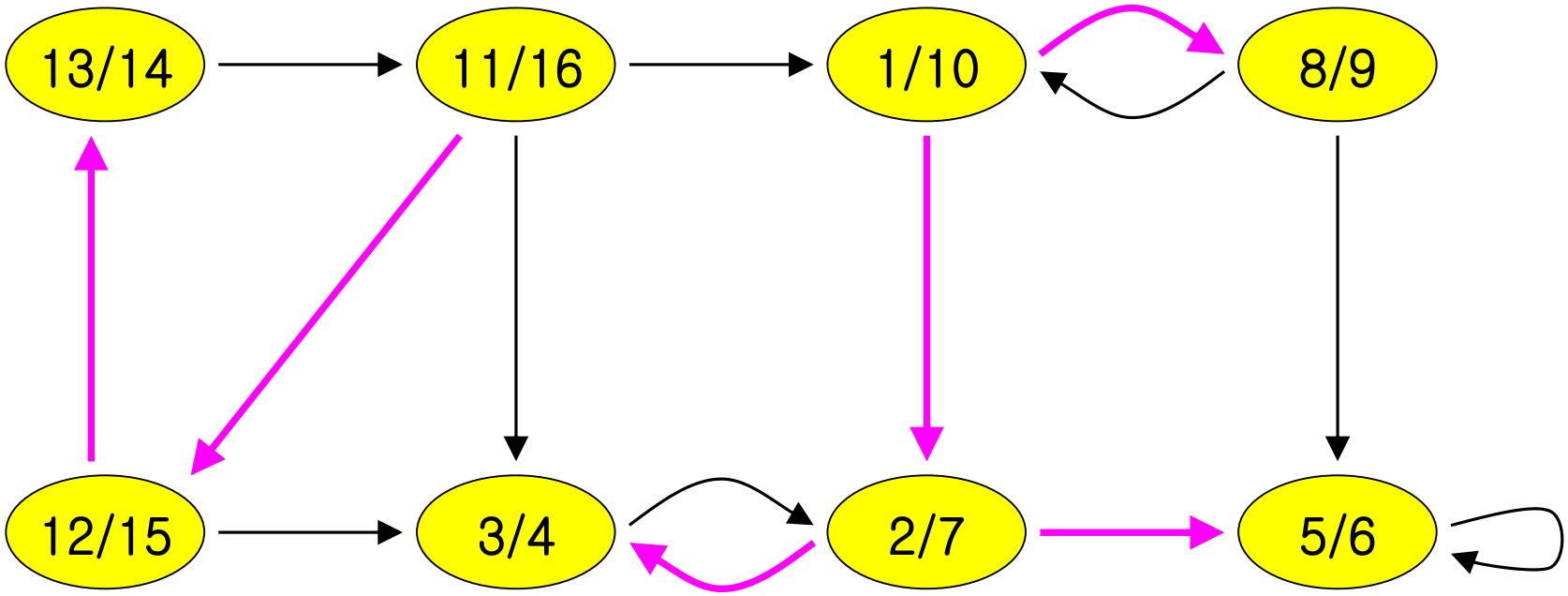
$\therefore$ **The running time of this algorithm is $\Theta(V+E)$**

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

Finish time of SCC

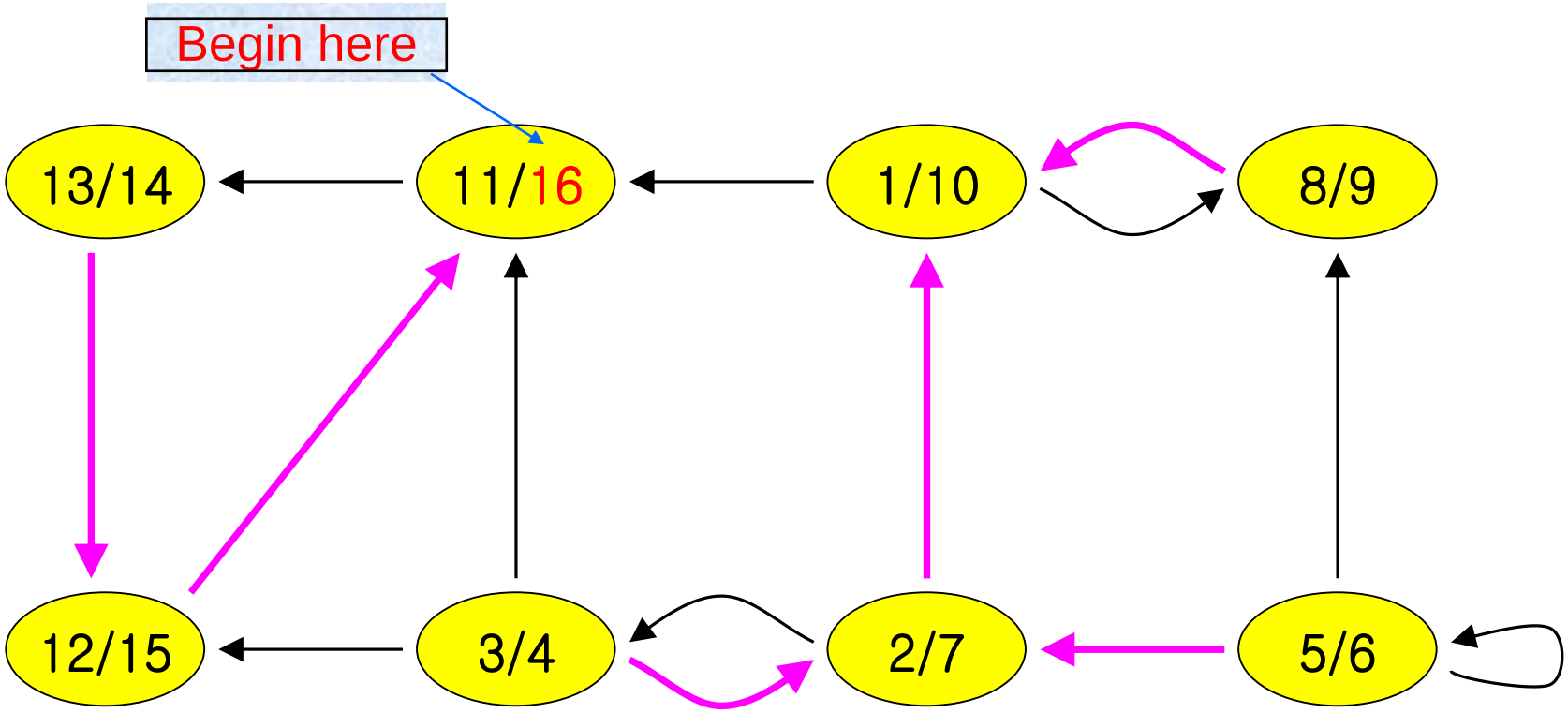


Algorithm : step1



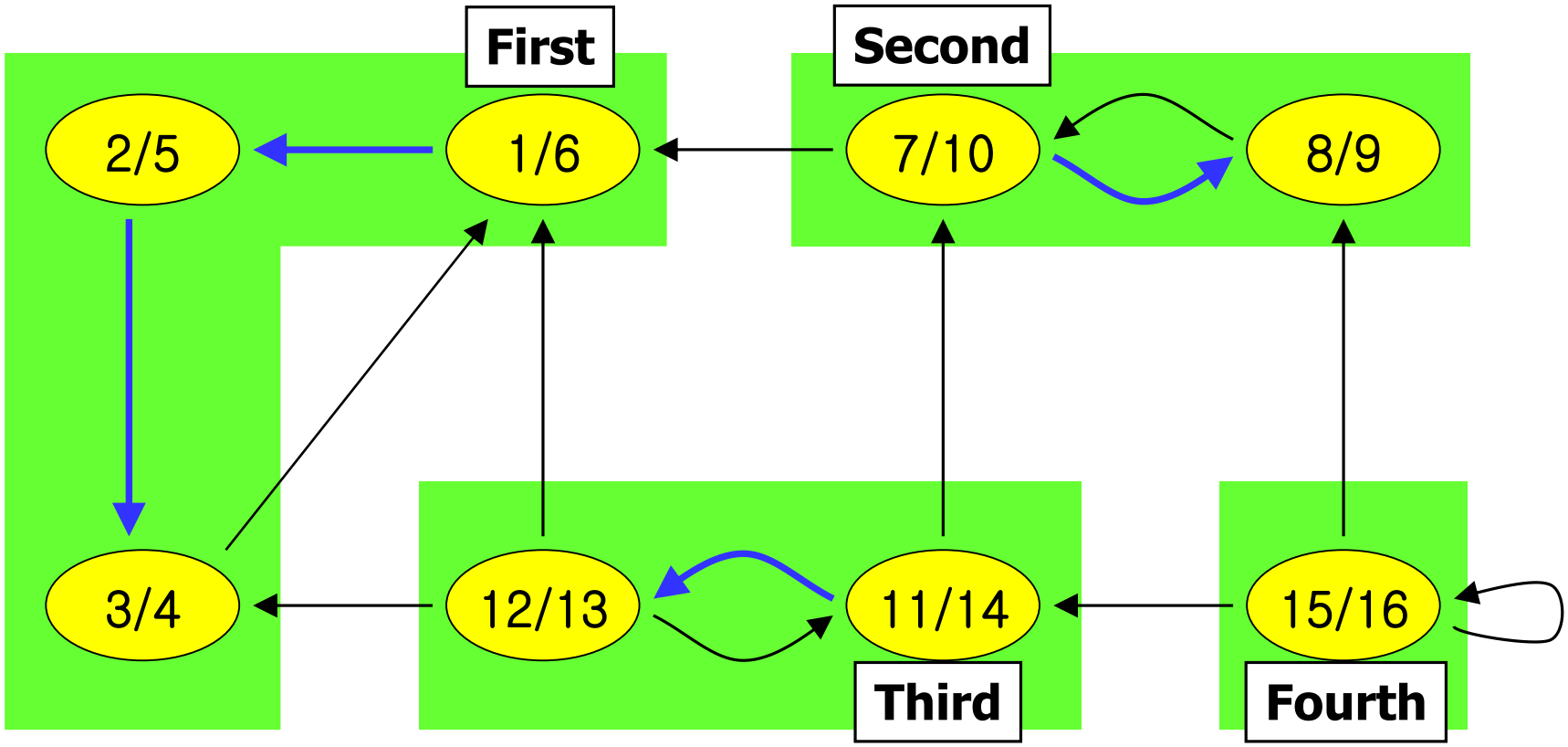
Call DFS(*G*)

Algorithm : step2



Compute transpose of *G* !!

Algorithm : step3



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

Lemma

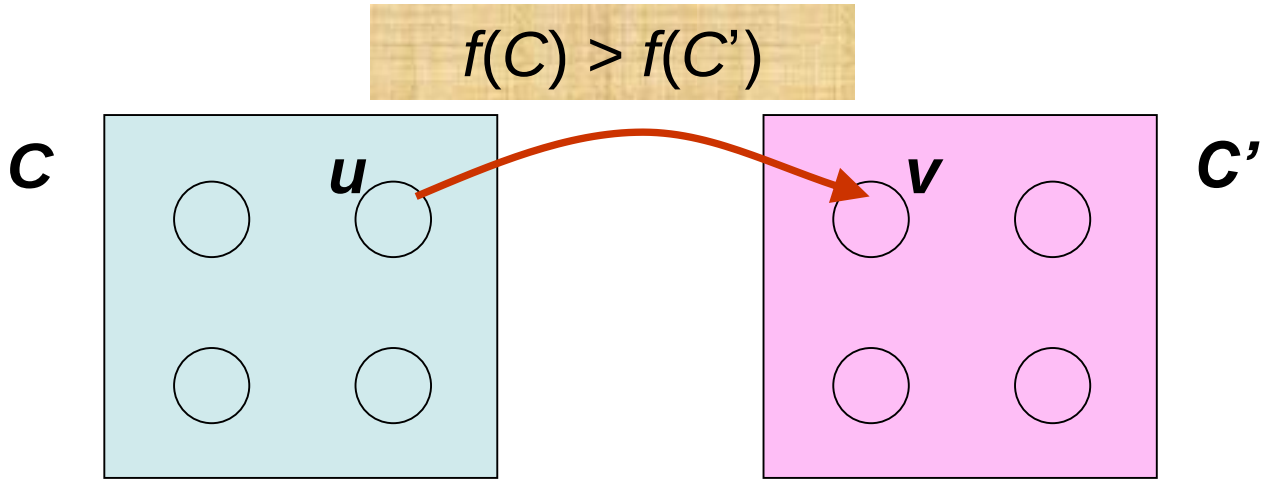
- Lemma 22.13

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

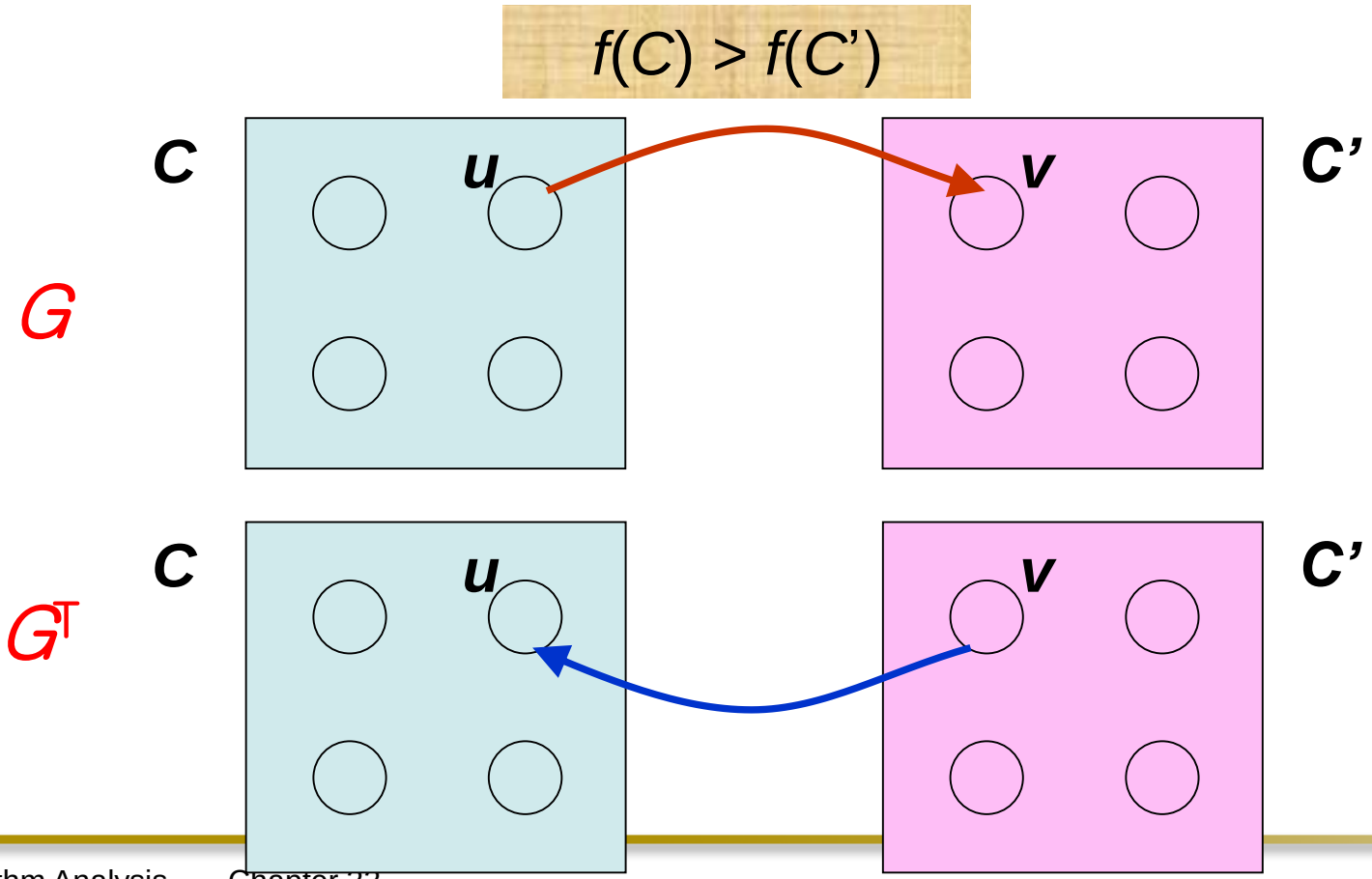


- Note, $f(U) = \max_{u \in U} \{f[u]\}$. That is $f(U)$ is latest finishing time of any vertex in U.

Lemma and Corollary

- Corollary 22.15

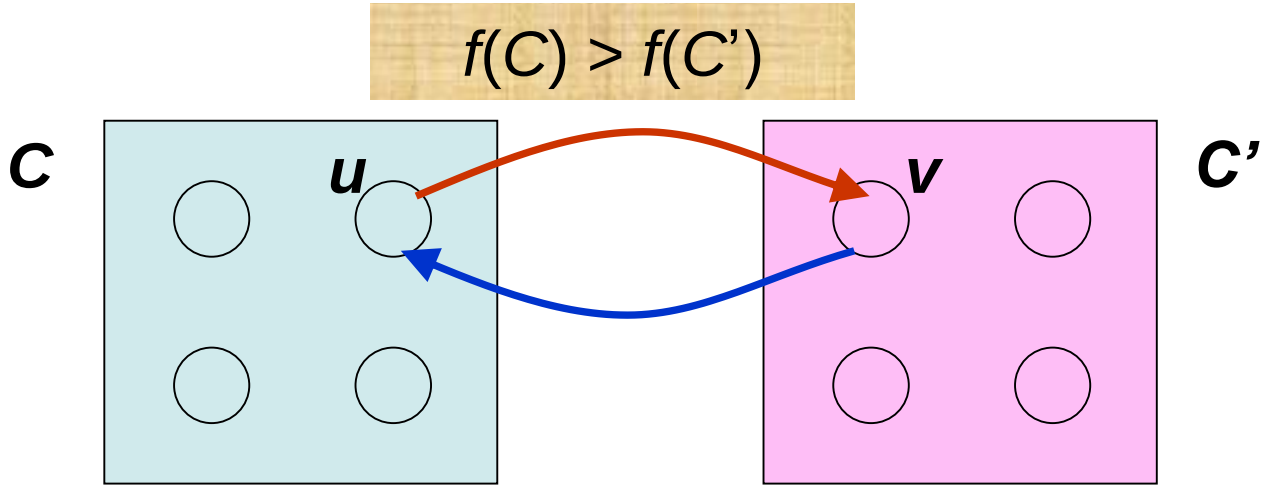
  Let C and C' be distinct SCC's in $G = (V, E)$. Suppose there is an edge $(v,u) \in E^T$, where $u \in C$ and $v \in C'$. Then $f(C) > f(C')$.



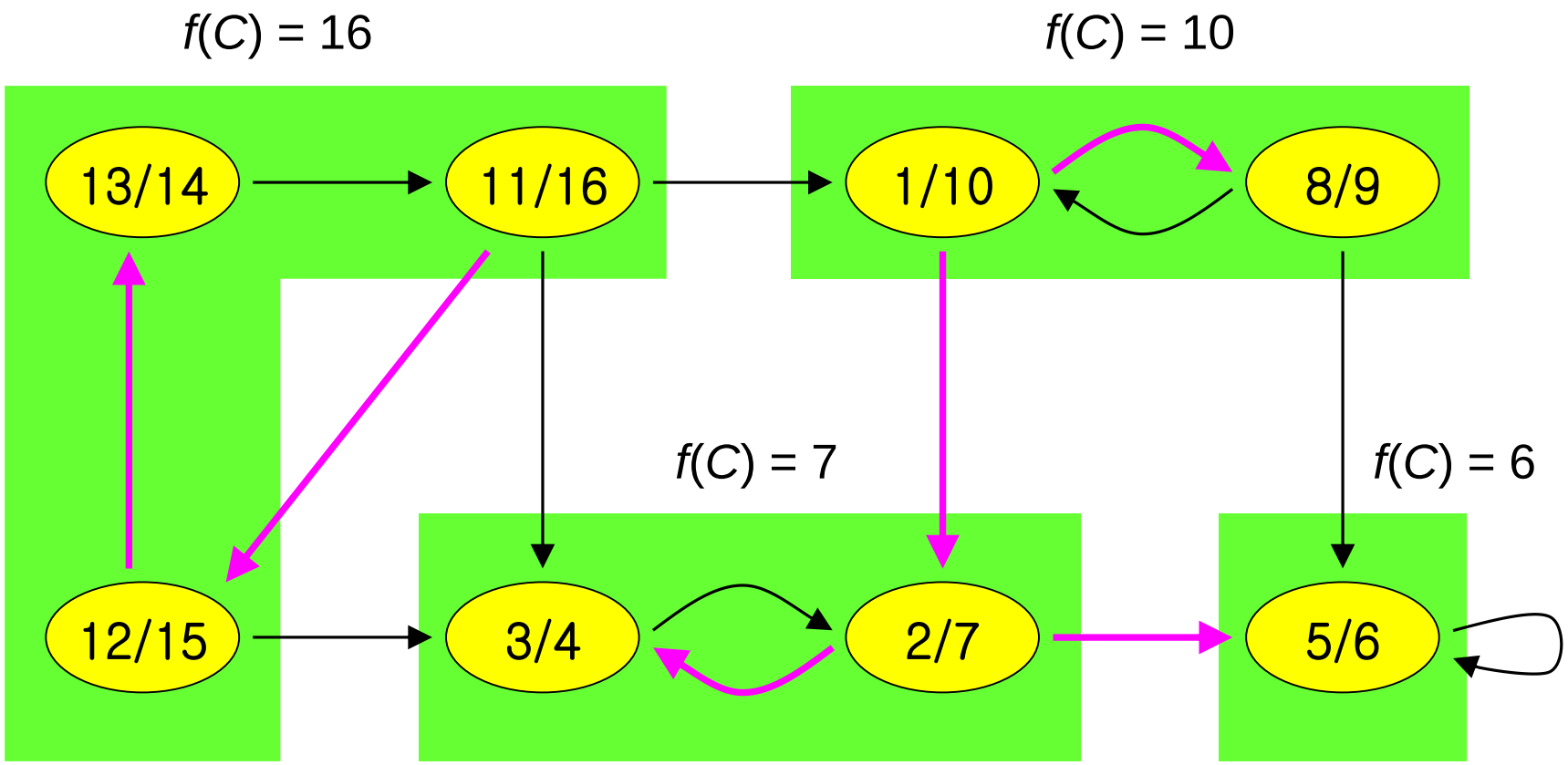
Lemma and Corollary

- 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.

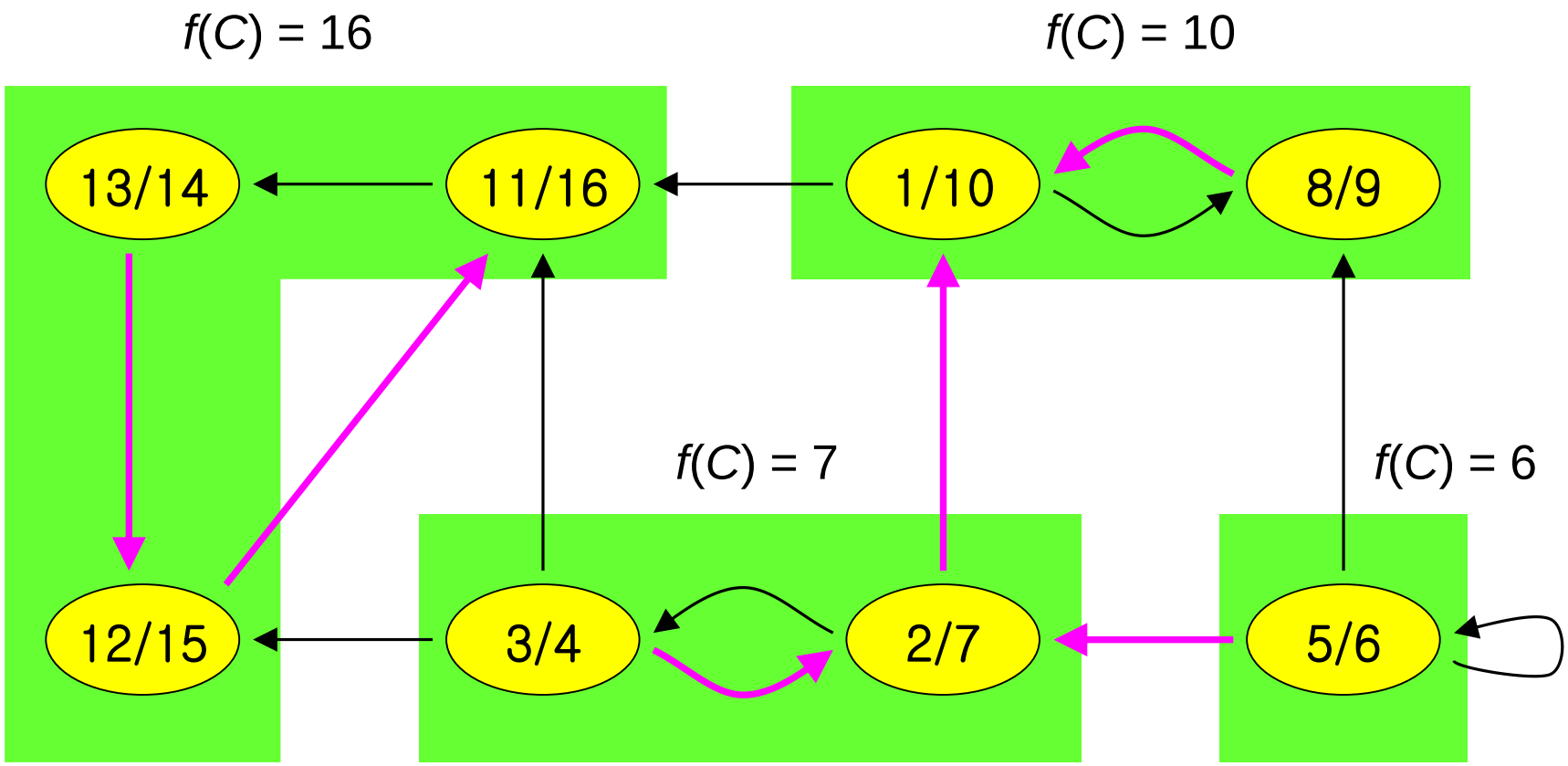


Recall Algorithm



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

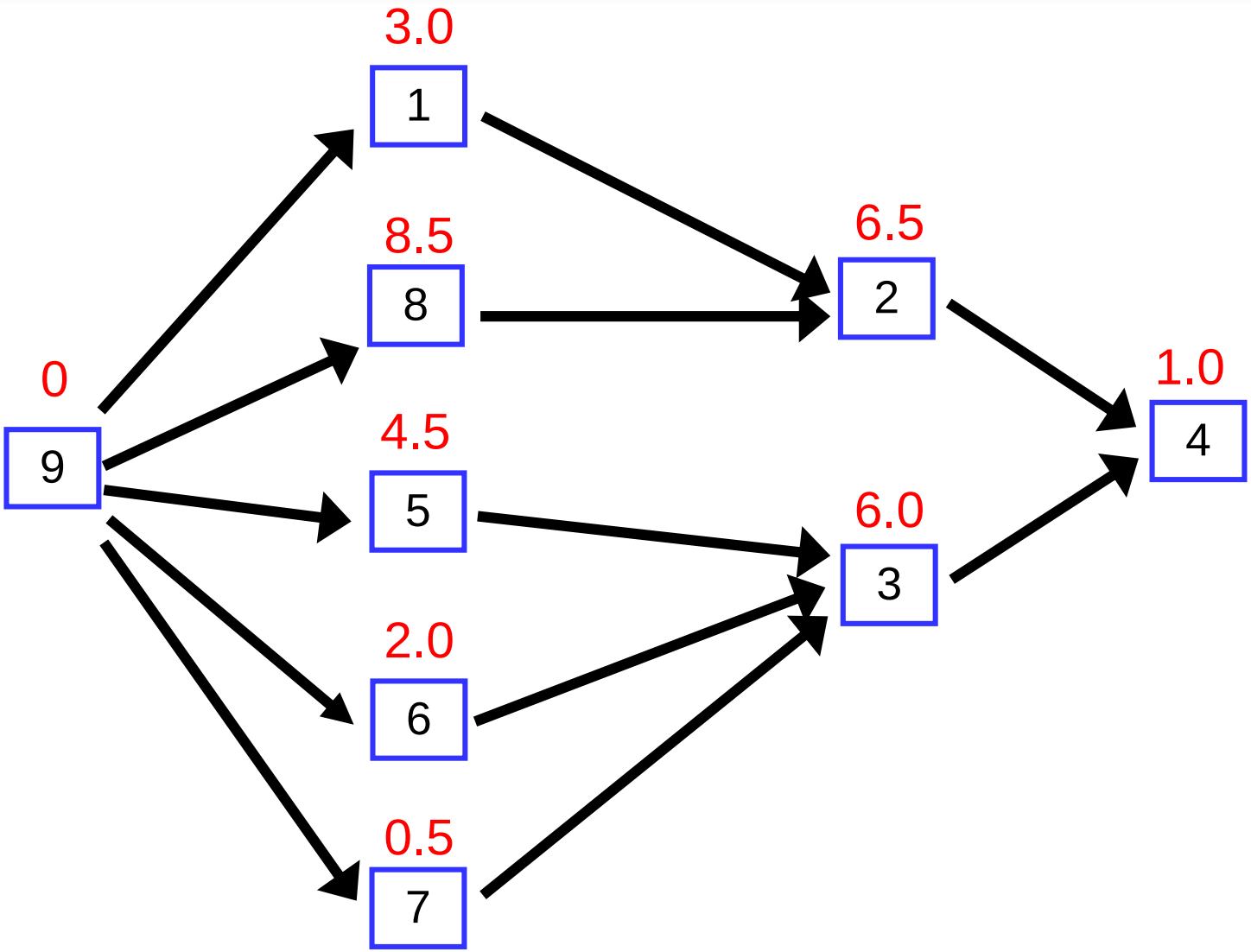
- Say a DAG represents a dependency graph for a series of tasks
  - Each task has a duration
  - We can work on more than one task at a time
- Note there could be several paths from start node to end node
  - Each requires a certain amount of total time
  - One is the longest: the critical path

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*]

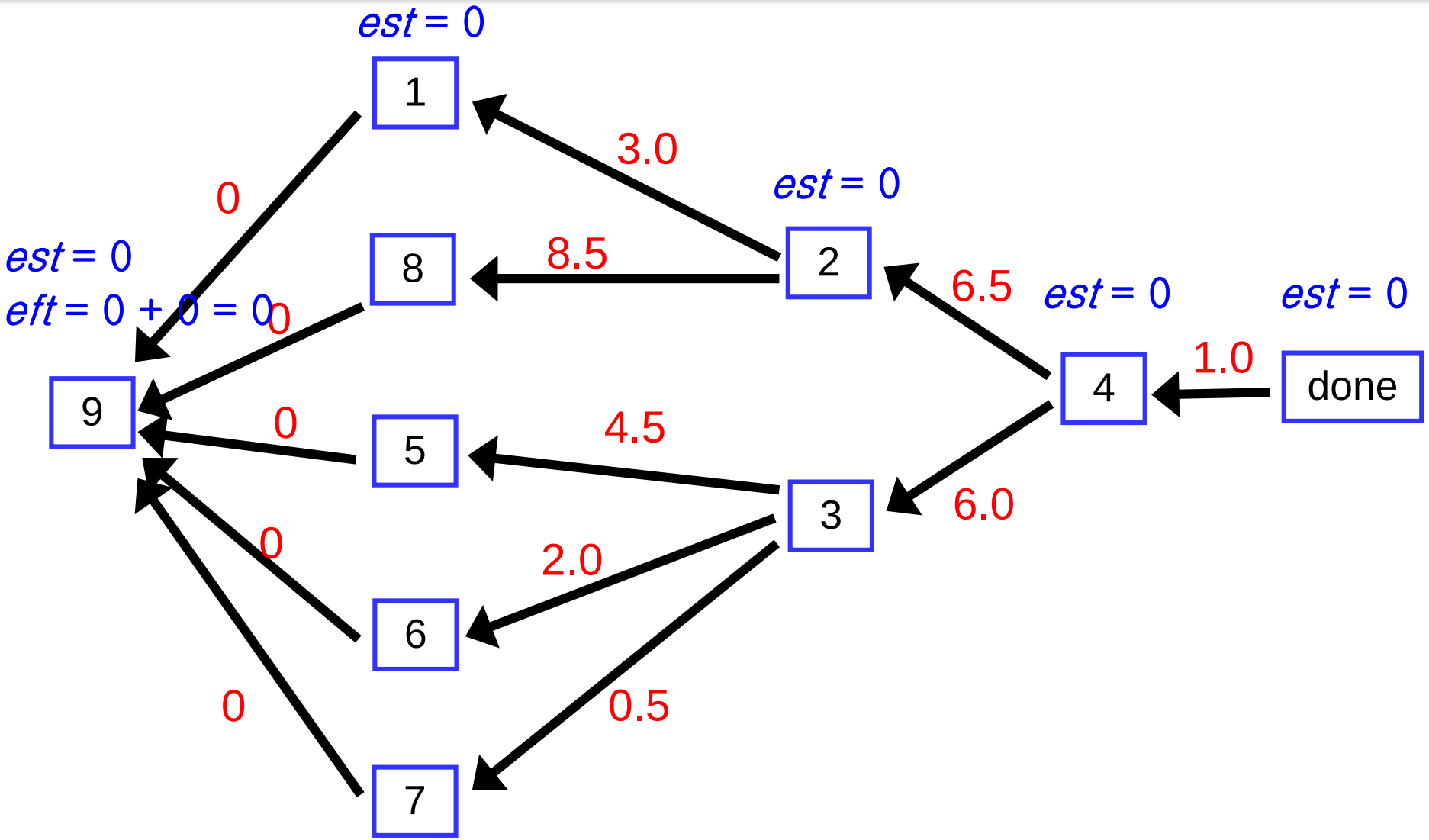
  est[*v*] = *eft*[*w*]

  Crit_Dep[*v*] = *w*

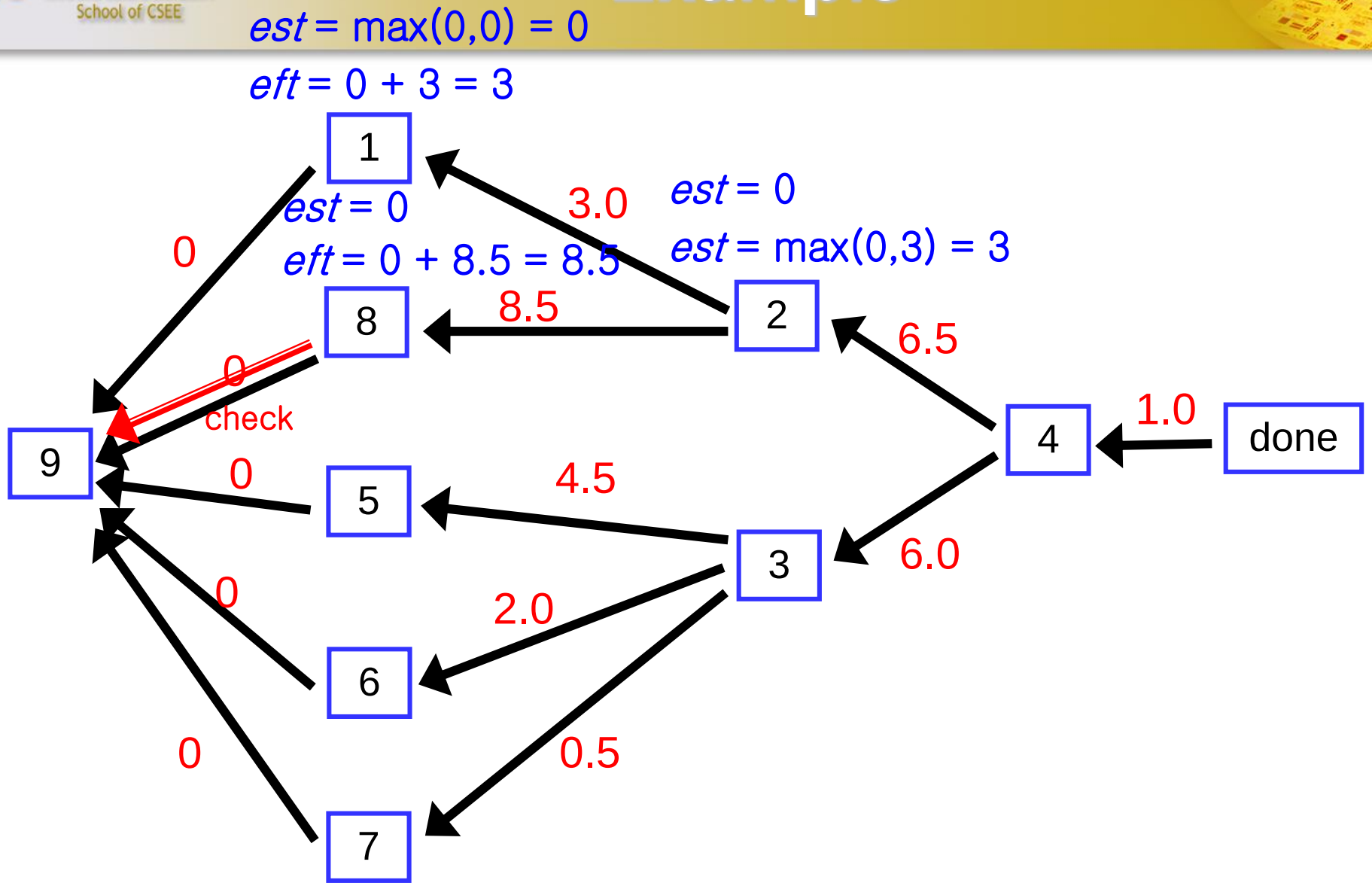
- At postorder processing time, insert

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

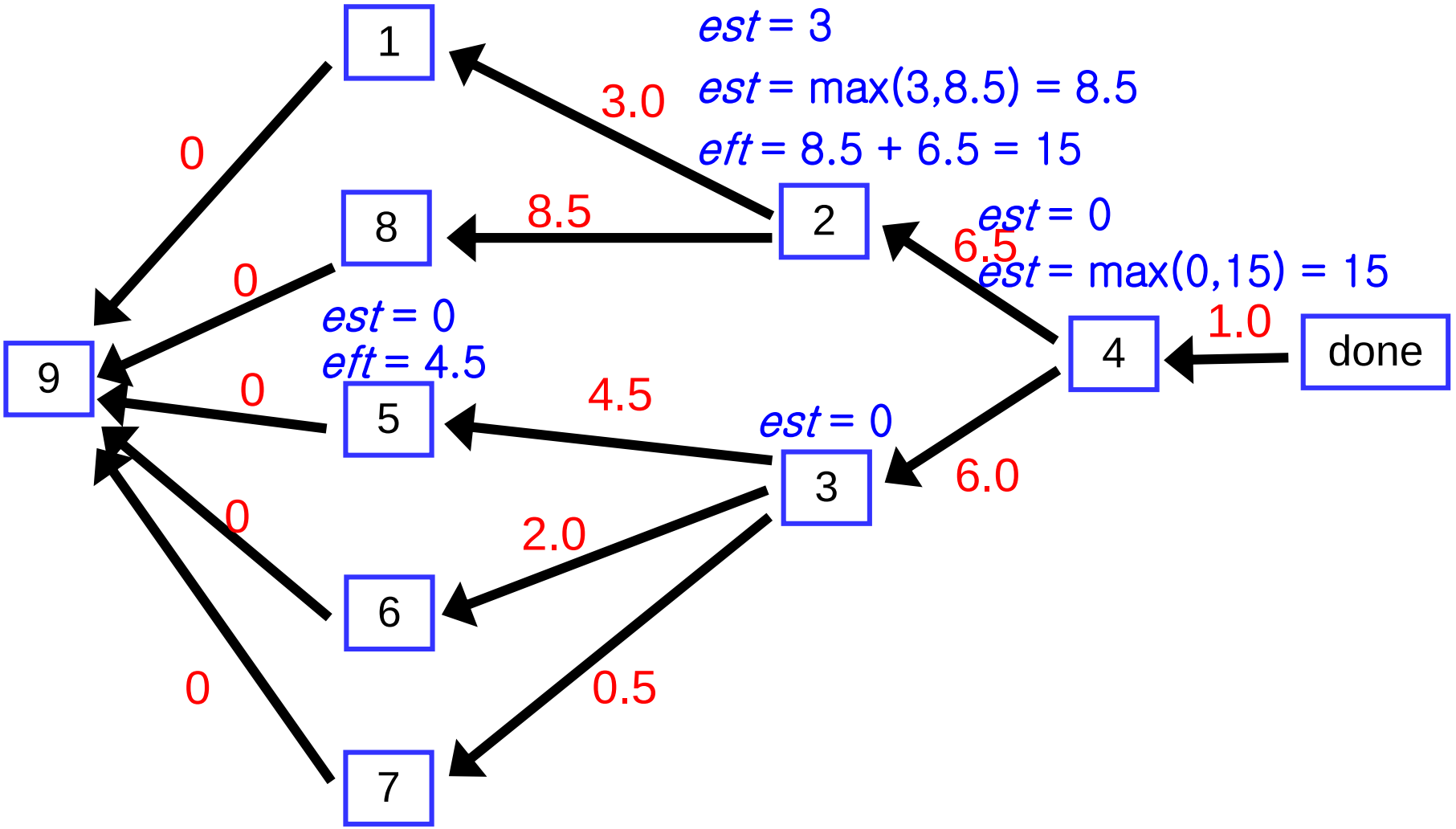
Example


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

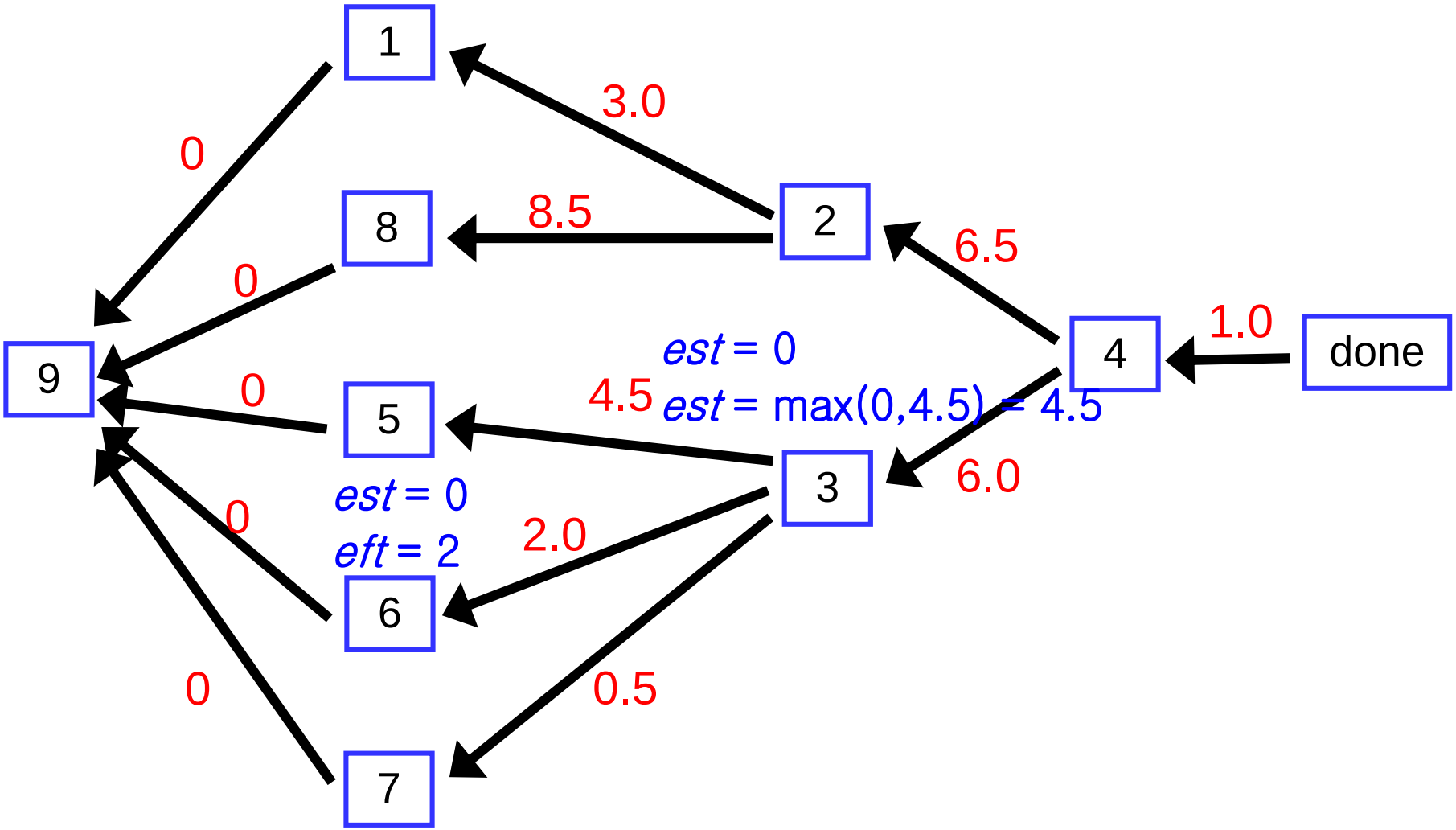
Example


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

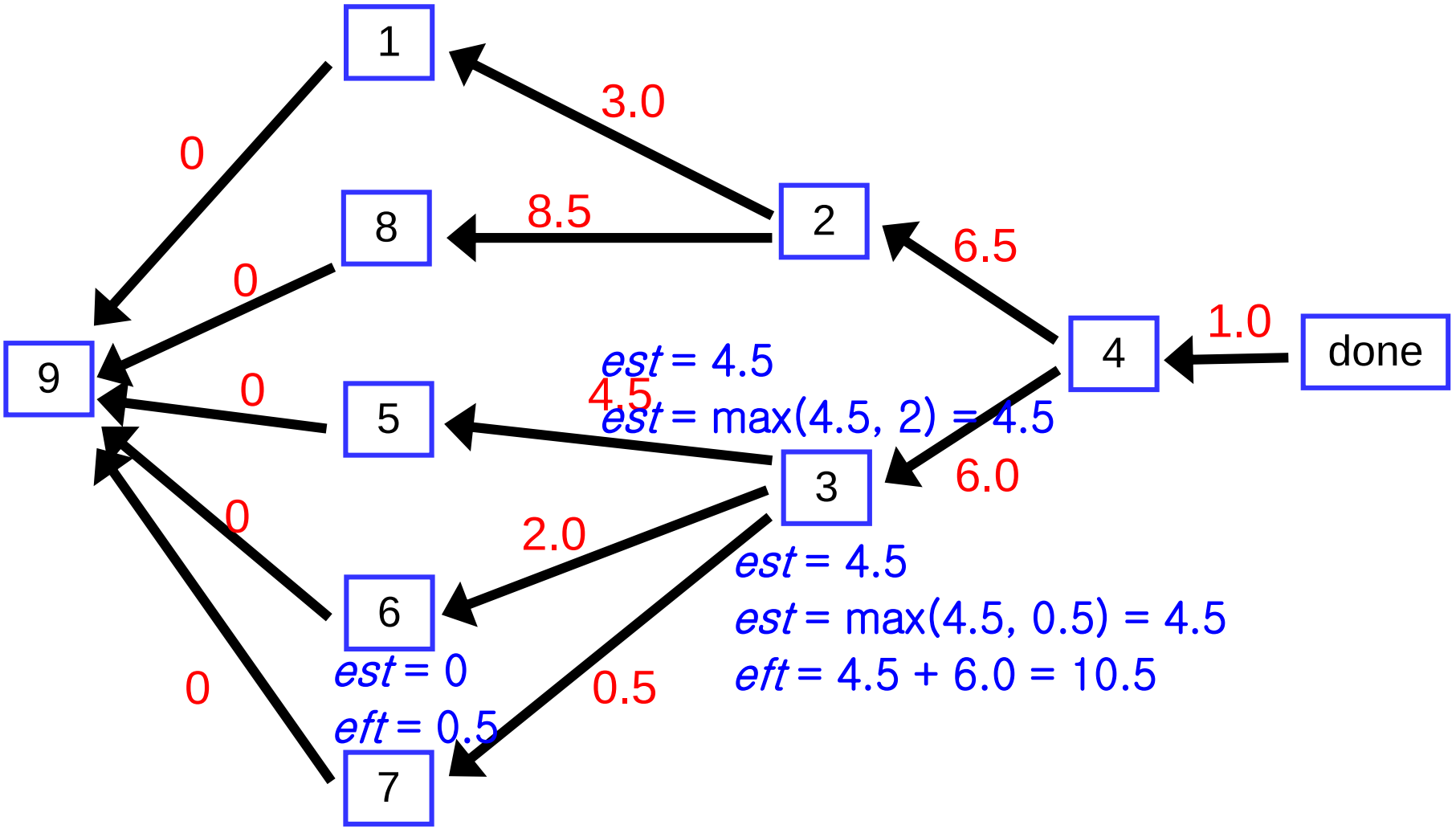
Example


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
est = 0
est = max(0,15) = 15
est = 0
eft = 4.5
est = 0
3.0
8.5
6.5
1.0
0
0
0
4.5
6.0
0
2.0
0
0.5

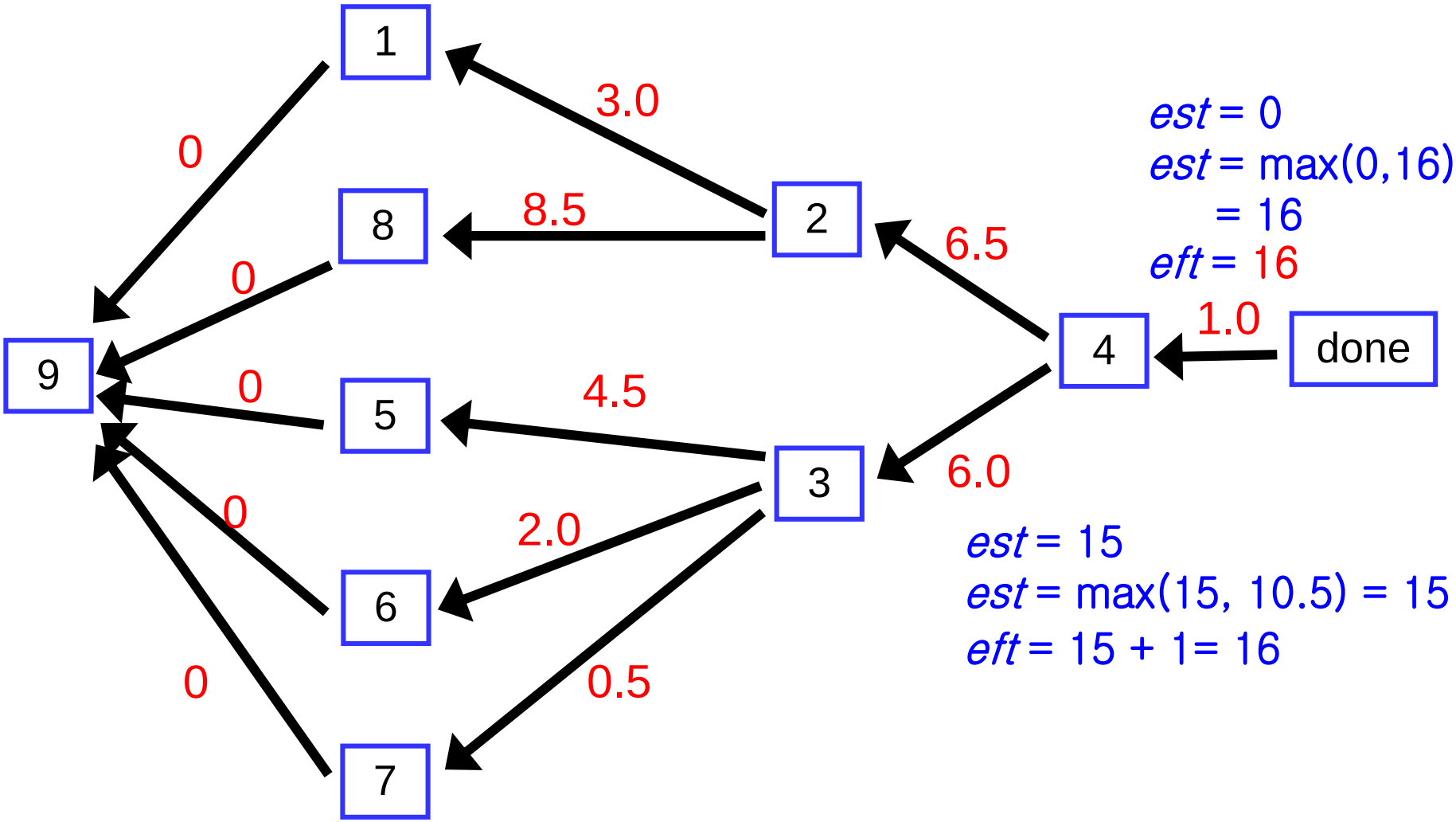
Example


1
2
3
4
5
6
7
8
9
done
3.0
0
8.5
0
6.5
1.0
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

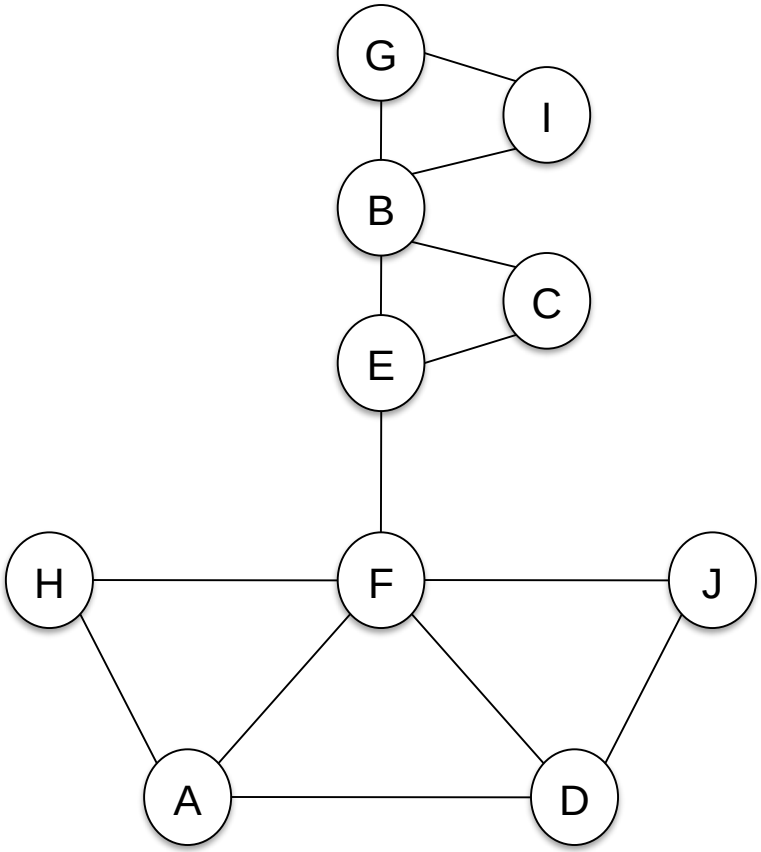
- Problem:
  - If any *one* vertex (and the edges incident upon it) are removed from a connected graph, is the remaining subgraph still connected?
- Biconnected graph:
  - A connected undirected graph *G* is said to be biconnected if it remains connected after removal of any one vertex and the edges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

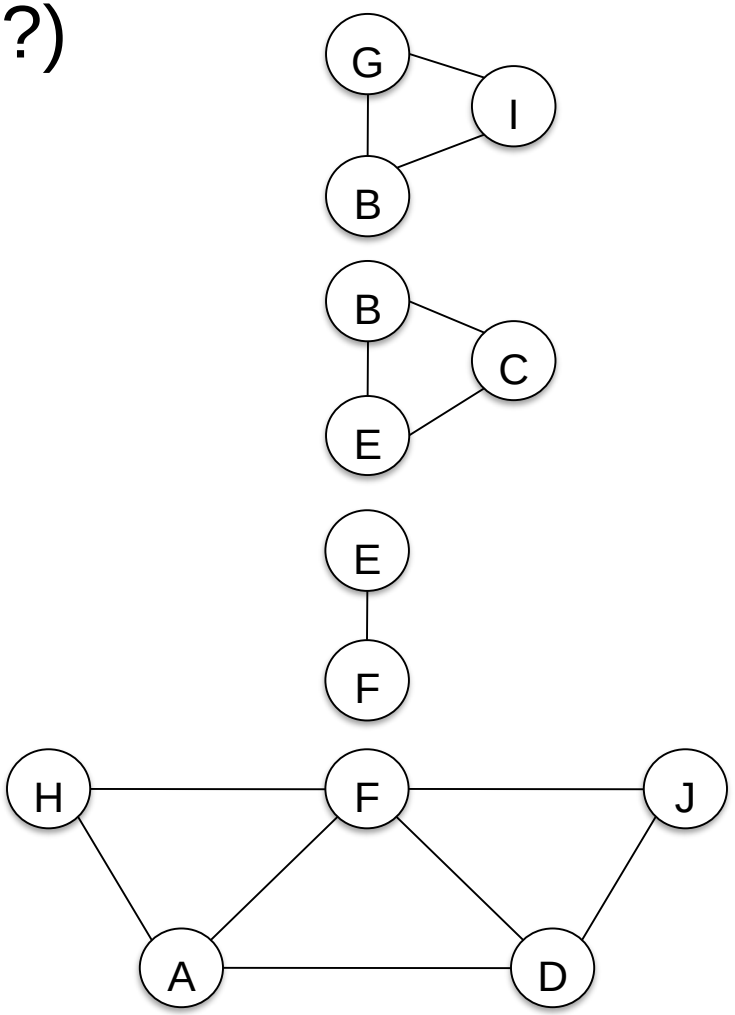
Bi-connected components

- 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:
  - 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

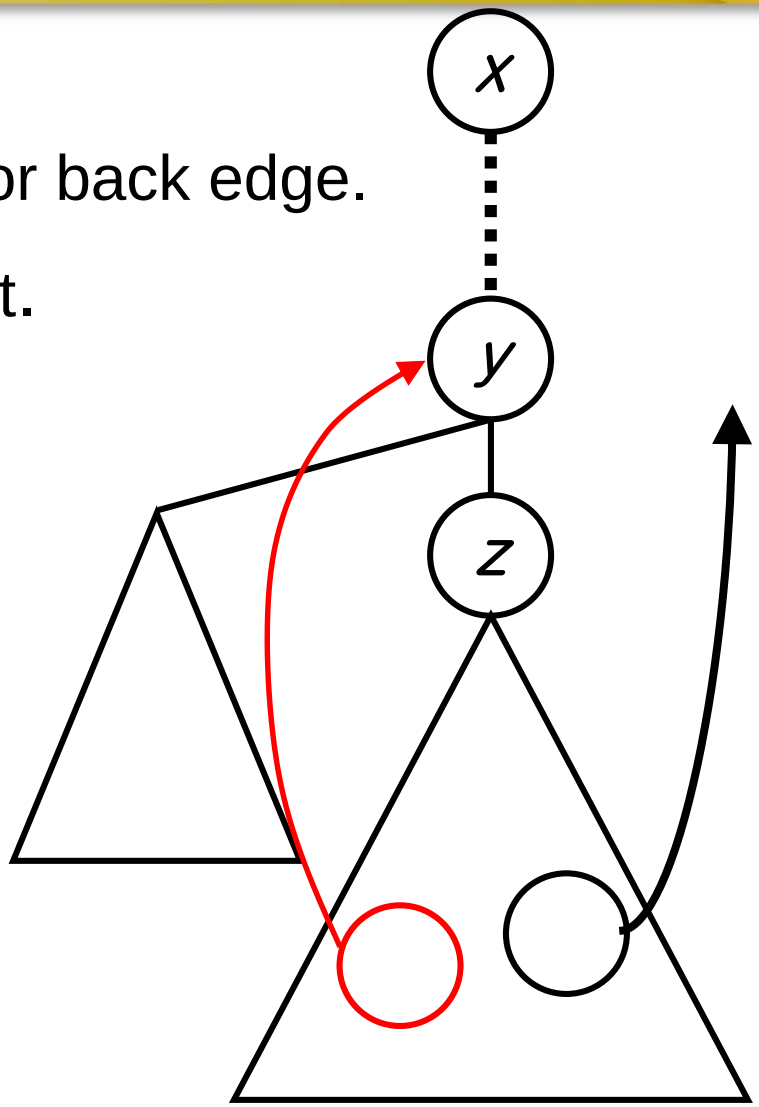
- Fact

  - 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 $d[v]$.

b. When encounteded with back edge vw

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

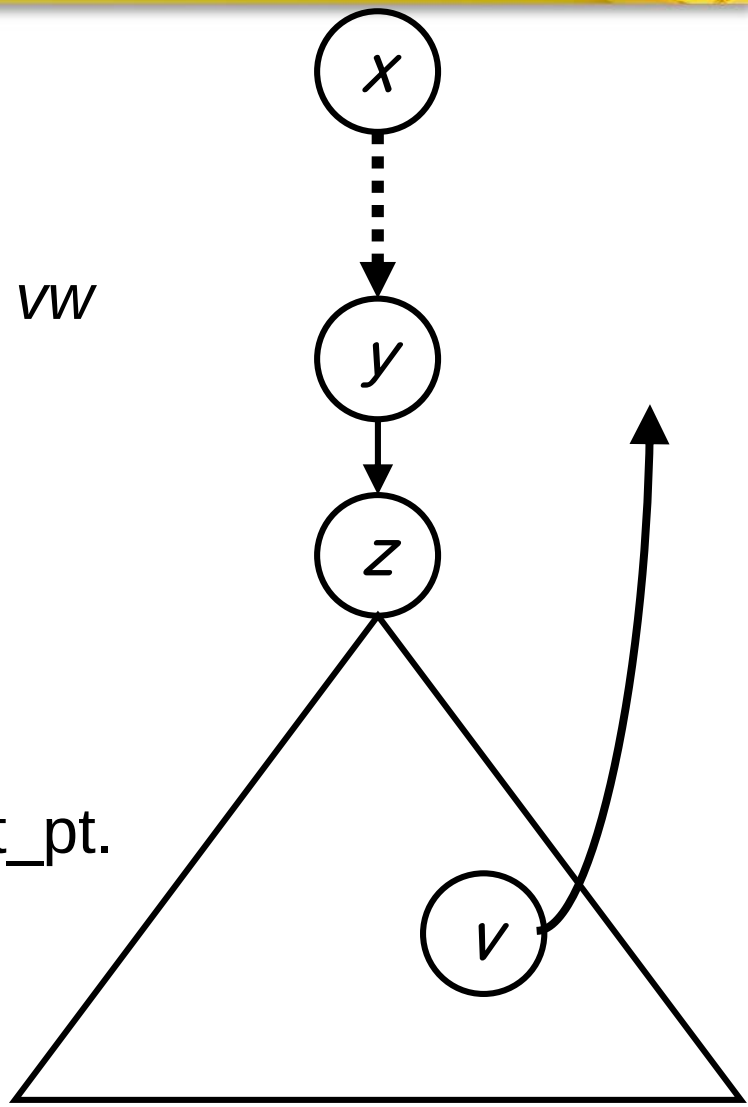
c. When backtracking from v to u,

$back_u = \min(back_u, back_v)$

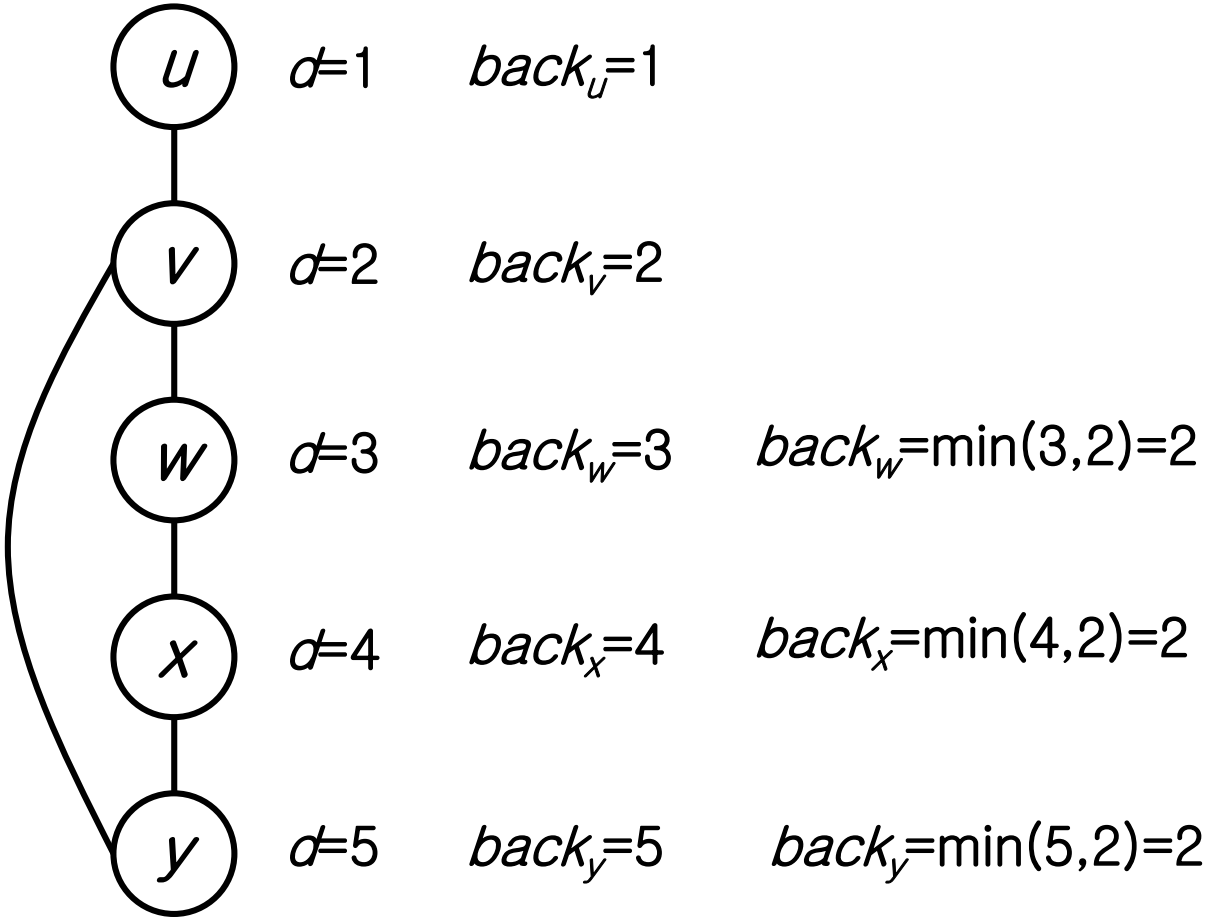
When backing up form z to y,

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

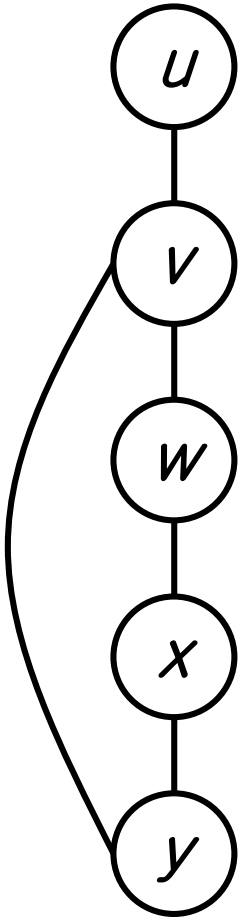
else $back_z < d(y)$. : y isn't art_pt.



Example


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
d=4
back_x=4
back_x=min(4,2)=2
d=5
back_y=5
back_y=min(5,2)=2

Example



d=1 $back_u$=1

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

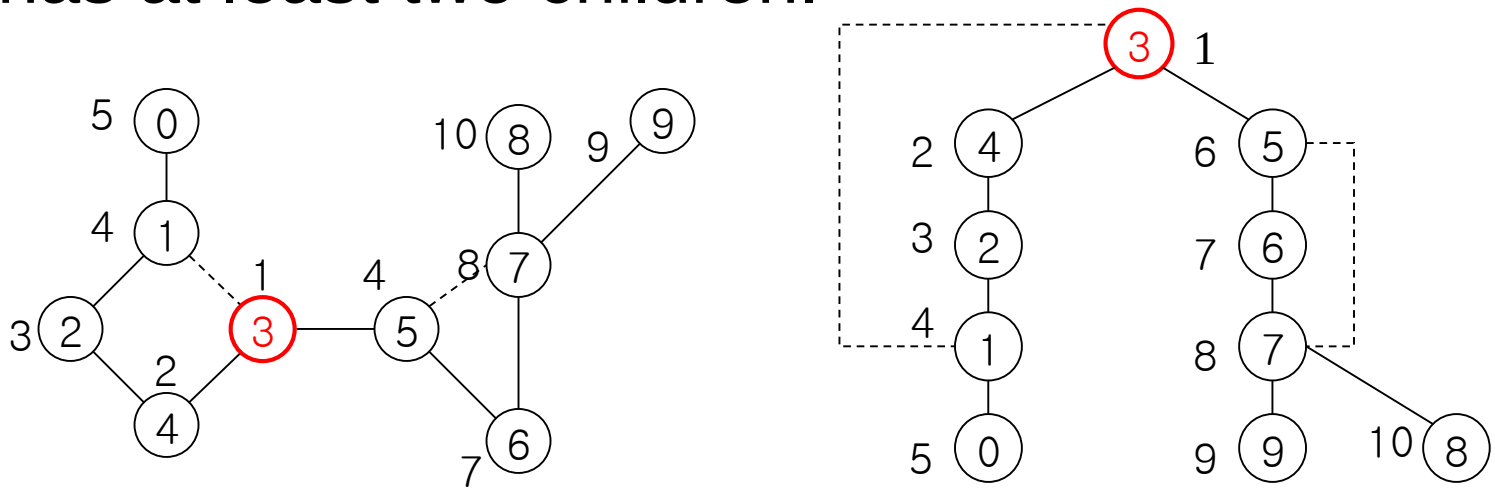
d=3 $back_w$=3 ($back_x$=2) < ($d(w)$=3) : w isn't art_pt

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.