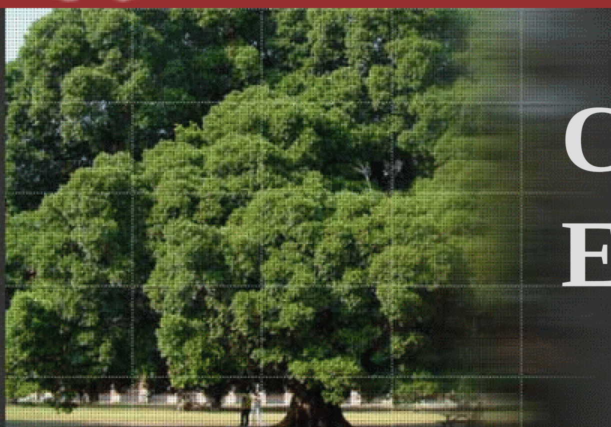
National Cheng Kung University



Chapter 22
Elementary Graph Algorithms

Sun-Yuan Hsieh

謝孫源　教授

成功大學資訊工程學系

Outline

- Graph representation
- Breadth-first search
- Depth-first search
- Topological sort
- Strongly connected components

Graph representation

Given graph $G = (V, E)$.

- May be either directed or undirected.
- Two common ways to represent for algorithms:

1. Adjacency lists.
2. Adjacency matrix.

When expressing the running time of an algorithm, it's often in terms of both $|V|$ and $|E|$.

In asymptotic notation - and *only* in asymptotic notation - we'll drop the cardinality.

Example: $O(V+E)$.

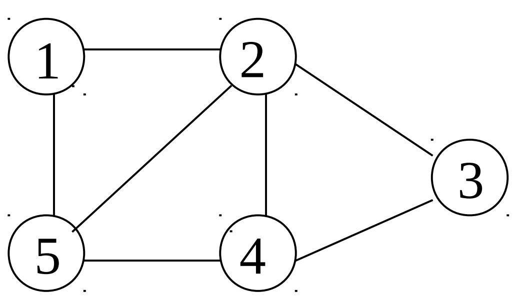
Adjacency lists

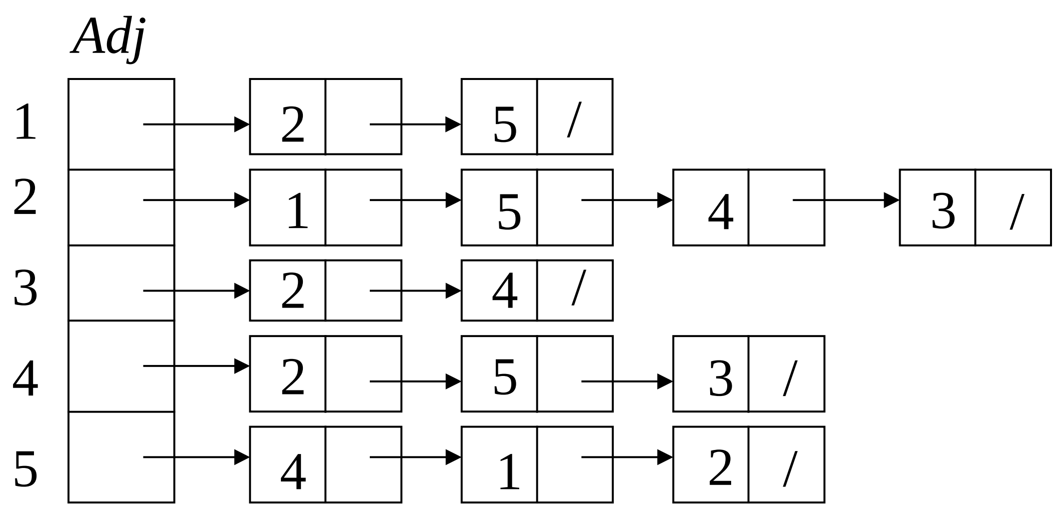
Array *Adj* of $|V|$ lists, one per vertex.

Vertex *u*'s list has all vertices *v* such that $(u, v) \in E$.

(Works for both directed and undirected graphs.)

Example: For an undirected graph:





Adjacency lists

If edges have *weights*, can put the weights in the lists.

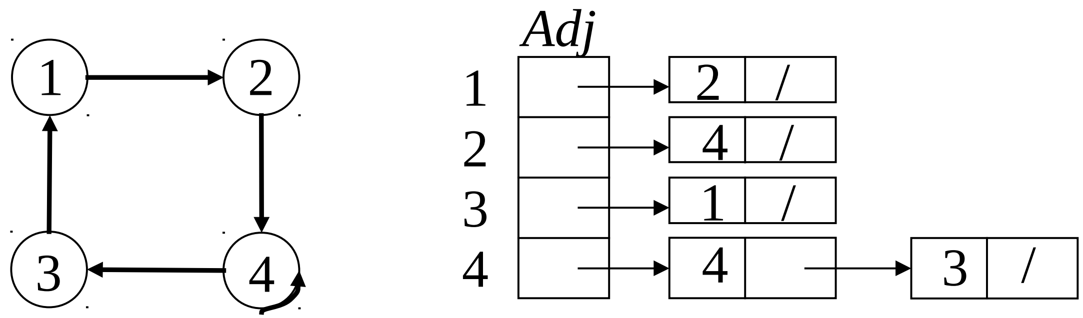
Weight: $w : E \to \boldsymbol{R}$

Space: $\Theta(V+E)$

Time: to list all vertices adjacent to u: $\Theta(\text{degree}(u))$

Time: to determine if $(u, v) \in E$: $O(\text{degree}(u))$

Example: **For a directed graph:**

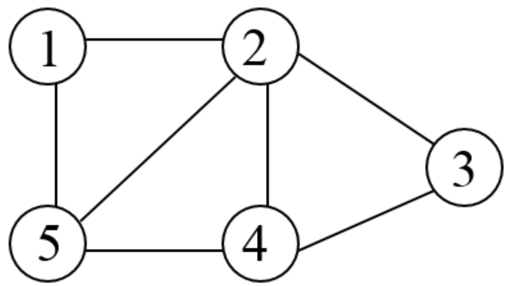


Same asymptotic space and time

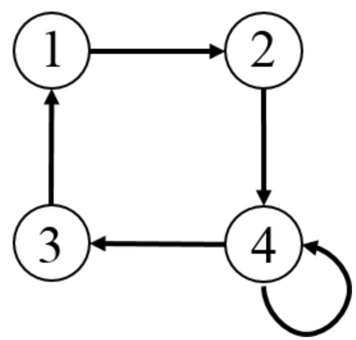
Adjacency matrix

$|V| \times |V|$ matrix $A = (a_{ij})$

$$a_{ij} = \begin{cases} 1, & \text{if}(i, j) \in E \\ 0, & \text{otherwise} \end{cases}$$



$$\begin{array}{c|ccccc|} & 1 & 2 & 3 & 4 & 5 \\ \hline 1 & 0 & 1 & 0 & 0 & 1 \\ 2 & 1 & 0 & 1 & 1 & 1 \\ 3 & 0 & 1 & 0 & 1 & 0 \\ 4 & 0 & 1 & 1 & 0 & 1 \\ 5 & 1 & 1 & 0 & 1 & 0 \\ \hline \end{array}$$



$$\begin{array}{c|cccc|} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 1 & 0 & 0 \\ 2 & 0 & 0 & 0 & 1 \\ 3 & 1 & 0 & 0 & 0 \\ 4 & 0 & 0 & 1 & 1 \\ \hline \end{array}$$

Space: $\Theta(V^2)$
Time: to list all vertices adjacent to u: $\Theta(V)$
Time: to determine if $(u, v) \in E$: $\Theta(1)$
Can store weights instead of bits for weighted graph

Breadth-first search

Input: Graph $G = (V,E)$, either directed or undirected, and ***source vertex*** $s \in V$

Output: $d[v]$ = distance (smallest # of edges) from s to v, for all $v \in V$

Also $\pi[v] = u$ such that (u, v) is last edge on a shortest path $s \to v$

- u is v's ***predecessor***
- set of edges $\{(\pi[v], v) : v \neq s\}$ forms a tree

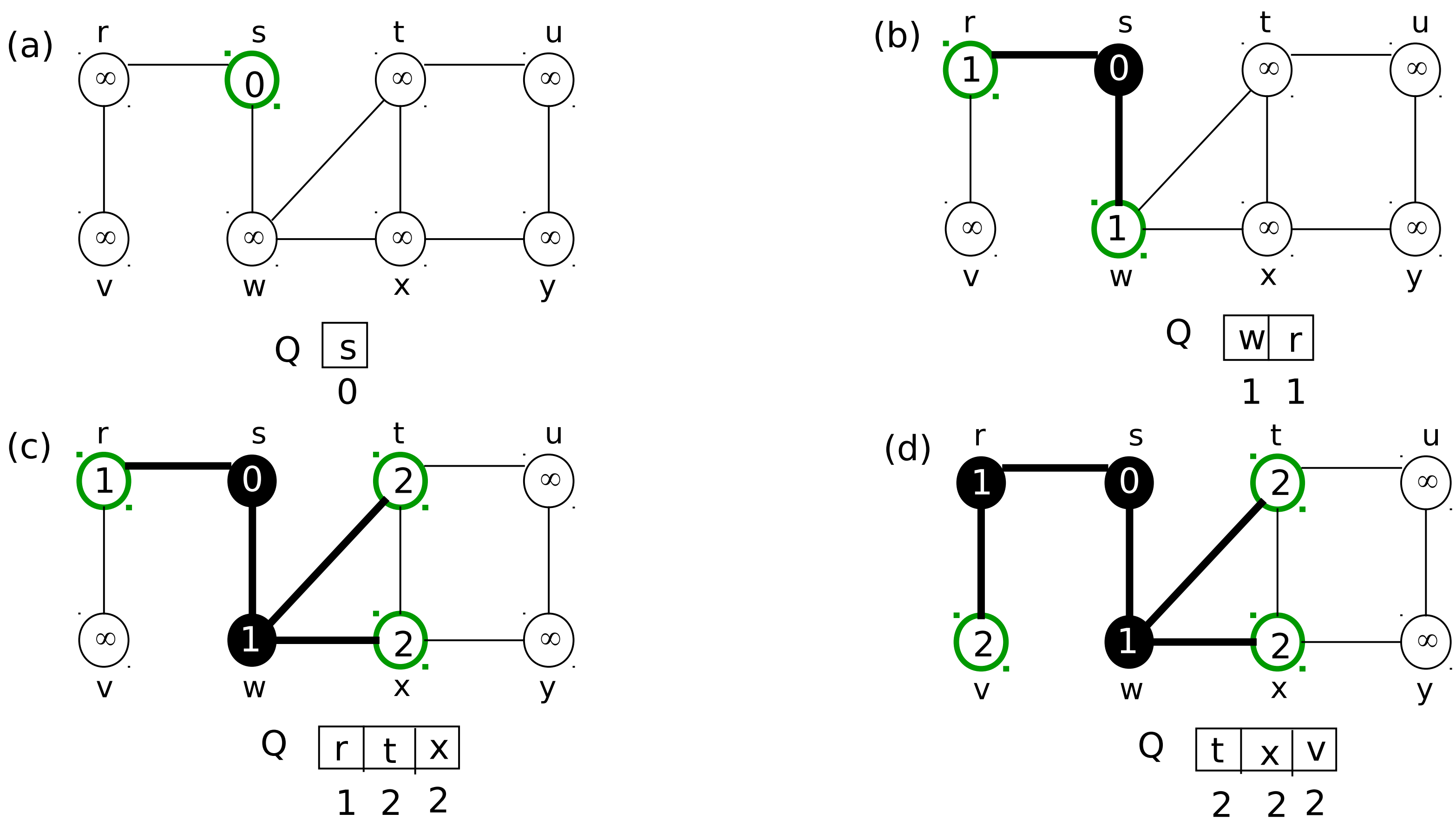
Breadth-first search

Idea: Send a wave out from *s*.

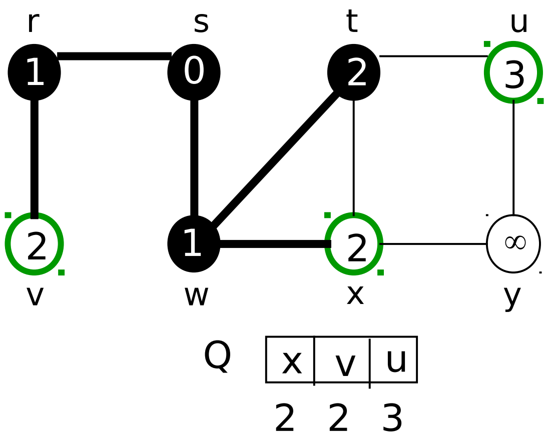
- First hits all vertices 1 edge from *s*.
- From there, hits all vertices 2 edges from *s*.
- Etc.

Use FIFO queue Q to maintain wavefront.

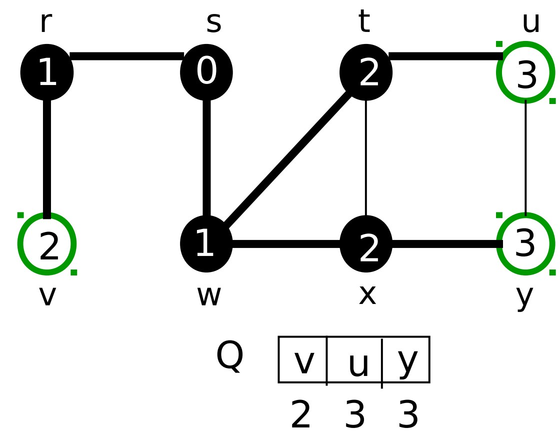
- $v \in Q$ if and only if wave has hit v but has not come out of v yet.


(a)
r s t u
∞ 0 ∞ ∞
∞ ∞ ∞ ∞
v w x y
Q s
0
(b)
r s t u
1 0 ∞ ∞
∞ 1 ∞ ∞
v w x y
Q w r
1 1
(c)
r s t u
1 0 2 ∞
∞ 1 2 ∞
v w x y
Q r t x
1 2 2
(d)
r s t u
1 0 2 ∞
2 1 2 ∞
v w x y
Q t x v
2 2 2

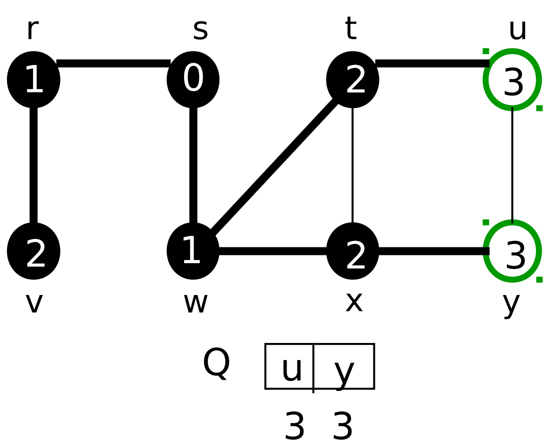
(e)


r s t u
1 0 2 3
2 1 2 ∞
v w x y
Q
x v u
2 2 3

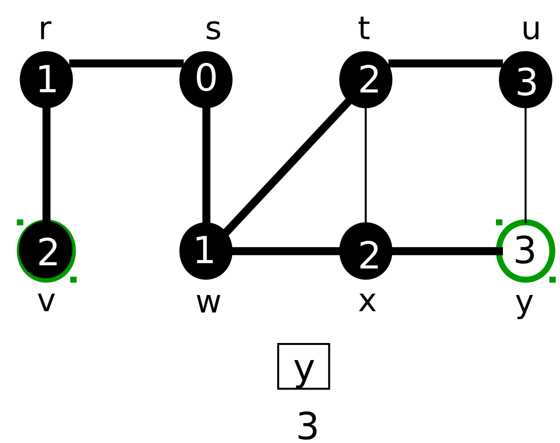
(f)


r s t u
1 0 2 3
2 1 2 3
v w x y
Q
v u y
2 3 3

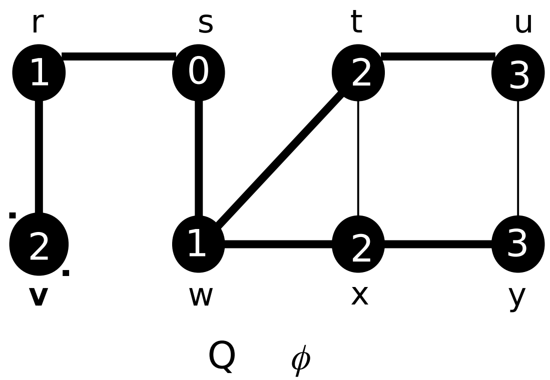
(g)


r s t u
1 0 2 3
2 1 2 3
v w x y
Q
u y
3 3

(h)

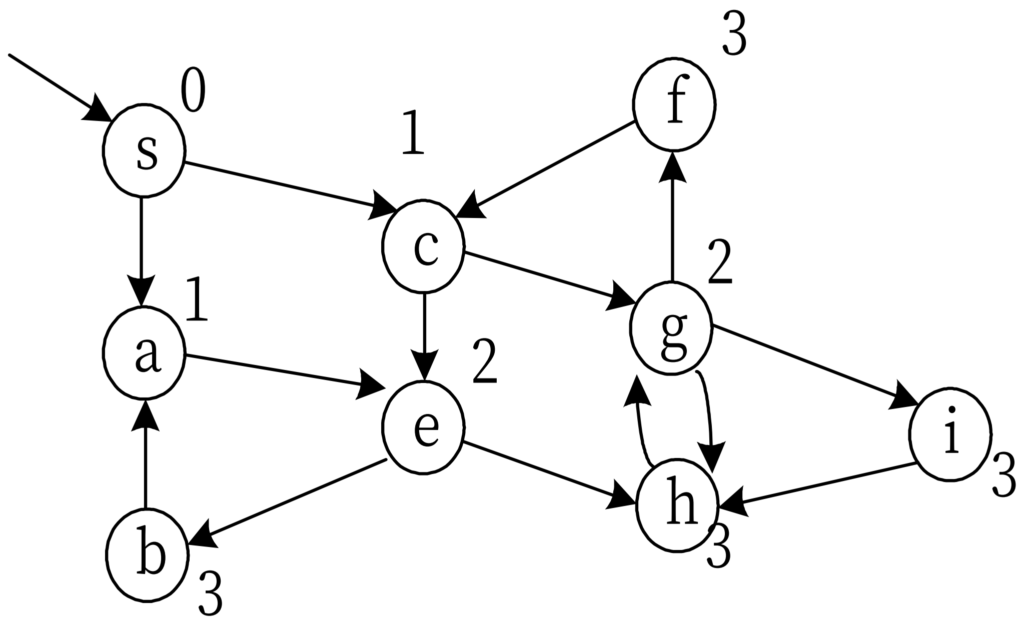

r s t u
1 0 2 3
2 1 2 3
v w x y
y
3

(i)



BFS(V, E, s)
for each $u \in V - \{s\}$
    do $d[u] \leftarrow \infty$
$d[s] \leftarrow 0$
$Q \leftarrow \varphi$
ENQUEUE(Q, s)
While $Q \neq \varphi$
  do $u \leftarrow$ DEQUEUE(Q)
    for each $v \in Adj[u]$
        do if $d[v] = \infty$
            then $d[v] \leftarrow d[u] + 1$
                ENQUEUE(Q, v)

Example: **directed graph**



Can show that *Q* consists of vertices with *d* values.

i   *i*   *i* …. *i*   *i*+1   *i*+1   …   *i*+1

- Only 1 or 2 values.
- If 2, differ by 1 and all smallest are first.

Since each vertex gets a finite d value at most once, values assigned to vertices are monotonically increasing over time.

BFS may not reach all vertices.

Time = $O(V + E)$.

- $O(V)$ because every vertex enqueued at most once.
- $O(E)$ because every vertex dequeued at most once and we examine (u, v) only when u is dequeued. Therefore, every edge examined at most once if directed, at most twice if undirected.

Depth-first search

Input: $G = (V, E)$, directed or undirected. No source vertex given!

Output: 2 ***timestamps*** on each vertex:

- ▷ $d[v]$ = **discovery time**
- ▷ $f[v]$ = **finishing time**

These will be useful for other algorithms later on.

Can also compute $\pi[v]$.

Will methodically explore *every* edge.

- Start over from different vertices as necessary.

As soon as we discover a vertex, explore from it.

- Unlike BFS, which puts a vertex on a queue so that we explore from it later.

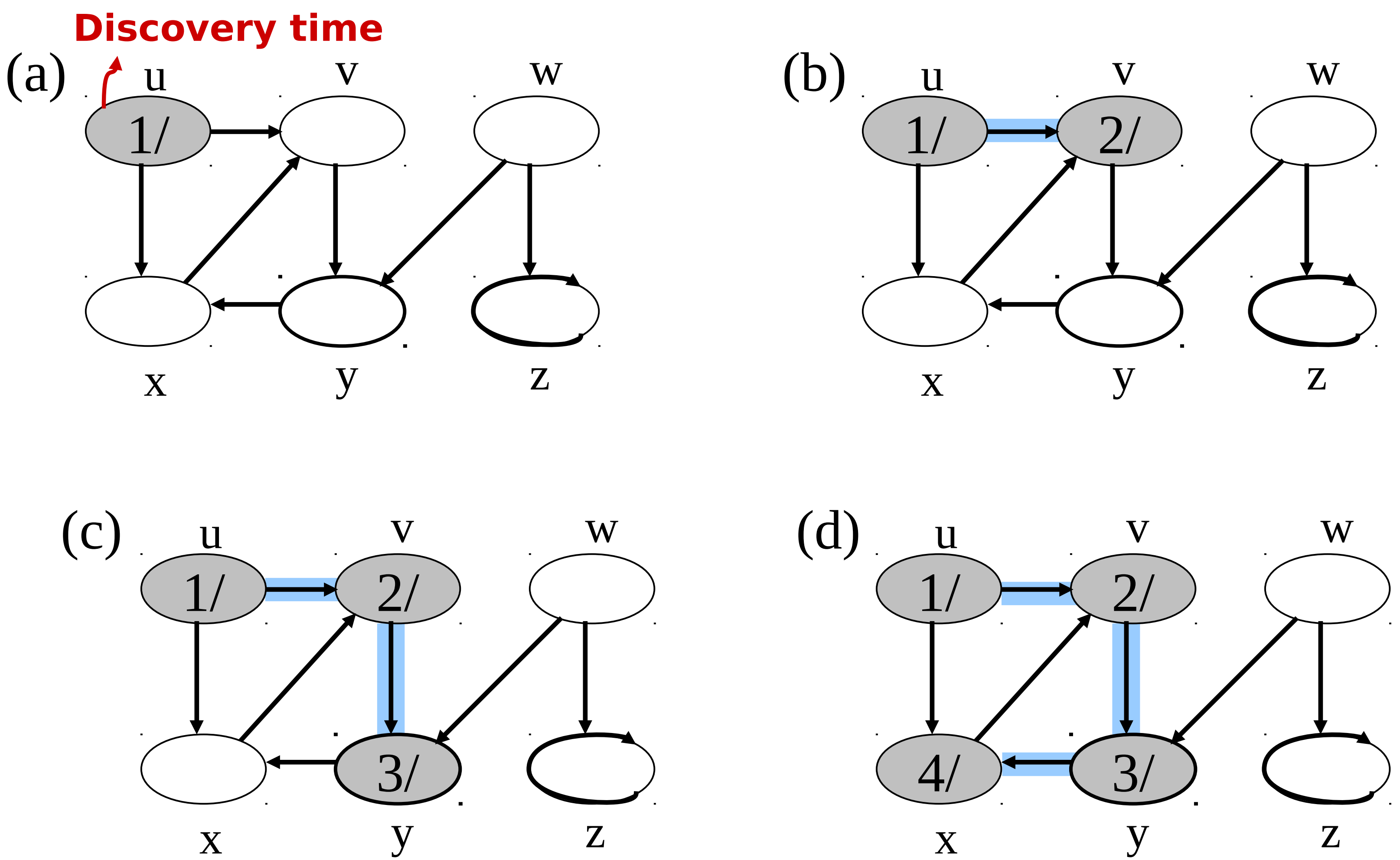
As DFS progresses, every vertex has a ***color***:

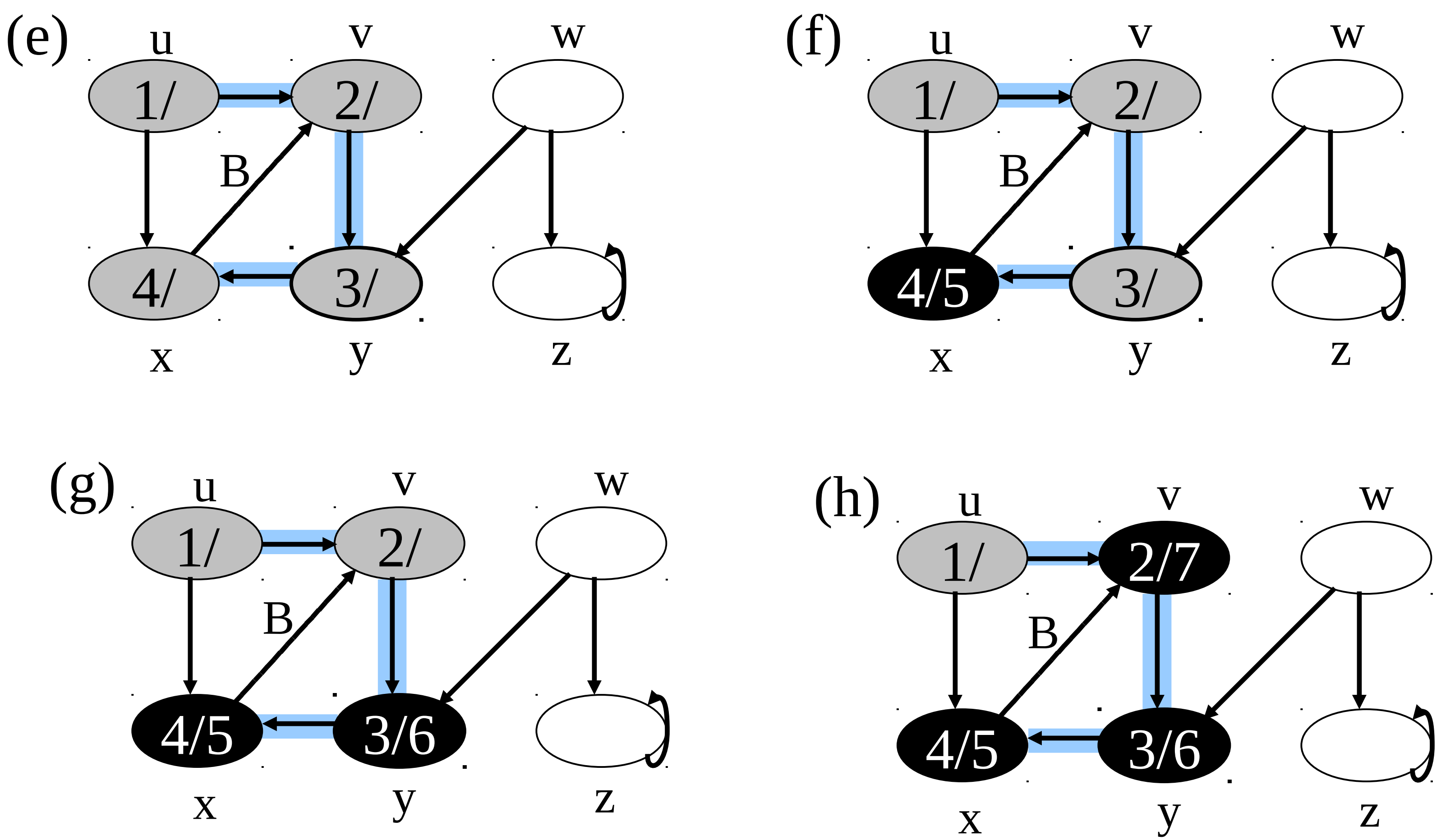
- WHITE = undiscovered
- GRAY = discovered, but not finished (not done exploring from it)
- BLACK = finished (have found everything reachable from it)

Discovery and finish times:

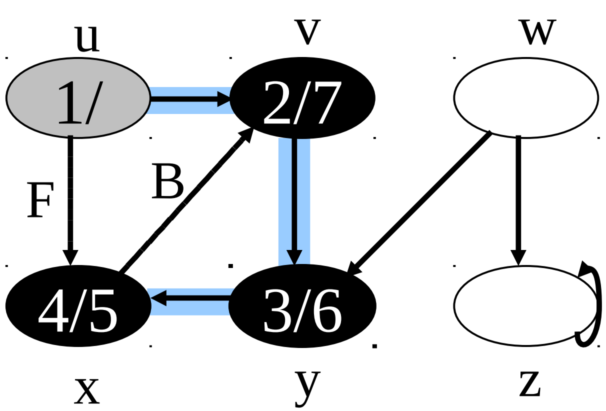
- Unique integers from 1 to $2|V|$.
- For all v, $d[v] < f[v]$.

In other words, $1 \le d[v] < f[v] \le 2|V|$.

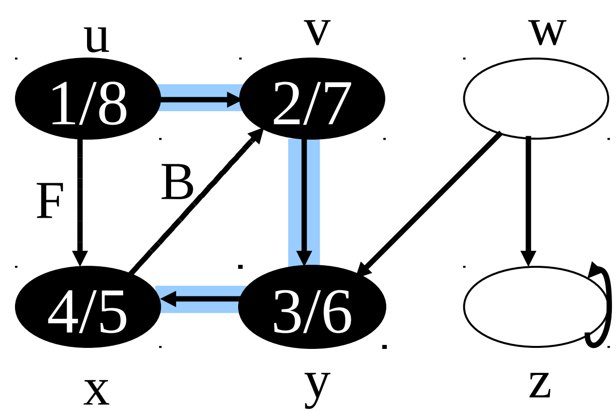

Discovery time
(a)
u
v
w
1/
x
y
z
(b)
u
v
w
1/
2/
x
y
z
(c)
u
v
w
1/
2/
3/
x
y
z
(d)
u
v
w
1/
2/
4/
3/
x
y
z


(e)
u
v
w
1/
2/
B
4/
3/
x
y
z
(f)
u
v
w
1/
2/
B
4/5
3/
x
y
z
(g)
u
v
w
1/
2/
B
4/5
3/6
x
y
z
(h)
u
v
w
1/
2/7
B
4/5
3/6
x
y
z

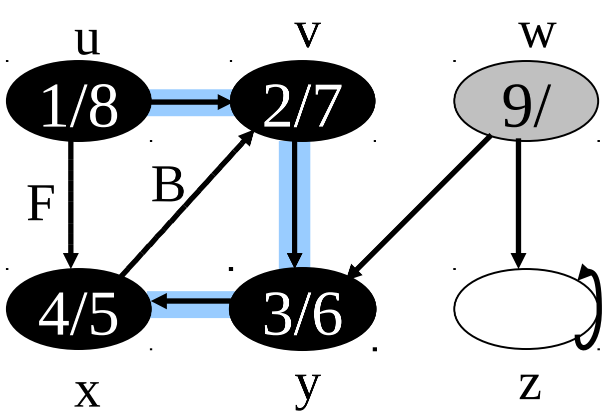
(i)


u
v
w
1/
2/7
F
B
4/5
3/6
x
y
z

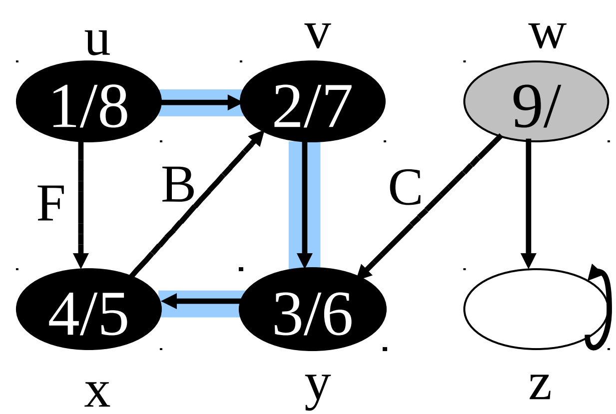
(j)

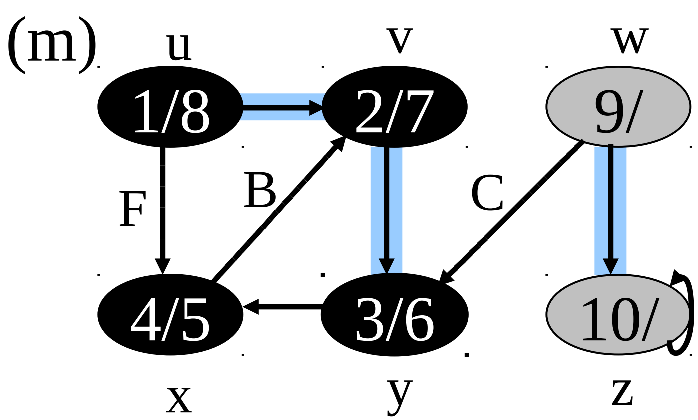

u
v
w
1/8
2/7
F
B
4/5
3/6
x
y
z

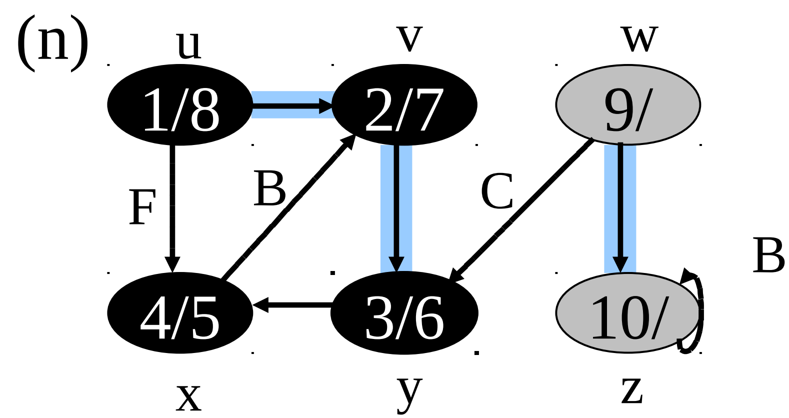
(k)

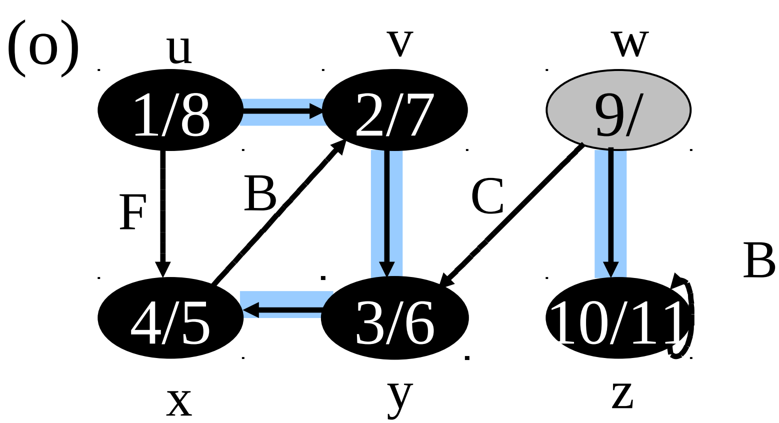

u
v
w
1/8
2/7
9/
F
B
4/5
3/6
x
y
z

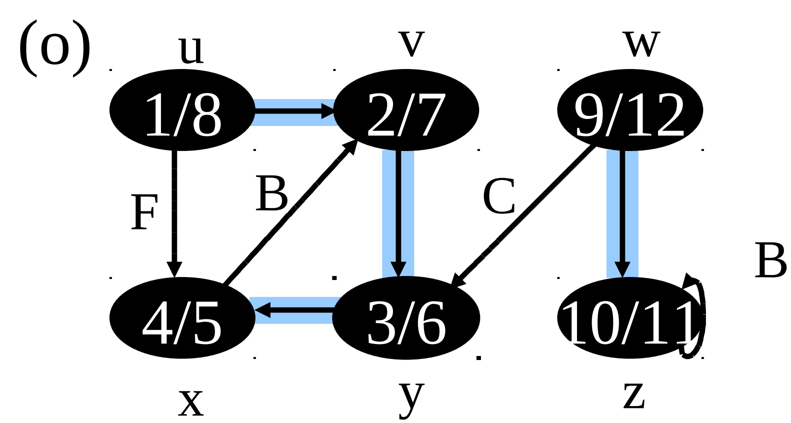
(l)


u
v
w
1/8
2/7
9/
F
B
C
4/5
3/6
x
y
z


(m)
u
v
w
1/8
2/7
9/
F
B
C
4/5
3/6
10/
x
y
z


(n)
u
v
w
1/8
2/7
9/
F
B
C
B
4/5
3/6
10/
x
y
z


(o)
u
v
w
1/8
2/7
9/
F
B
C
B
4/5
3/6
10/11
x
y
z


(o)
u
v
w
1/8
2/7
9/12
F
B
C
B
4/5
3/6
10/11
x
y
z

Pseudocode: Uses a global timestamp *time*.

DFS(V,E)

for each $u \in V$

    do *color*[u] $\leftarrow$ WHITE

time $\leftarrow$ 0

for each $u \in V$

    do if *color*[u] = WHITE

        then DFS - Visit(u)

DFS - Visit(u)

$color[u] \leftarrow$ GRAY ❑ discover u

$time \leftarrow time + 1$

d[u] $\leftarrow time$

for each $v \in Adj[u]$ ❑ explore (u, v)

  do if $color[v]$ = WHITE

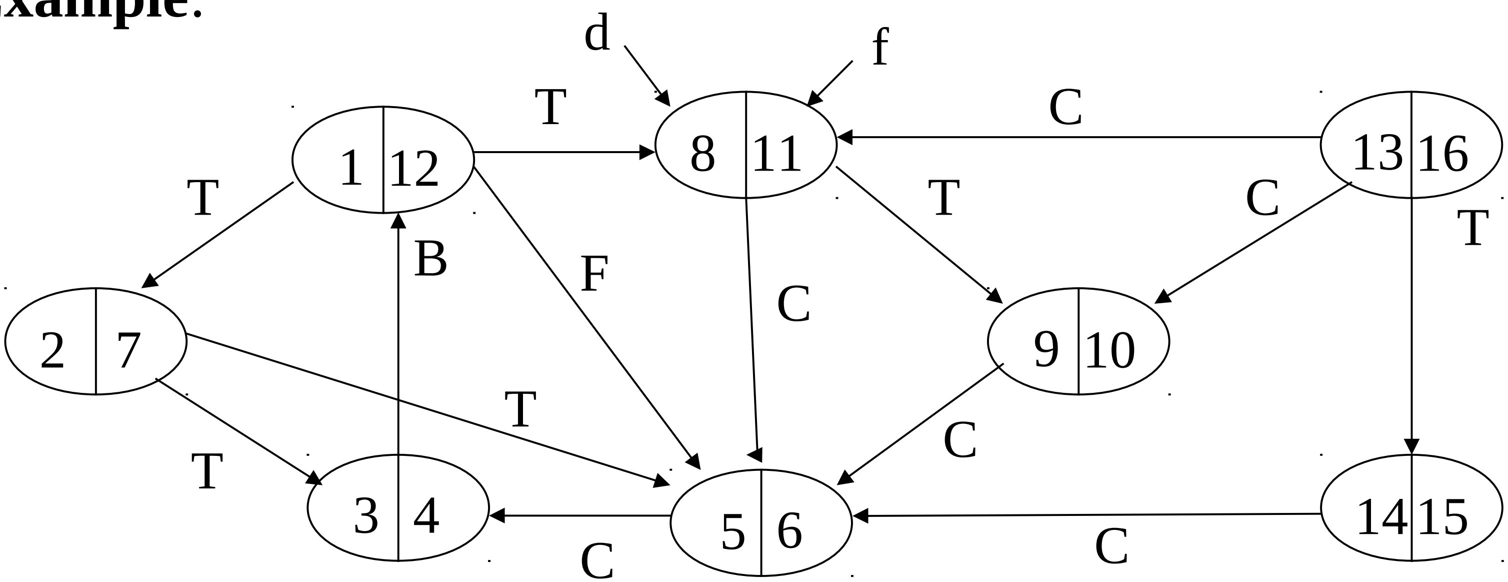
    then DFS - Visit(v)

$color[u] \leftarrow$ BLACK

$time \leftarrow time + 1$

$f[u] \leftarrow time$ ❑ finish u

Example:



Time = $\Theta(V + E)$.

- Similar to BFS analysis.
- Θ, not just O, since guaranteed to examine every vertex and edge.

DFS forms a ***depth-first forest*** comprised of ≥ 1 ***depth-first trees***.

Each tree is made of edges (u, v) such that
u is gray and v is white when (u, v) is explored.

***Theorem* (*Parenthesis theorem*)**

*[*Proof omitted.*]*

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.

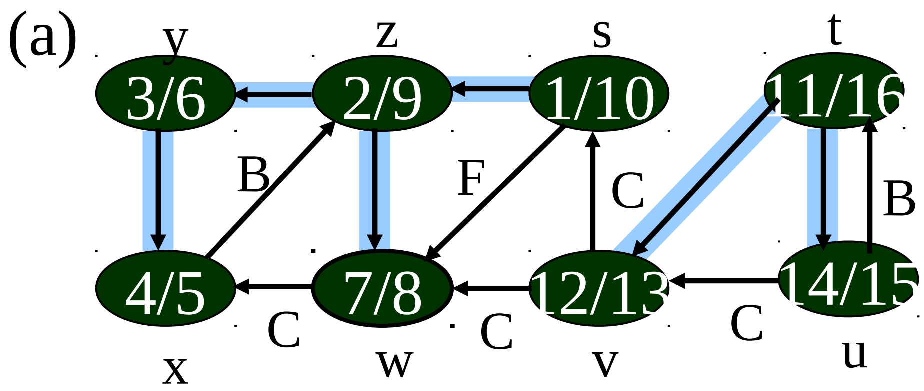
So $d[u] < d[v] < f[u] < f[v]$ *cannot* happen.

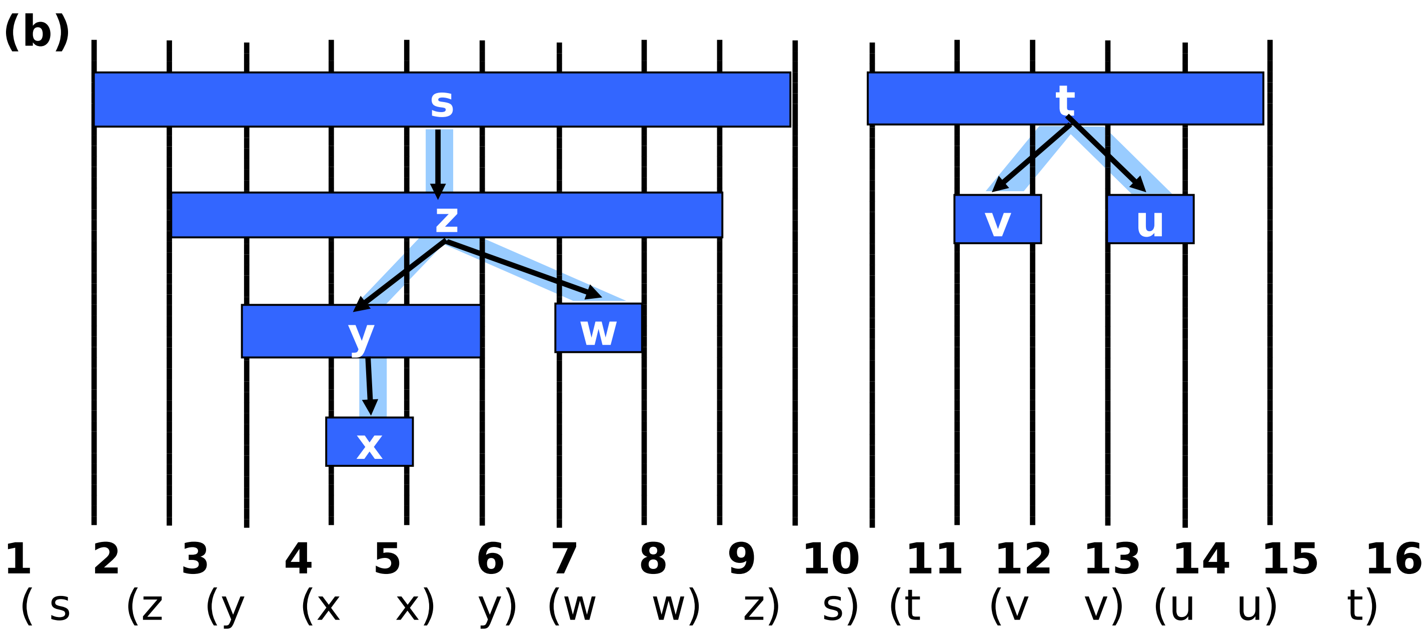
Like parentheses:

- OK: ( ) [ ] ([ ]) [( )]
- Not OK: ([ )] [( ])

Corollary

v is a proper descendant of u if and only if $d[u] < d[v] < f[v] < f[u]$.


(a)
y
z
s
t
3/6
2/9
1/10
11/16
B
F
C
B
4/5
7/8
12/13
14/15
C
C
C
x
w
v
u


(b)
s
t
z
v
u
y
w
x
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
(s (z (y (x x) y) (w w) z) s) (t (v v) (u u) t)

Theorem (White-path theorem)

[Proof omitted.]

v is a descendant of *u* if and only if at time *d*[*u*], there is a path $u \to v$ consisting of only white vertices.
(Except for *u*, which was *just* colored gray.)

Classification of edges

- ***Tree edge*:** in the depth-first forest.
  Found by exploring (*u*, *v*).
- ***Back edge*:** (*u*, *v*), where *u* is a descendant of *v*.
- ***Forward edge*:** (*u*, *v*), where *v* is a descendant of *u*, but not a tree edge.
- ***Cross edge*:** any other edge. Can go between vertices in the same depth-first tree or in different depth-first trees.

In an undirected graph, there may be some ambiguity since (u, v) and (v, u) are the same edge. Classify by the first type above that matches.

Theorem [Proof omitted.]

In DFS of an undirected graph, we get only tree and back edges. No forward or cross edges.

Topological sort

Directed acyclic graph (dag)

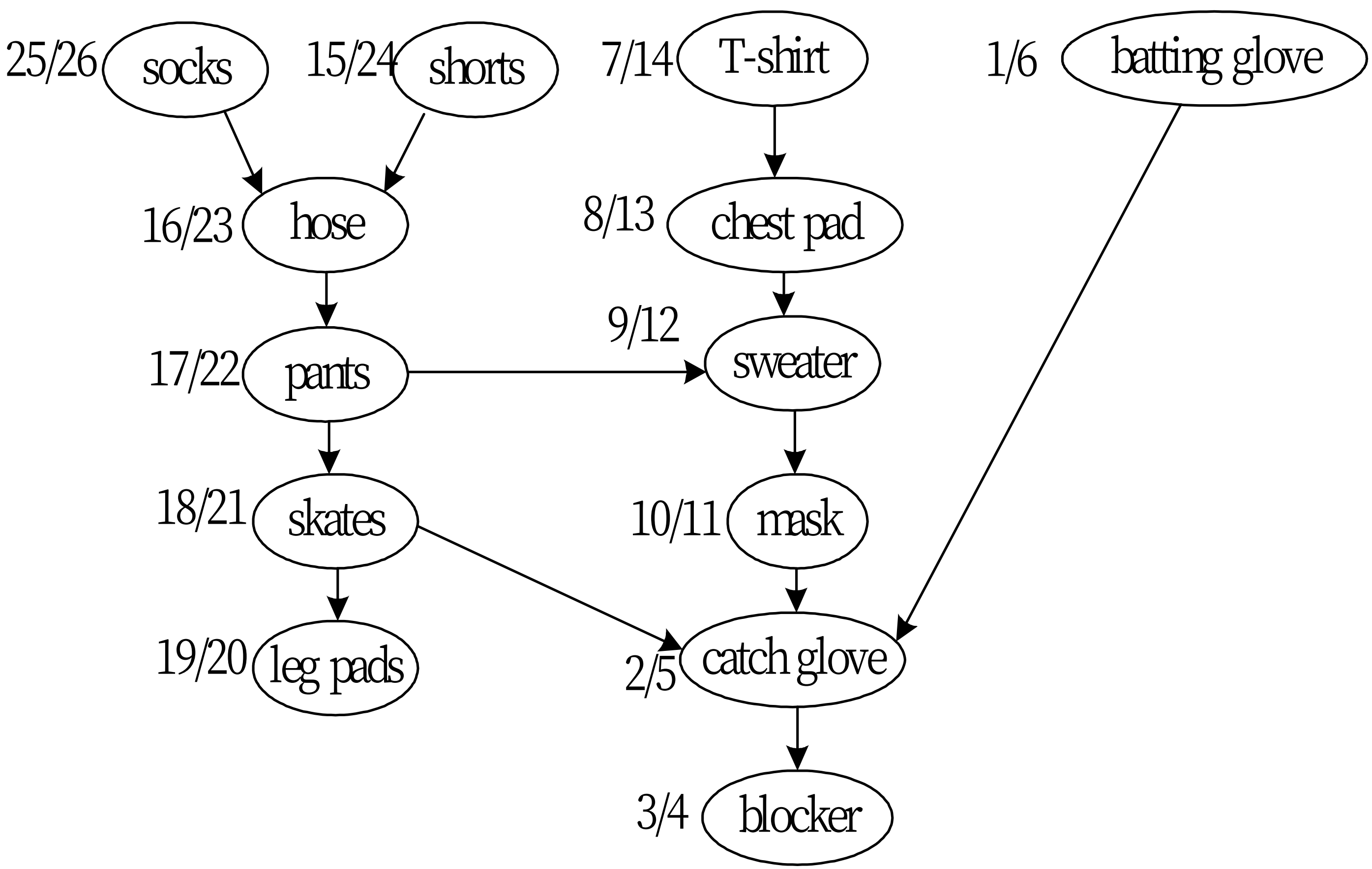
A directed graph with no cycles.

Good for modeling processes and structures that have a ***partial order:***

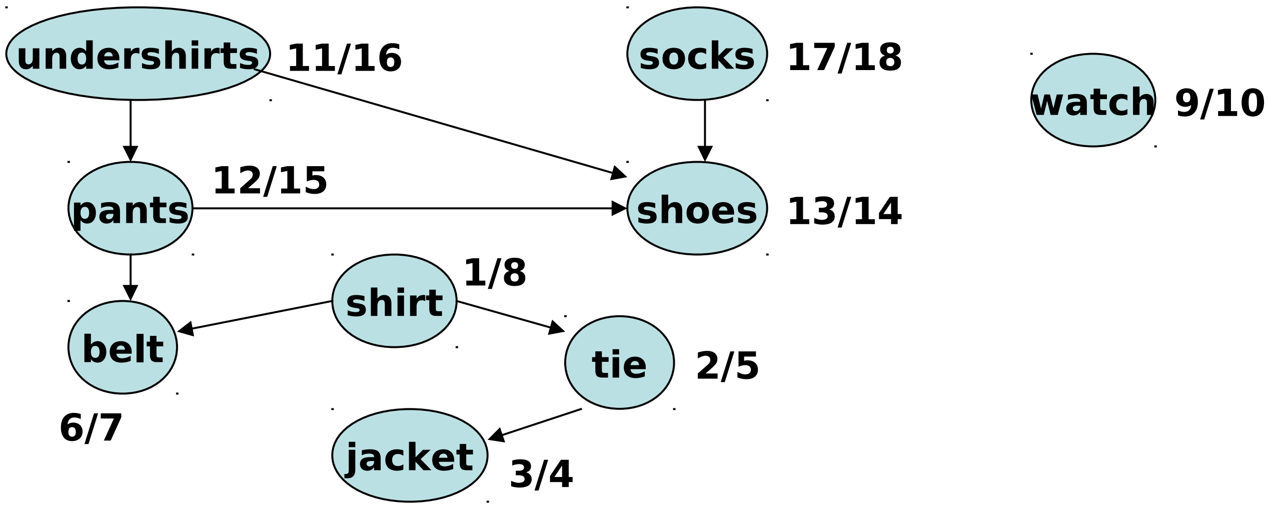
- $a > b$ and $b > c \Rightarrow a > c$.
- But may have a and b such that neither $a > b$ nor $b > a$.

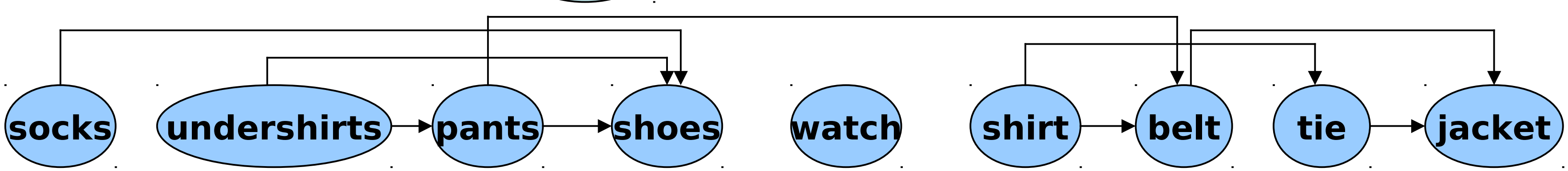
Can always make a ***total order*** (either $a > b$ or $b > a$ for all $a \neq b$) from a partial order. In fact, that's what a topological sort will do.

Example: dag of dependencies for putting on goalie equipment:



- Topological sort
  - 定義：
    A topological sort of a dag G=(*V*, *E*) is a linear ordering of all its vertices. (dag: Directed acyclic graph) 如 edge(*u*, *v*), *u* appears before *v* in the ordering





Lemma

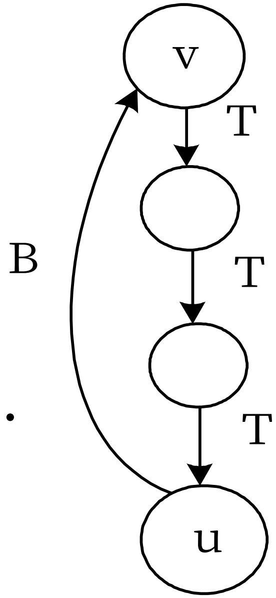
A directed graph G is acyclic if and only if a DFS of G yields no back edges.

Proof ($\Rightarrow$): Show that back edge $\Rightarrow$ cycle.

Suppose there is a back edge (u, v).

Then v is ancestor of u in depth-first forest.

Therefore, there is a path $v \rightarrow u$, so $v \rightarrow u \rightarrow v$ is a cycle.



($\Leftarrow$) :

Show that cycle $\Rightarrow$ back edge.

Suppose G contains cycle c.

At time $d[v]$, vertices of c form a white path $v \rightarrow u$

(since v is the first vertex discovered in c).

By white-path theorem, u is descendant of v in

depth-first forest.

Therefore, (u, v) is a back edge.

Topological sort of a dag: a linear ordering of vertices such that if $(u, v) \in E$, then u appears somewhere before v. (Not like sorting numbers.)

TOPOLOGICAL-SORT(V, E)
call DFS(V, E) to compute finishing times $f[v]$ for all $v \in V$
output vertices in order of *decreasing* finish times

Don't need to sort by finish times.

- Can just output vertices as they're finished and understand that we want the *reverse* of this list.
- Or put them onto the *front* of a linked list as they're finished. When done, the list contains vertices in topological sorted order.

Time: $\Theta(V+E)$

Order:

26	socks
27	shorts
28	hose
29	pants
21	skates
20	leg pads
14	t-shirt
13	chest pad
12	sweater
11	mask
6	batting glove
5	catch glove
4	blocker

Correctness: Just need show if $(u, v) \in E$, then $f[v] < f[u]$.

When we explore (u, v), what are the colors of u and v?

- u is gray.
- Is v gray, too?
  - No, because then v would be ancestor of u.

    $\Rightarrow$ (u, v) is a back edge.

    $\Rightarrow$ contradiction of previous lemma (dag has no back edges).

- Is v white?
  - Then becomes descendant of u.

  By parenthesis theorem, $d[u] < d[v] < \underline{f[v] < f[u]}$.
- Is v black?
  - Then v is already finished.

  Since we're exploring (u, v), we have not yet finished u.

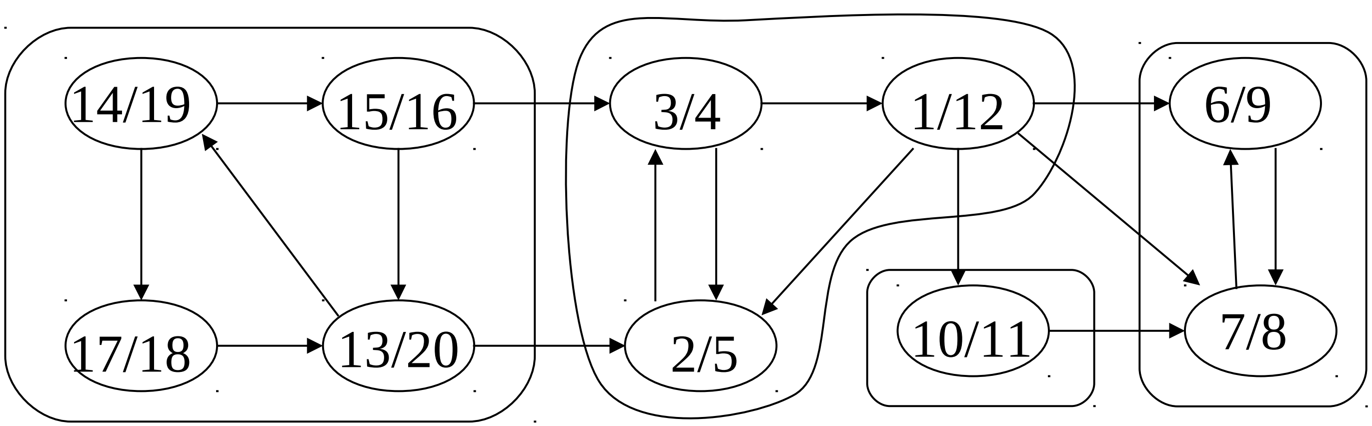
  Therefore, $f[v] < f[u]$.

Strongly connected components

Given directed graph $G = (V,E)$.

A ***strongly connected component (SCC)*** of G is a maximal set of vertices $C \subseteq V$ such that for all $u, v \in C$, both u $\rightarrow$ v and v $\rightarrow$ u.

Example: [Just show SCC's at first. Do DFS a little later.]



Algorithm uses G^T = ***transpose*** of G.

- $G^T = (V, E^T)$, $E^T = \{(u, v) : (v, u) \in E\}$.
- G^T is G with all edges reversed.

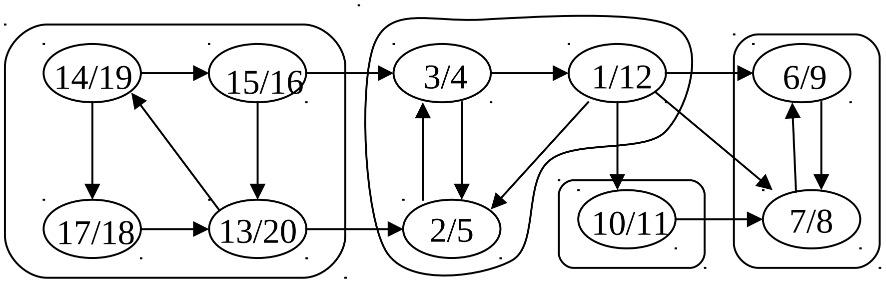
Can create G^T in $\Theta(V + E)$ time if using adjacency lists.

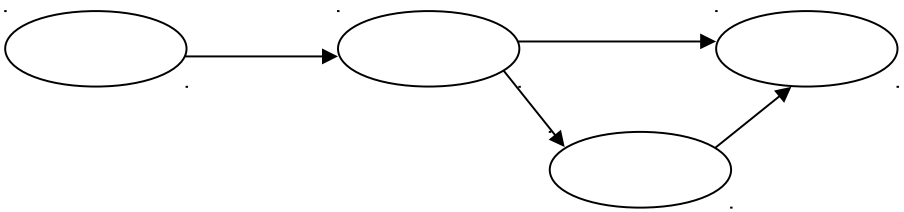
Observation: G and G^T have the same SCC's.
(u and v are reachable from each other in G if and only if reachable from each other in G^T.)

Component graph

- $G^{SCC} = (V^{SCC}, E^{SCC})$.
- V^{SCC} has one vertex for each SCC in G.
- E^{SCC} has an edge if there's an edge between the corresponding SCC's in G.

For our example:





Lemma

G^{SCC} is a dag. More formally, let *C* and *C'* be distinct SCC's in *G*, let $u, v \in C$, $u', v' \in C'$, and suppose there is a path $u \rightarrow u'$ in *G*. Then there cannot also be a path $v' \rightarrow v$ in *G*.

Proof Suppose there is a path $v' \rightarrow v$ in *G*. Then there are paths $u \rightarrow u' \rightarrow v'$ and $v' \rightarrow v \rightarrow u$ in *G*. Therefore, *u* and *v'* are reachable from each other, so they are not in separate SCC's.

SCC(G)

call DFS(G) to compute finishing times $f[u]$ for all u

compute G^T

call DFS(G^T), but in the main loop, consider vertices in order of decreasing $f[u]$

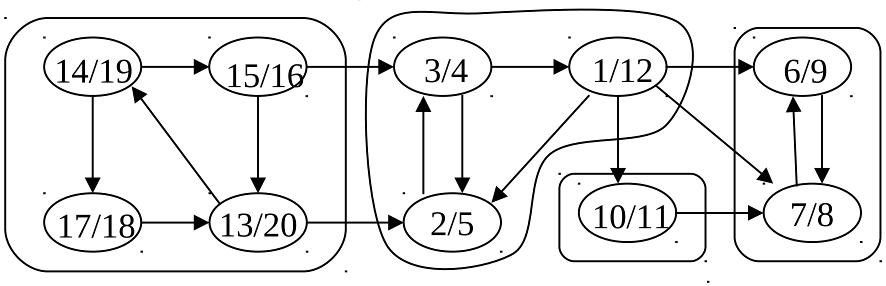
(as computed in first DFS)

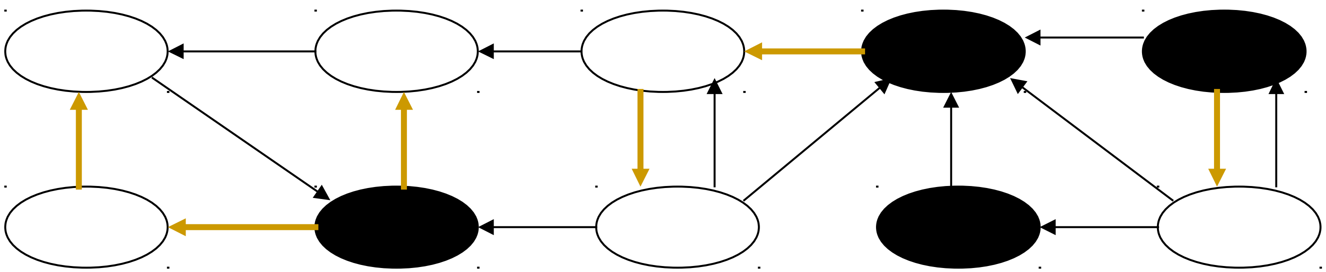
output the vertices in each tree of the depth-first forest formed in second DFS

as a separate SCC

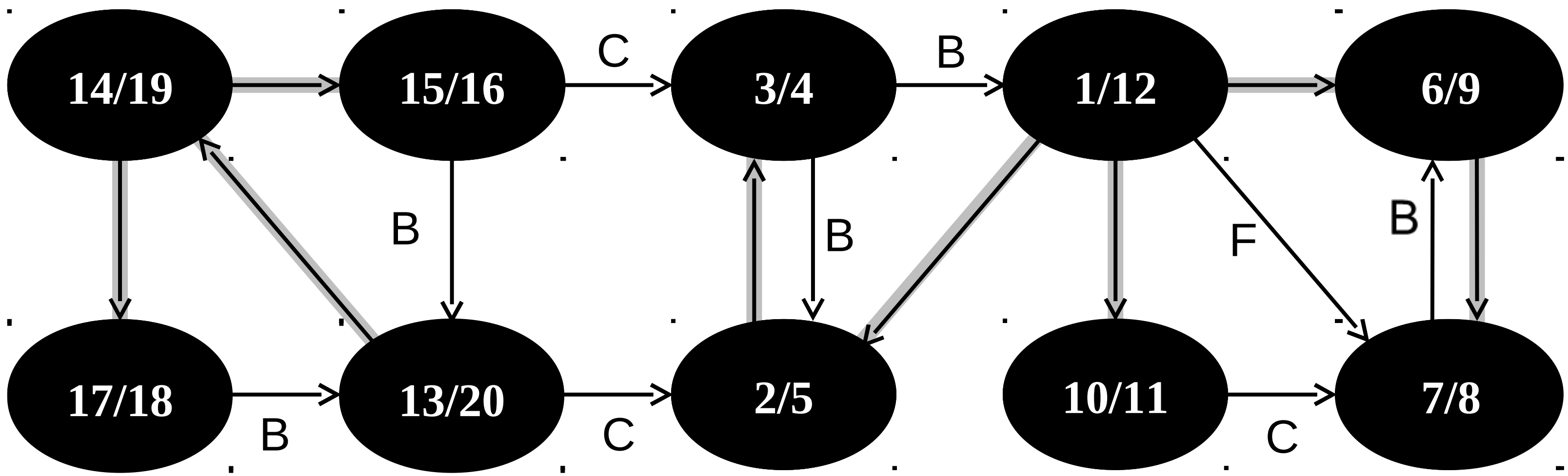
Example:

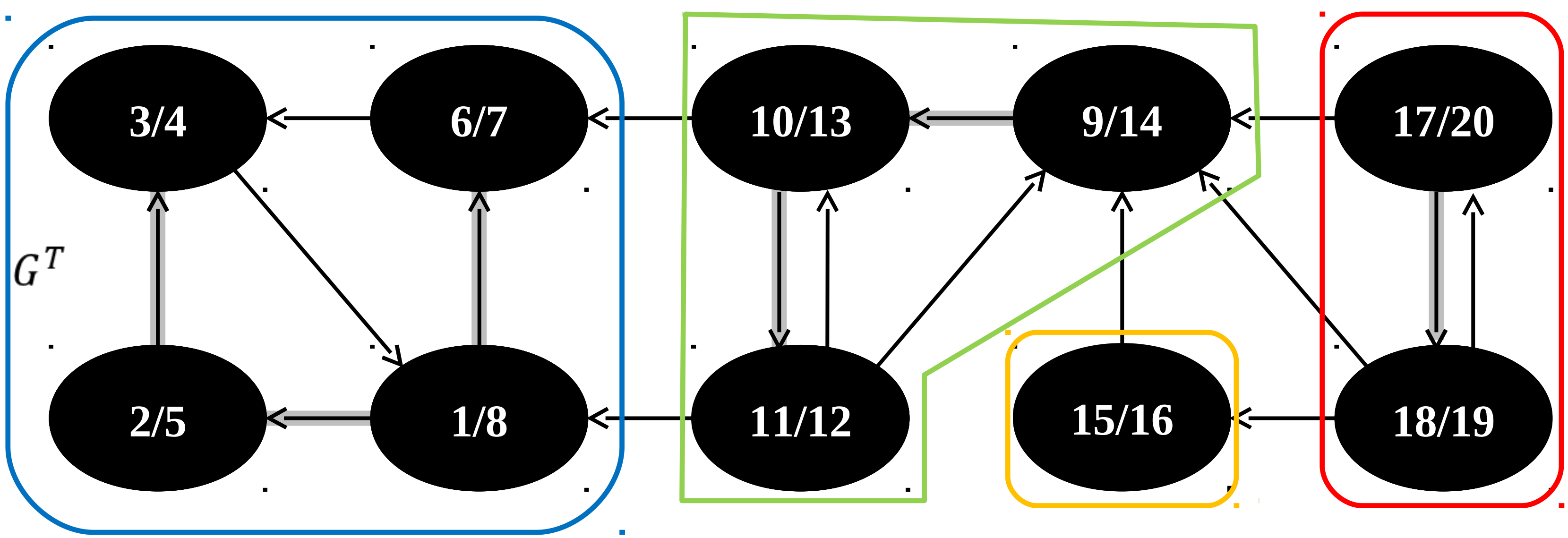
1. Do DFS
2. G^T
3. DFS (roots blackened)





Time: $\Theta(V + E)$.


14/19
15/16
3/4
1/12
6/9
17/18
13/20
2/5
10/11
7/8
C
B
B
B
F
B
B
C
C


G^T
3/4
6/7
10/13
9/14
17/20
2/5
1/8
11/12
15/16
18/19

How can this possibly work?

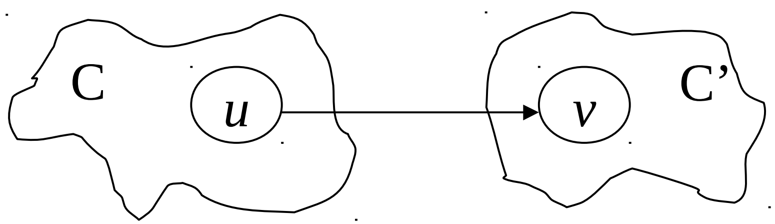
Idea: By considering vertices in second DFS in decreasing order of finishing times from first DFS, we are visiting vertices of the component graph in topological sort order.

To prove that it works, first deal with 2 notational issues:

- Will be discussing $d[u]$ and $f[u]$. These always refer to *first* DFS.
- Extend notation for d and f to sets of vertices $U \subseteq V$:
  - $d(\mathrm{U}) = \min_{u \in \mathrm{U}} \{d[u]\}$ (earliest discovery time)
  - $f(\mathrm{U}) = \max_{u \in \mathrm{U}} \{f[u]\}$ (latest finishing time)

Lemma

Let C and C' be distinct SCC's in $G = (V, E)$. Suppose there is an edge $(u, v) \in E$ such that $u \in C$ and $v \in C'$.



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

Proof

Two cases, depending on which SCC had the first discovered vertex during first DFS.

- If $d(C) < d(C')$, let x be the first vertex discovered in C. At time $d[x]$, all vertices in C and C' are white. Thus, there exist paths of white vertices from x to all vertices in C and C'.

  By the white-path theorem, all vertices in C and C' are descendants of x in depth-first tree.

  By the parenthesis theorem, $f[x] = f(C) > f(C')$.

- If $d(C) > d(C')$, let y be the first vertex discovered in C'. At time $d[y]$, all vertices in C' are white and there is a white path from y to each vertex in C' $\Rightarrow$ all vertices in C' become descendants of y.

Again, $f[y] = f(C')$.

At time $d[y]$, all vertices in C are white.

By earlier lemma, since there is an edge (u, v), we cannot have a path from C' to C.

So no vertex in C is reachable from y.

Therefore, at time $f[y]$, all vertices in C are still white.

Therefore, for all $w \in C$, $f[w] > f[y]$, which implies that $f(C) > f(C')$.

Corollary

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

Proof $(u, v) \in E^T \Rightarrow (v, u) \in E$. Since SCC's of G and G^T are the same, $f(C') > f(C)$.

Corollary

Let C and C' be distinct SCC's in $G = (V, E)$, and suppose that $f(C) > f(C')$.

Then there cannot be an edge from C to C' in G^T.

Proof It's the contrapositive of the previous corollary.

Why the SCC procedure works:

- When we do the second DFS, on G^T, start with SCC C such that $f(C)$ is maximum.
  - The second DFS starts 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.

- The next root chosen in the second DFS is in SCC *C*' such that *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*.

- Each time we choose a root for the second 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.